Décomposition ABC et réduction de Sleathor- Weinfurter

Introduction à l'informatique quantique

Nom : **Lapu**
Prénom : **Matthias**

Tables des matières

1 Écrire la porte X comme décomposition ZYZ

D'après le cours, nous savons que :

$$Y = iXZ \Rightarrow X = -iYZ \tag{1}$$

De plus, nous savons également que :

$$i- = e^{-i\frac{\pi}{2}} \tag{2}$$

Ainsi, nous pouvons remplacer la valeur de $-i$ dans l'expression de base. Nous avons donc :

$$X = e^{-i\frac{\pi}{2}}YZ \tag{3}$$

Rappelons que les portes rotations sont appelés ainsi car une rotation de π dans le même axe est effectué. Cela indique que :

$$\begin{aligned} R_x(\pi) &= X \\ R_y(\pi) &= Y \\ R_z(\pi) &= Z \end{aligned} \tag{4}$$

Y et Z sont présents dans (3), nous avons donc trouvés la décomposition ZYZ en tant que produit de porte paramétrées.

2 Déduire du résultat précédent les porte ABC en appliquant les formules ci-dessus

Nous avons ces 2 formules :

$$\begin{aligned} X &= e^{-i\frac{\pi}{2}}R_y(\pi)R_z(\pi) \\ U &= e^{i\alpha}R_z(\theta_2)R_y(\theta_1)R_z(\theta_0) \end{aligned} \tag{5}$$

Et d'après l'énoncé, la décomposition ABC peut s'écrire ainsi :

$$\begin{aligned} A &= R_z(\theta_2)R_y(\frac{\theta_1}{2}) \\ B &= R_y(\frac{-\theta_1}{2})R_z(\frac{-\theta_0-\theta_2}{2}) \\ C &= R_z(\frac{\theta_0-\theta_2}{2}) \end{aligned} \tag{6}$$

En identifiant avec (5), nous pouvons donc dire que :

$$\begin{aligned} A &= R_y(\frac{\pi}{2}) \\ B &= R_y(\frac{-\pi}{2})R_z(\frac{-\pi}{2}) \\ C &= R_z(\frac{\pi}{2}) \end{aligned} \tag{7}$$

Ainsi, nous avons les valeurs des angles. Finalement :

$$\begin{aligned} \alpha &= \frac{\pi}{2} \\ \theta_2 &= 0 \\ \theta_1 &= \pi \\ \theta_0 &= \pi \end{aligned} \tag{8}$$

N'ayant pas trouvé ces valeurs après 20 min de manière analytique, ils semblaient plus pertinents selon moi d'écrire un code python qui trouvait l'ensemble des bonnes valeurs. L'écriture de ce code m'a pris moins de temps que la réponse analytique. Par la suite, j'ai trouvé le raisonnement, ces valeurs sont bien comprises dans l'ensemble des solutions données par mon code python. Ce code se sert du postulat simple que les valeurs de pi sont des valeurs trouvables par l'être humain (ex : $\frac{\pi}{2}$)

```
######################################################################

# Lapu Matthias

# The goal of this script is to find the values of alpha, theta0, theta1
# and theta2. That will give us the unitary matrix X.
# We will use the formula :
# U = exp(i*alpha) * Rz(theta2) * Ry(theta1) * Rz(theta0)
#
# We will brute force the values. (Yes, It's not elegant, but I find
# it faster than trying to solve the equation with a pen and paper.)

######################################################################

from math import *
import cmath
import numpy as np

# The First question asks us to find an ABC decomposition to find the unitary
#matrix X.

##It's always nice to have a function that converts the values to pi.
# it's just pretty printing.
def convert_to_pi(value):
    if np.isclose(value, 0):
        return "0"
    elif np.isclose(value, np.pi/4):
        return "pi/4"
    elif np.isclose(value, np.pi/2):
        return "pi/2"
    elif np.isclose(value, np.pi):
        return "pi"
    elif np.isclose(value, -np.pi/4):
        return "-pi/4"
```

```
    elif np.isclose(value, -np.pi/2):
        return "-pi/2"
    elif np.isclose(value, -np.pi):
        return "-pi"
    else:
        return str(value)


## We define the Ry and Rz matrices as functions of the angle theta.
#
def Ry_matrix(theta):
    return np.array([[cos(theta/2), -sin(theta/2)],
                     [sin(theta/2), cos(theta/2)]])

def Rz_matrix(theta):
    return np.array([[cmath.exp(-1j*theta/2), 0],
                     [0, cmath.exp(1j*theta/2)]])

## To find the unitary matrix X,
# we use the formula : U = exp(i*alpha) * Rz(theta2) * Ry(theta1) * Rz(theta0)
def U(alpha,theta2,theta1,theta0):
    rz_part_1 = Rz_matrix(theta2)
    ry_part = Ry_matrix(theta1)
    rz_part_2 = Rz_matrix(theta0)
    return cmath.exp(1j*alpha) * np.dot(rz_part_1, np.dot(ry_part, rz_part_2))

## We can now find the values of alpha, theta0, theta1 and theta2
# that will give us the X matrix. Yes, we will brute force it.
# Yes it's O(n^4), but there's only 11 values to check.
def find_X_matrix():
    range_values = [0, pi/2, pi, pi/3, pi/4, pi/6,
                    -pi/2, -pi, -pi/3, -pi/4, -pi/6]
    result = []
    for i in range_values: #alpha
        for j in range_values: #theta2
            for k in range_values: #theta1
                for l in range_values: #theta0
                    if np.allclose(U(i,j,k,l), np.array([[0, 1], [1, 0]])):
                        result.append((i,j,k,l))
    return result



res = find_X_matrix()

print ("The values of alpha, theta2, theta1 and theta0 are :")
for val in res :
    print("-----------------------------------")
    print(convert_to_pi(val[0]), convert_to_pi(val[1]),
          convert_to_pi(val[2]), convert_to_pi(val[3]))

print("-----------------------------------")
print("Number of solutions found : ", len(res))
print("-----------------------------------")
print("Number of iterations : ", 11**4)
```

Les résultats sont :

```
The values of alpha, theta2, theta1 and theta0 are :
-----------------------------------
pi/2 0 pi pi
-----------------------------------
pi/2 0 -pi -pi
-----------------------------------
pi/2 pi/2 -pi -pi/2
-----------------------------------
pi/2 pi -pi 0
-----------------------------------
pi/2 -pi/2 pi pi/2
-----------------------------------
pi/2 -pi pi 0
-----------------------------------
-pi/2 0 pi -pi
-----------------------------------
-pi/2 0 -pi pi
-----------------------------------
-pi/2 pi/2 pi -pi/2
-----------------------------------
-pi/2 pi pi 0
-----------------------------------
-pi/2 -pi/2 -pi pi/2
-----------------------------------
-pi/2 -pi -pi 0
-----------------------------------
Number of solutions found :  12
-----------------------------------
Number of iterations :  14641
```

Nous retrouvons bien les valeurs utilisés plus haut.

3 Faire un programme myqlm, sur un qubit, qui écrit cette sous forme ABC (en utilisant l'AbstractGate pour écrire l'opérateur de phase globale) et vérifier que le résultat est bien celui attendu

```
########################################################################

# Lapu Matthias

# Our goal is to create a circuit that will apply the X gate
# using the ABC decomposition.
# To build the X gate we will use rotation gates, and the formula :
# U = exp(i*alpha) * Rz(theta2) * Ry(theta1) * Rz(theta0)

########################################################################


from qat.lang.AQASM import *
```

```
from qat.qpus import PyLinalg
from qat.lang.AQASM import AbstractGate
import matplotlib.pyplot as plt
import numpy as np
from math import *
import cmath

# We had 12 values to choose from :
# Let's take :
# alpha = pi/2
# theta2 = 0
# theta1 = pi
# theta0 = pi

def Phase_generator(theta):
    return np.array([[np.exp(1j * theta), 0],
    [0, np.exp(1j * theta)]])

GlobalPhase = AbstractGate("Phase", [float], arity=1,
                           matrix_generator=Phase_generator)

## This fonction will build the circuit that applies the ABC decomposition
# with the values we chose. It will normally output the X gate.
# which mean that with 1 qbit set to 0 we should have 1 qbit set to 1 with 100%.
def build_ABC_decomposition_circuit(alpha, theta2, theta1, theta0):

    prog = Program()
    qbits = prog.qalloc(1)


    # Reminder of the formula :
    # U = exp(i*alpha) * Rz(theta2) * Ry(theta1) * Rz(theta0)
    # We must apply it in the reverse order.

    # We apply the Rz(theta2) gate
    prog.apply(RZ(theta0), qbits[0])
    # Then the Ry(theta1) gate
    prog.apply(RY(theta1), qbits[0])
    # Finally the Rz(theta0) gate
    prog.apply(RZ(theta2), qbits[0])

    # We apply the global phase that corresponds to exp(i*alpha)
    prog.apply(GlobalPhase(alpha), qbits[0])

    # Running the circuit
    circuit = prog.to_circ()
    circuit.display()

    job = circuit.to_job()

    linalgqpu = PyLinalg()

    # printing the result
    result = linalgqpu.submit(job)
    l = len(result)
    states = ['']*l
    probabilities= [0]*l

    i=0
    for sample in result:
        print("State",sample.state,"with amplitude",
```

```
                 sample.amplitude,"and probability",
                 round(sample.probability*100,2),"%")
        states[i] = str(sample.state)
        probabilities[i] = round(sample.probability*100,2)
        i = i+1
    plt.bar(states, probabilities, color='skyblue')
    plt.xlabel('States')
    plt.ylabel('Probabilities')
    plt.show()

build_ABC_decomposition_circuit(np.pi/2, 0, np.pi, np.pi)

# As we can see, with a qbit of 0, we have a qbit of 1 with 100% probability.
# Thus we have successfully implemented the X gate with the ABC decomposition.
```

Pour ce programme MyQLM, nous avons :

- Nombre de portes à 1 qubit : 4
- Nombre de CNOT sur 2 qubits : 0

4 Faire un programme myqlm sur 2 qubits qui encode une porte contrôlée basée sur cette porte, grâce à la décomposition ABC

```
#######################################################################

# Lapu Matthias

# The goal is to create a controlled gate that will apply the X gate
# using the ABC decomposition.

#######################################################################

import numpy as np
from math import *
import matplotlib.pyplot as plt
from qat.lang.AQASM import *
from qat.qpus import PyLinalg
from qat.lang.AQASM import AbstractGate


def Phase_generator(theta):
    return np.array([[np.exp(1j * theta), 0],
    [0, np.exp(1j * theta)]])


## This function will build the circuit that applies the ABC decomposition
# with the values we chose. It will normally output a controlled X gate.
# Reminder : a controlled gate works like a if then else statement.
def X_controlled_gate(alpha,theta2,theta1,theta0,qbit_to_1=False):
```

```

    GlobalPhase = AbstractGate("Phase", [float], arity=1,
                               matrix_generator=Phase_generator)

    prog = Program()
    qbits = prog.qalloc(2)

    # If we want to test if the control works,
    # we can change the value of the first qbit before the circuit.
    if qbit_to_1:
        prog.apply(X, qbits[0])

    # Building the circuit

    # Then the C gate
    prog.apply(RZ((theta0-theta2)/2), qbits[1])

    # Then the CNOT
    prog.apply(CNOT,qbits[0], qbits[1])

    #We put B on the first qbit
    prog.apply(RZ((-theta0-theta2)/2), qbits[1])
    prog.apply(RY(-theta1/2), qbits[1])

    # Then the CNOT
    prog.apply(CNOT,qbits[0], qbits[1])

    # We put the Rz rotation on the second qbit
    prog.apply(GlobalPhase(alpha), qbits[0])

    # We put A on the first qbit
    prog.apply(RY(theta1/2), qbits[1])
    prog.apply(RZ(theta2), qbits[1])

    # Running the circuit
    circuit = prog.to_circ()
    circuit.display()

    job = circuit.to_job()

    linalgqpu = PyLinalg()

    # printing the result and showing the probabilities
    result = linalgqpu.submit(job)
    l = len(result)
    states = ['']*l
    probabilities= [0]*l

    i=0
    for sample in result:
        print("State",sample.state,"with amplitude",
                sample.amplitude,"and probability",
                round(sample.probability*100,2),"%")
        states[i] = str(sample.state)
        probabilities[i] = round(sample.probability*100,2)
        i = i+1
    plt.bar(states, probabilities, color='skyblue')
    plt.xlabel('States')
    plt.ylabel('Probabilities')
    plt.show()

```

```
# We said previously that :
# alpha = pi/2
# theta2 = 0
# theta1 = pi
# theta0 = pi

X_controlled_gate(np.pi/2,0,np.pi,np.pi)

# A controlled gate works like a if then else statement.
# We put |00> in input, if the first qbit is 1,
# we apply the X gate on the second qbit.
# By default, both qbits are set to 0, so we should have |00> in output.

# The output is indeed |00> with 100% probability.

# If we change the value of the first qbit to 1, we should have |11> in output.

X_controlled_gate(np.pi/2,0,np.pi,np.pi,True)

# The output is indeed |11> with 100% probability.
```

Pour ce programme MyQLM, nous avons :

- Nombre de portes à 1 qubit : 6
- Nombre de CNOT sur 2 qubits : 2

5 Écrire les racines carrée et quatrième de X sous forme de porte paramétrée

5.1 Racine carrée de la porte X

D'après le cours (slide 72) , la racine carrée de la porte X est :

$$\sqrt{X} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix} \tag{9}$$

D'après le site d'IBM (ici), cette porte se nomme la "SX gate". Nous pouvons donc la représenter sous forme de portes paramétrée. Il semble logique que la porte SX soit la porte paramétré Rx. En effet, rappelons que la porte X réalise une rotation d'un angle π autour de l'axe X, ils semble donc cohérent que sa racine effectue une rotation dans ce sens. Il est encore plus cohérent que celle-ci soit la moitié de la rotation de la porte X soit $\frac{\pi}{2}$. Démontrons donc que :

$$R_x(\frac{\pi}{2}) = \sqrt{X} \tag{10}$$

Pour cela nous utiliserons la formule de la porte paramétré rotation de X.

$$R_x(\alpha) = e^{-i\frac{\alpha}{2}X} = \begin{pmatrix} cos(\frac{\alpha}{2}) & -isin(\frac{\alpha}{2}) \\ -isin(\frac{\alpha}{2}) & cos(\frac{\alpha}{2}) \end{pmatrix} \tag{11}$$

En effet, en remplaçant par $\frac{\pi}{2}$, nous avons :

$$R_x(\frac{\pi}{2}) = \begin{pmatrix} cos(\frac{\pi}{4}) & -isin(\frac{\pi}{4}) \\ -isin(\frac{\pi}{4}) & cos(\frac{\pi}{4}) \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{2}}{2} & -i\frac{\sqrt{2}}{2} \\ -i\frac{\sqrt{2}}{2} & \frac{\sqrt{2}}{2} \end{pmatrix} \tag{12}$$

En factorisant , nous avons bien :

$$R_x(\frac{\pi}{2}) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix} = \sqrt{X} \tag{13}$$

5.2 Racine quatrième de la porte X

On supposera que la racine quatrième correspond à une rotation de $\frac{\pi}{4}$ de l'axe X.

Nous avons donc :

$$R_x(\frac{\pi}{4}) = e^{-i\frac{\pi}{8}X} = X^{\frac{1}{4}} \tag{14}$$

6 En utilisant la racine carrée de X, faire un programme myqlm qui encode une porte doublement contrôlée basée sur cette porte,

```python
#######################################################################

# Lapu Matthias

# The goal is to create a circuit that will apply the X gate
# with a squared gate using the Sleathor Weinfurter reduction

#######################################################################

import cmath
from math import *
import numpy as np
import matplotlib.pyplot as plt
from qat.lang.AQASM import *
from qat.qpus import get_default_qpu
from itertools import product
from scipy.linalg import sqrtm

# Must create all the possible states for 2 bits
# like creating_2n_states(2) will return [[0,0],[0,1],[1,0],[1,1]]
def creating_2n_states(n):
```

```python
    return list(product([0, 1], repeat=n))


# We need to create the X squared gate, because we apply it to 2 qubits
# we need it to be a 4x4 matrix
def X_squared_gate():
    return np.array([[1, 0, 0, 0],
                     [0, 1, 0, 0],
                     [0, 0,np.sqrt(2)/2, -1j*np.sqrt(2)/2],
                     [0, 0, -1j*np.sqrt(2)/2, np.sqrt(2)/2]])

# We will also need the dagger of the X squared gate
# it's the adjoint of the matrix
def X_squared_gate_dagger():
    return X_squared_gate().conj().T

# To build the controlled gate using 3qbits, we need the fourth root
# of the X gate
def X_fourth_gate():
    return sqrtm(X_squared_gate())

def X_fourth_gate_dagger():
    return X_fourth_gate().conj().T

# We will create a generic function that will apply the
# Sleathor Weinfurter reduction, thus we need V and V dagger gates
def Sleathor_Weinfurter_2qubits_X(mat,mat_dagger,state):

    # Starting the program and allocating 3 qubits
    prog = Program()
    qbits = prog.qalloc(3)

    # Creating the initial state
    for i in range(len(state)):
        if state[i] == 1:
            prog.apply(X, qbits[i])

    # Then, we create the V and V dagger gates using the function
    # passed as an argument
    V = AbstractGate("V", [], arity=2, matrix_generator=mat)
    VD = AbstractGate("VD", [], arity=2, matrix_generator=mat_dagger)


    # The Sleathor Weinfurter reduction with 2 qbits is the following :
    # - apply the V gate on the 2nd and 3rd qubits
    # - apply a CNOT gate. The X is on the 2nd qbit, the target is the 1st
    # - apply the V dagger gate on the 2nd and 3rd qubits
    # - apply a CNOT gate. The X is on the 2nd qbit, the target is the 1st
    # - apply the V gate on the 1st and 3rd qubits

    # Building the circuit
    prog.apply(V(), qbits[1], qbits[2])
    prog.apply(CNOT, qbits[0], qbits[1])
    prog.apply(VD(), qbits[1], qbits[2])
    prog.apply(CNOT, qbits[0], qbits[1])
    prog.apply(V(), qbits[0], qbits[2])

    # Running the circuit
    circuit = prog.to_circ()
    # circuit.display()

```

```
    mypylinalgqpu = get_default_qpu()

    job = circuit.to_job()
    result = mypylinalgqpu.submit(job)

    # Plotting the results and the percentage of each state :w!
    l = len(result)
    states = ['']*l
    probabilities= [0]*l

    i=0
    for sample in result:
        print("State",sample.state,"with amplitude",
                sample.amplitude,"and probability",
                round(sample.probability*100,2),"%")
        states[i] = str(sample.state)
        probabilities[i] = round(sample.probability*100,2)
        i = i+1
    plt.bar(states, probabilities, color='skyblue')
    plt.xlabel('States')
    plt.ylabel('Probabilities')
    plt.show()

## Same goal, function kind of similar, but with 3 qubits
def Sleathor_Weinfurter_3qubits_X(mat,mat_dagger,state):

    # Starting the program and allocating 3 qubits
    prog = Program()
    qbits = prog.qalloc(4)

    # Creating the initial state
    for i in range(len(state)):
        if state[i] == 1:
            prog.apply(X, qbits[i])

    # Then, we create the V and V dagger gates using the function
    # passed as an argument
    V = AbstractGate("V", [], arity=2, matrix_generator=mat)
    VD = AbstractGate("VD", [], arity=2, matrix_generator=mat_dagger)


    # The Sleathor Weinfurter reduction with 3 qbits is the following :
    # - apply the V gate on the 1st qbit
    # - apply a CNOT gate. The X is on the 2nd qbit, the target is the 1st
    # - apply a VD gate on the 2nd qbit
    # - apply a CNOT gate. The X is on the 2nd qbit, the target is the 1st
    # - apply the V gate on the 2nd qbit
    # - apply a CNOT gate. The X is on the 3rd qbit, the target is the 2nd
    # - apply a VD gate on the 3rd qbit
    # - apply a CNOT gate. The X is on the 3rd qbit, the target is the 1st
    # - apply the V gate on the 3nd qbit
    # - apply a CNOT gate. The X is on the 3rd qbit, the target is the 2nd
    # - apply a VD gate on the 3rd qbit
    # - apply a CNOT gate. The X is on the 3rd qbit, the target is the 1st
    # - apply the V gate on the 3nd qbit

    # Building the circuit
    prog.apply(V(),qbits[0],qbits[3])
    prog.apply(CNOT, qbits[0], qbits[1])
    prog.apply(VD(), qbits[1], qbits[3])
    prog.apply(CNOT, qbits[0], qbits[1])
```

```
    prog.apply(V(), qbits[1], qbits[3])
    prog.apply(CNOT, qbits[1], qbits[2])
    prog.apply(VD(), qbits[2], qbits[3])
    prog.apply(CNOT, qbits[0], qbits[2])
    prog.apply(V(), qbits[2], qbits[3])
    prog.apply(CNOT, qbits[1], qbits[2])
    prog.apply(VD(), qbits[2], qbits[3])
    prog.apply(CNOT, qbits[0], qbits[2])
    prog.apply(V(), qbits[2], qbits[3])


    # Running the circuit
    circuit = prog.to_circ()
    # circuit.display()

    mypylinalgqpu = get_default_qpu()

    job = circuit.to_job()
    result = mypylinalgqpu.submit(job)

    # Plotting the results and the percentage of each state :w!
    l = len(result)
    states = ['']*l
    probabilities= [0]*l

    i=0
    for sample in result:
        print("State",sample.state,"with amplitude",
                sample.amplitude,"and probability",
                round(sample.probability*100,2),"%")
        states[i] = str(sample.state)
        probabilities[i] = round(sample.probability*100,2)
        i = i+1
    plt.bar(states, probabilities, color='skyblue')
    plt.xlabel('States')
    plt.ylabel('Probabilities')
    plt.show()


## To test if the X squared gate works,
# if the first qbit is one, the second qbit should switch
# otherwise, the second qbit should stay the same
def testing_X_sq2_sq4(state,th_gate):

    # The test is only for the squared gate and the fourth gate
    if th_gate != 2 and th_gate != 4:
        print("The th_gate must be 2 or 4")
        return

    prog = Program()
    qbits = prog.qalloc(2)

    # Creating the gate depending on the th_gate
    if th_gate == 2:
        V = AbstractGate("V", [], arity=2, matrix_generator=X_squared_gate)
    else:
        V = AbstractGate("V", [], arity=2, matrix_generator=X_fourth_gate)


    # Creating the initial state
    for i in range(len(state)):
```

```
        if state[i] == 1:
            prog.apply(X, qbits[i])

    # Applying the gate th_gate times should always return the X gate
    for i in range(th_gate):
        prog.apply(V(), qbits[0], qbits[1])

    # Running the circuit
    circuit = prog.to_circ()
    # circuit.display()

    mypylinalgqpu = get_default_qpu()

    job = circuit.to_job()
    result = mypylinalgqpu.submit(job)

    # printing the percentage of each state
    for sample in result:
        print("State",sample.state,", probability",
                round(sample.probability*100,2),"%")


def main():

    # Testing the X squared gate
    for state in creating_2n_states(2):
        print("Testing the X squared gate with the state : ",state)
        testing_X_sq2_sq4(state,2)
        # looks ok

    print("-------------------------------------------------")

    # Testing the X fourth gate
    for state in creating_2n_states(2):
        print("Testing the X fourth gate with the state : ",state)
        testing_X_sq2_sq4(state,4)
        # looks ok
    print("-------------------------------------------------")

    # Testing the Sleathor Weinfurter reduction with 2 qubits
    for state in creating_2n_states(2):
        print("Testing the Sleathor Weinfurter reduction with the state : ",state)
        Sleathor_Weinfurter_2qubits_X(X_squared_gate, X_squared_gate_dagger, state)

    print("-------------------------------------------------")

    # Testing the Sleathor Weinfurter reduction with 3 qubits
    for state in creating_2n_states(3):
        print("Testing the Sleathor Weinfurter reduction with the state : ",state)
        Sleathor_Weinfurter_3qubits_X(X_fourth_gate, X_fourth_gate_dagger, state)

main()
```

Pour ce programme MyQLM, nous avons :

- Nombre de portes à 1 qubit : 0
- Nombre de CNOT sur 2 qubits : 2

- Nombre de porte SX (dagger ou non) controlé : 3

7 En utilisant la racine carrée de X, faire un programme myqlm qui encode une porte doublement contrôlée basée sur cette porte,

Le programme donnée plus haut s'occupe également de faire la décomposition avec 4 qubits pour la racine quatrième de X.

Pour ce programme MyQLM, nous avons :

- Nombre de portes à 1 qubit : 0
- Nombre de CNOT sur 2 qubits : 6
- Nombre de porte quatrième de X (dagger ou non) controlé : 7

8 Écrire un programme générique qui écrit une porte contrôlée sur n qubits à base de NOT contrôlée sur plusieurs qubits. Écrivez en particulier une sous-routine qui implémente la "CNOT à n qubits de contrôle" de manière récursive.

Ce code ne fonctionne pas.

Afin de créer une porte CNOT controllé, nous utilisons la propriété par bloc de la matrice afin de la crée récursivement, en effet, nous avons que

$$C_xNOT = \begin{pmatrix} I & 0 \\ 0 & CX \end{pmatrix} \tag{15}$$

Ainsi, il est possible de créer une matrice par bloc peu importe la taille demandé avec la porte CX avec une augmentation de matrice. Cela est construit récursivement dans la fonction "recursive_CNOT" (ligne 63 du code ci-dessous.).

Ensuite, afin d'écrire un programme générique, il est nécessaire d'appliquer de manière générique le circuit décrit dans la 1.6 de l'énoncé. Pour ce faire, nous avons besoin de la porte V, car nous souhaitons construire la porte X, nous utiliserons donc la racine carrée de la matrice X. La V dagger, sera donc l'adjointe de cette matrice. Ensuite, le circuit décrit utilise une porte V contrôlant plusieurs qubits. Cette porte serait alors construite récursivement à l'aide de la réduction de Sleathor-Weinfurter pour contrôlant 2 qbits et 3 qbits implémenté plus haut. Lors de cette récurrence, la porte V serait donc vue comme une porte U ou la racine quatrième serait utilisé comme porte V afin de la résoudre. Le code proposé, possède des erreurs de syntaxes que je n'ai pas réussi à régler. Dans la documentation, je n'ai pas trouvé de moyen de faire des portes abstraites sur un nombre variables de qbits. Je n'ai donc pas pu continuer mon implémentation.

```
######################################################################

# Lapu Matthias

# The goal is to create a controled gate on n qbits, using NOT that
# controls multiple qbits.
######################################################################

from math import *
import numpy as np
from qat.qpus import PyLinalg
from qat.lang.AQASM import AbstractGate
from qat.lang.AQASM import *
import matplotlib.pyplot as plt
from scipy.linalg import sqrtm


# We need to create the X squared gate, because we apply it to 2 qubits
# we need it to be a 4x4 matrix
def X_squared_gate():
    return np.array([[1, 0, 0, 0],
                     [0, 1, 0, 0],
                     [0, 0,np.sqrt(2)/2, -1j*np.sqrt(2)/2],
                     [0, 0, -1j*np.sqrt(2)/2, np.sqrt(2)/2]])

# We will also need the dagger of the X squared gate
# it's the adjoint of the matrix
def X_squared_gate_dagger():
    return X_squared_gate().conj().T

# To build the controlled gate using 3qbits, we need the fourth root
# of the X gate
def X_fourth_gate():
    return sqrtm(X_squared_gate())

def X_fourth_gate_dagger():
    return X_fourth_gate().conj().T

## This function will create the CNOT matrix
# Reminder : the X gate matrix is : [[0,1],[1,0]]
# The identity matrix is : [[1,0],[0,1]]
# The CX matrix can be written as a block matrix like this :
# [[I, 0], [0, X]]
def CNOT_matrix():
    # we need to create 2 matrix of size (n*n), the identity matrix
    # and the X gate matrix
    # Identity matrix
    I = np.identity(2)
    Zero = np.zeros((2,2))
    X = np.flip(np.identity(2),0)

    # Creating the CNOT matrix
    CNOT = np.block([[I, Zero],
                    [Zero, X]])
    return CNOT

## The toffoli gate, which is a controlled not gate on 3 qbits can be
# written as a block matrix like this :
# [[I, 0], [0, CX]] with CX the CNOT matrix written above
# Thus, we can write a recursive function that will create a CNOT gate
```

```
# on n qbits
# n=3 should return the Toffoli gate
def recursive_CNOT(n):
    if n == 2:
        return CNOT_matrix()
    else:
        return np.block([[np.identity(2**(n-1)), np.zeros((2**(n-1),2**(n-1)))],
                        [np.zeros((2**(n-1),2**(n-1))), recursive_CNOT(n-1)]])


## Build the Sleathor Weinfurter reduction with 2 qbits :w!
def Sleathor_Weinfurter_2qubits_X(prog,mat,mat_dagger,starting_qbit:int):

    V = AbstractGate("V", [], arity=2, matrix_generator=mat)
    VD = AbstractGate("VD", [], arity=2, matrix_generator=mat_dagger)

    final_qbit = starting_qbit+3 # we must hop one of the qbit
    # Building the circuit
    prog.apply(V(), qbits[starting_qbit+1], qbits[final_qbit])
    prog.apply(CNOT, qbits[starting_qbit], qbits[starting_qbit+1])
    prog.apply(VD(), qbits[starting_qbit+1], qbits[final_qbit])
    prog.apply(CNOT, qbits[starting_qbit], qbits[starting_qbit+1])
    prog.apply(V(), qbits[starting_qbit], qbits[final_qbit])


## Build the Sleathor Weinfurter reduction with 3 qbits
# explained in Sleahtor_Weinfurter_reduction.py
def Sleathor_Weinfurter_3qubits_X(prog,mat,mat_dagger,
                                  starting_qbit:int,):

    # the starting qbit is (from the top) where the firts qbit available
    # for the scope of the reduction is located

    # Then, we create the V and V dagger gates using the function
    # passed as an argument
    V = AbstractGate("V", [], arity=2, matrix_generator=mat)
    VD = AbstractGate("VD", [], arity=2, matrix_generator=mat_dagger)

    final_qbit = starting_qbit+4    # Building the circuit

    # Building the circuit
    prog.apply(V(),qbits[starting_qbit],qbits[final_qbit])
    prog.apply(CNOT, qbits[starting_qbit], qbits[starting_qbit+1])
    prog.apply(VD(), qbits[starting_qbit+1], qbits[final_qbit])
    prog.apply(CNOT, qbits[starting_qbit], qbits[starting_qbit+1])
    prog.apply(V(), qbits[starting_qbit+1], qbits[final_qbit])
    prog.apply(CNOT, qbits[starting_qbit+1], qbits[starting_qbit+2])
    prog.apply(VD(), qbits[starting_qbit+2], qbits[final_qbit])
    prog.apply(CNOT, qbits[starting_qbit], qbits[starting_qbit+2])
    prog.apply(V(), qbits[starting_qbit+2], qbits[final_qbit])
    prog.apply(CNOT, qbits[starting_qbit+1], qbits[starting_qbit+2])
    prog.apply(VD(), qbits[starting_qbit+2], qbits[final_qbit])
    prog.apply(CNOT, qbits[starting_qbit], qbits[starting_qbit+2])
    prog.apply(V(), qbits[starting_qbit+2], qbits[final_qbit])




# To create a controlled gate on m qbits,
# we need a recursive CNOT to apply to m-2 qbits located at the m-1 qbit
# (so just 1 before the last qbit)
```

```
# We also need the V controlled gate on m-2 qbits located at the m qbit

def controlled_gate_n_qbits_X(m,mat,squared_mat,mat_dagger,squared_mat_dagger):
    prog = Program()
    qbits = prog.qalloc(m)

    # We need to create the controlled CNOT gate on m-2 qbits

    CCNOT = AbstractGate("CCNOT", [], arity=(m-1),
                         matrix_generator=recursive_CNOT(m-2))

    # We also need to create the V gate and the V dagger gate
    V = AbstractGate("V", [], arity=2, matrix_generator=mat)
    VD = AbstractGate("VD", [], arity=2, matrix_generator=mat_dagger)

    # Then we apply the V gate on the m-2 and m-1 qbits

    prog.apply(V(), qbits[m-2], qbits[m-1])

    # the list of qbits that the cnot gate will control
    qbits_to_control = [qbits[i] for i in range(m-2)]

    # Then we apply the CNOT gate on the m-2 qbits to control the m-3 qbit
    # The syntax error comes from here, I did not find a way to fix it
    # The documentation only talks about gate with a clear number of qbits
    # I did not find any example of a gate that controls variable qbits
    prog.apply(CCNOT,qbits_to_control, qbits[m-1])
    prog.apply(VD(), qbits[m-2], qbits[m-1])
    prog.apply(CCNOT,qbits_to_control, qbits[m-1])

    # The only thing we need is to create the V controlled gate on m-2 qbits
    # It can be seen as a U gate, in this case the V and VD gate of this one
    # would be the fourth root of the X gate

    if m == 4:
        Sleathor_Weinfurter_2qubits_X(prog,squared_mat,squared_mat_dagger,m-1)
    elif m == 5:
        Sleathor_Weinfurter_3qubits_X(prog,squared_mat,squared_mat_dagger,m-2)
    else:
        controlled_gate_n_qbits_X(m-1,mat,squared_mat,
                                  mat_dagger,squared_mat_dagger)

     # Running the circuit
    circuit = prog.to_circ()
    circuit.display()

    mypylinalgqpu = get_default_qpu()

    job = circuit.to_job()
    result = mypylinalgqpu.submit(job)

    # Plotting the results and the percentage of each state :w!
    l = len(result)
    states = ['']*l
    probabilities= [0]*l

    i=0
    for sample in result:
        print("State",sample.state,"with amplitude",
                sample.amplitude,"and probability",
                round(sample.probability*100,2),"%")
```

```
        states[i] = str(sample.state)
        probabilities[i] = round(sample.probability*100,2)
        i = i+1
    plt.bar(states, probabilities, color='skyblue')
    plt.xlabel('States')
    plt.ylabel('Probabilities')
    plt.show()

controlled_gate_n_qbits_X(5,X_squared_gate,X_fourth_gate,
                          X_squared_gate_dagger,X_fourth_gate_dagger)
```