

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 | 3 |
| 2 | Déduire du résultat précédent les porte ABC en appliquant les formules ci-dessus | 3 |
| 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 . | 6 |
| 4 | Faire un programme myqlm sur 2 qubits qui encode une porte contrôlée basée sur cette porte, grâce à la décomposition ABC . | 8 |
| 5 | Écrire les racines carrée et quatrième de X sous forme de porte paramétrée | 10 |
| 5.1 | Racine carrée de la porte X | 10 |
| 5.2 | Racine quatrième de la porte X | 11 |
| 6 | En utilisant la racine carrée de X , faire un programme myqlm qui encode une porte doublement contrôlée basée sur cette porte, | 11 |
| 7 | En utilisant la racine carrée de X , faire un programme myqlm qui encode une porte doublement contrôlée basée sur cette porte, | 16 |
| 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. | 16 |

1 Écrire la porte X comme décomposition ZYZ

D'après le cours, nous savons que :

$$Y = iXZ \Rightarrow X = -iYZ \quad (1)$$

De plus, nous savons également que :

$$-i = e^{-i\frac{\pi}{2}} \quad (2)$$

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

$$X = e^{-i\frac{\pi}{2}}YZ \quad (3)$$

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

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

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

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

Nous avons ces 2 formules :

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

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

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

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

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

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 : $\frac{\pi}{2}$)

```

1 #####
2
3 # Lapu Matthias
4
5 # The goal of this script is to find the values of alpha, theta0, theta1
6 # and theta2. That will give us the unitary matrix X.
7 # We will use the formula :
8 # U = exp(i*alpha) * Rz(theta2) * Ry(theta1) * Rz(theta0)
9 #
10 # We will brute force the values. (Yes, It's not elegant, but I find
11 # it faster than trying to solve the equation with a pen and paper.)
12
13 #####
14
15 from math import *
16 import cmath
17 import numpy as np
18
19 # The First question asks us to find an ABC decomposition to find the unitary
20 #matrix X.
21
22 ##It's always nice to have a function that converts the values to pi.
23 # it's just pretty printing.
24 def convert_to_pi(value):
25     if np.isclose(value, 0):
26         return "0"
27     elif np.isclose(value, np.pi/4):
28         return "pi/4"
29     elif np.isclose(value, np.pi/2):
30         return "pi/2"
31     elif np.isclose(value, np.pi):
32         return "pi"
33     elif np.isclose(value, -np.pi/4):
34         return "-pi/4"

```

```

35     elif np.isclose(value, -np.pi/2):
36         return "-pi/2"
37     elif np.isclose(value, -np.pi):
38         return "-pi"
39     else:
40         return str(value)
41
42
43 ## We define the Ry and Rz matrices as functions of the angle theta.
44 #
45 def Ry_matrix(theta):
46     return np.array([[cos(theta/2), -sin(theta/2)],
47                     [sin(theta/2), cos(theta/2)]])
48
49 def Rz_matrix(theta):
50     return np.array([[cmath.exp(-1j*theta/2), 0],
51                     [0, cmath.exp(1j*theta/2)]])
52
53 ## To find the unitary matrix X,
54 # we use the formula :  $U = \exp(i*\alpha) * Rz(\theta_2) * Ry(\theta_1) * Rz(\theta_0)$ 
55 def U(alpha, theta2, theta1, theta0):
56     rz_part_1 = Rz_matrix(theta2)
57     ry_part = Ry_matrix(theta1)
58     rz_part_2 = Rz_matrix(theta0)
59     return cmath.exp(1j*alpha) * np.dot(rz_part_1, np.dot(ry_part, rz_part_2))
60
61 ## We can now find the values of alpha, theta0, theta1 and theta2
62 # that will give us the X matrix. Yes, we will brute force it.
63 # Yes it's  $O(n^4)$ , but there's only 11 values to check.
64 def find_X_matrix():
65     range_values = [0, pi/2, pi, pi/3, pi/4, pi/6,
66                    -pi/2, -pi, -pi/3, -pi/4, -pi/6]
67     result = []
68     for i in range_values: #alpha
69         for j in range_values: #theta2
70             for k in range_values: #theta1
71                 for l in range_values: #theta0
72                     if np.allclose(U(i,j,k,l), np.array([[0, 1], [1, 0]])):
73                         result.append((i,j,k,l))
74     return result
75
76
77
78 res = find_X_matrix()
79
80 print ("The values of alpha, theta2, theta1 and theta0 are :")
81 for val in res :
82     print("-----")
83     print(convert_to_pi(val[0]), convert_to_pi(val[1]),
84           convert_to_pi(val[2]), convert_to_pi(val[3]))
85
86 print("-----")
87 print("Number of solutions found : ", len(res))
88 print("-----")
89 print("Number of iterations : ", 11**4)

```

Les résultats sont :

```
1 The values of alpha, theta2, theta1 and theta0 are :
```

```
2 -----
3 pi/2 0 pi pi
4 -----
5 pi/2 0 -pi -pi
6 -----
7 pi/2 pi/2 -pi -pi/2
8 -----
9 pi/2 pi -pi 0
10 -----
11 pi/2 -pi/2 pi pi/2
12 -----
13 pi/2 -pi pi 0
14 -----
15 -pi/2 0 pi -pi
16 -----
17 -pi/2 0 -pi pi
18 -----
19 -pi/2 pi/2 pi -pi/2
20 -----
21 -pi/2 pi pi 0
22 -----
23 -pi/2 -pi/2 -pi pi/2
24 -----
25 -pi/2 -pi -pi 0
26 -----
27 Number of solutions found : 12
28 -----
29 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

```
1 #####
2
3 # Lapu Matthias
4
5 # Our goal is to create a circuit that will apply the X gate
6 # using the ABC decomposition.
7 # To build the X gate we will use rotation gates, and the formula :
8 # U = exp(i*alpha) * Rz(theta2) * Ry(theta1) * Rz(theta0)
9
10 #####
11
12
13 from qat.lang.AQASM import *
```

```

14 from qat.qpus import PyLinalg
15 from qat.lang.AQASM import AbstractGate
16 import matplotlib.pyplot as plt
17 import numpy as np
18 from math import *
19 import cmath
20
21 # We had 12 values to choose from :
22 # Let's take :
23 # alpha = pi/2
24 # theta2 = 0
25 # theta1 = pi
26 # theta0 = pi
27
28 def Phase_generator(theta):
29     return np.array([[np.exp(1j * theta), 0],
30                     [0, np.exp(1j * theta)]])
31
32 GlobalPhase = AbstractGate("Phase", [float], arity=1,
33                             matrix_generator=Phase_generator)
34
35 ## This fonction will build the circuit that applies the ABC decomposition
36 # with the values we chose. It will normally output the X gate.
37 # which mean that with 1 qbit set to 0 we should have 1 qbit set to 1 with 100%.
38 def build_ABC_decomposition_circuit(alpha, theta2, theta1, theta0):
39
40     prog = Program()
41     qbits = prog.qalloc(1)
42
43
44     # Reminder of the formula :
45     #  $U = \exp(i\alpha) * R_z(\theta_2) * R_y(\theta_1) * R_z(\theta_0)$ 
46     # We must apply it in the reverse order.
47
48     # We apply the  $R_z(\theta_2)$  gate
49     prog.apply(RZ(theta2), qbits[0])
50     # Then the  $R_y(\theta_1)$  gate
51     prog.apply(RY(theta1), qbits[0])
52     # Finally the  $R_z(\theta_0)$  gate
53     prog.apply(RZ(theta0), qbits[0])
54
55     # We apply the global phase that corresponds to  $\exp(i\alpha)$ 
56     prog.apply(GlobalPhase(alpha), qbits[0])
57
58     # Running the circuit
59     circuit = prog.to_circ()
60     circuit.display()
61
62     job = circuit.to_job()
63
64     linalgqpu = PyLinalg()
65
66     # printing the result
67     result = linalgqpu.submit(job)
68     l = len(result)
69     states = ['']*l
70     probabilities= [0]*l
71
72     i=0
73     for sample in result:
74         print("State", sample.state, "with amplitude",

```

```

75         sample.amplitude,"and probability",
76         round(sample.probability*100,2),"%")
77     states[i] = str(sample.state)
78     probabilities[i] = round(sample.probability*100,2)
79     i = i+1
80     plt.bar(states, probabilities, color='skyblue')
81     plt.xlabel('States')
82     plt.ylabel('Probabilities')
83     plt.show()
84
85 build_ABC_decomposition_circuit(np.pi/2, 0, np.pi, np.pi)
86
87 # As we can see, with a qbit of 0, we have a qbit of 1 with 100% probability.
88 # 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

```

1 #####
2
3 # Lapu Matthias
4
5 # The goal is to create a controlled gate that will apply the X gate
6 # using the ABC decomposition.
7
8 #####
9
10 import numpy as np
11 from math import *
12 import matplotlib.pyplot as plt
13 from qat.lang.AQASM import *
14 from qat.qpus import PyLinalg
15 from qat.lang.AQASM import AbstractGate
16
17
18 def Phase_generator(theta):
19     return np.array([[np.exp(1j * theta), 0],
20                     [0, np.exp(1j * theta)]])
21
22
23 ## This function will build the circuit that applies the ABC decomposition
24 # with the values we chose. It will normally output a controlled X gate.
25 # Reminder : a controlled gate works like a if then else statement.
26 def X_controlled_gate(alpha, theta2, theta1, theta0, qbit_to_1=False):

```



```

27
28 GlobalPhase = AbstractGate("Phase", [float], arity=1,
29                             matrix_generator=Phase_generator)
30
31 prog = Program()
32 qbits = prog.qalloc(2)
33
34 # If we want to test if the control works,
35 # we can change the value of the first qbit before the circuit.
36 if qbit_to_1:
37     prog.apply(X, qbits[0])
38
39 # Building the circuit
40
41 # Then the C gate
42 prog.apply(RZ((theta0-theta2)/2), qbits[1])
43
44 # Then the CNOT
45 prog.apply(CNOT,qbits[0], qbits[1])
46
47 #We put B on the first qbit
48 prog.apply(RZ((-theta0-theta2)/2), qbits[1])
49 prog.apply(RY(-theta1/2), qbits[1])
50
51 # Then the CNOT
52 prog.apply(CNOT,qbits[0], qbits[1])
53
54 # We put the Rz rotation on the second qbit
55 prog.apply(GlobalPhase(alpha), qbits[0])
56
57 # We put A on the first qbit
58 prog.apply(RY(theta1/2), qbits[1])
59 prog.apply(RZ(theta2), qbits[1])
60
61 # Running the circuit
62 circuit = prog.to_circ()
63 circuit.display()
64
65 job = circuit.to_job()
66
67 linalgqpu = PyLinalg()
68
69 # printing the result and showing the probabilities
70 result = linalgqpu.submit(job)
71 l = len(result)
72 states = ['']*l
73 probabilities= [0]*l
74
75 i=0
76 for sample in result:
77     print("State",sample.state,"with amplitude",
78           sample.amplitude,"and probability",
79           round(sample.probability*100,2),"%")
80     states[i] = str(sample.state)
81     probabilities[i] = round(sample.probability*100,2)
82     i = i+1
83 plt.bar(states, probabilities, color='skyblue')
84 plt.xlabel('States')
85 plt.ylabel('Probabilities')
86 plt.show()
87

```

```

88 # We said previously that :
89 # alpha = pi/2
90 # theta2 = 0
91 # theta1 = pi
92 # theta0 = pi
93
94 X_controlled_gate(np.pi/2,0,np.pi,np.pi)
95
96 # A controlled gate works like a if then else statement.
97 # We put |00> in input, if the first qbit is 1,
98 # we apply the X gate on the second qbit.
99 # By default, both qbits are set to 0, so we should have |00> in output.
100
101 # The output is indeed |00> with 100% probability.
102
103 # If we change the value of the first qbit to 1, we should have |11> in output.
104
105 X_controlled_gate(np.pi/2,0,np.pi,np.pi,True)
106
107 # 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 :

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

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 paramétrée R_x . En effet, rappelons que la porte X réalise une rotation d'un angle π autour de l'axe X, il 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 :

$$R_x\left(\frac{\pi}{2}\right) = \sqrt{X} \quad (10)$$

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

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

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

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

En factorisant , nous avons bien :

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

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 :

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

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

```

1 #####
2
3 # Lapu Matthias
4
5 # The goal is to create a circuit that will apply the X gate
6 # with a squared gate using the Sleathor Weinfurter reduction
7
8 #####
9
10 import cmath
11 from math import *
12 import numpy as np
13 import matplotlib.pyplot as plt
14 from qat.lang.AQASM import *
15 from qat.qpus import get_default_qpu
16 from itertools import product
17 from scipy.linalg import sqrtm
18
19 # Must create all the possible states for 2 bits
20 # like creating_2n_states(2) will return [[0,0],[0,1],[1,0],[1,1]]
21 def creating_2n_states(n):

```

```

22     return list(product([0, 1], repeat=n))
23
24
25 # We need to create the X squared gate, because we apply it to 2 qubits
26 # we need it to be a 4x4 matrix
27 def X_squared_gate():
28     return np.array([[1, 0, 0, 0],
29                     [0, 1, 0, 0],
30                     [0, 0, np.sqrt(2)/2, -1j*np.sqrt(2)/2],
31                     [0, 0, -1j*np.sqrt(2)/2, np.sqrt(2)/2]])
32
33 # We will also need the dagger of the X squared gate
34 # it's the adjoint of the matrix
35 def X_squared_gate_dagger():
36     return X_squared_gate().conj().T
37
38 # To build the controlled gate using 3qubits, we need the fourth root
39 # of the X gate
40 def X_fourth_gate():
41     return sqrtm(X_squared_gate())
42
43 def X_fourth_gate_dagger():
44     return X_fourth_gate().conj().T
45
46 # We will create a generic function that will apply the
47 # Sleathor Weinfurter reduction, thus we need V and V dagger gates
48 def Sleathor_Weinfurter_2qubits_X(mat, mat_dagger, state):
49
50     # Starting the program and allocating 3 qubits
51     prog = Program()
52     qbits = prog.qalloc(3)
53
54     # Creating the initial state
55     for i in range(len(state)):
56         if state[i] == 1:
57             prog.apply(X, qbits[i])
58
59     # Then, we create the V and V dagger gates using the function
60     # passed as an argument
61     V = AbstractGate("V", [], arity=2, matrix_generator=mat)
62     VD = AbstractGate("VD", [], arity=2, matrix_generator=mat_dagger)
63
64
65     # The Sleathor Weinfurter reduction with 2 qubits is the following :
66     # - apply the V gate on the 2nd and 3rd qubits
67     # - apply a CNOT gate. The X is on the 2nd qbit, the target is the 1st
68     # - apply the V dagger gate on the 2nd and 3rd qubits
69     # - apply a CNOT gate. The X is on the 2nd qbit, the target is the 1st
70     # - apply the V gate on the 1st and 3rd qubits
71
72     # Building the circuit
73     prog.apply(V(), qbits[1], qbits[2])
74     prog.apply(CNOT, qbits[0], qbits[1])
75     prog.apply(VD(), qbits[1], qbits[2])
76     prog.apply(CNOT, qbits[0], qbits[1])
77     prog.apply(V(), qbits[0], qbits[2])
78
79     # Running the circuit
80     circuit = prog.to_circ()
81     # circuit.display()
82

```

```

83 mypylinalgqpu = get_default_qpu()
84
85 job = circuit.to_job()
86 result = mypylinalgqpu.submit(job)
87
88 # Plotting the results and the percentage of each state :w!
89 l = len(result)
90 states = ['']*l
91 probabilities= [0]*l
92
93 i=0
94 for sample in result:
95     print("State",sample.state,"with amplitude",
96           sample.amplitude,"and probability",
97           round(sample.probability*100,2),"%")
98     states[i] = str(sample.state)
99     probabilities[i] = round(sample.probability*100,2)
100    i = i+1
101 plt.bar(states, probabilities, color='skyblue')
102 plt.xlabel('States')
103 plt.ylabel('Probabilities')
104 plt.show()
105
106 ## Same goal, function kind of similar, but with 3 qubits
107 def Sleathor_Weinfurter_3qubits_X(mat,mat_dagger,state):
108
109     # Starting the program and allocating 3 qubits
110     prog = Program()
111     qbits = prog.qalloc(4)
112
113     # Creating the initial state
114     for i in range(len(state)):
115         if state[i] == 1:
116             prog.apply(X, qbits[i])
117
118     # Then, we create the V and V dagger gates using the function
119     # passed as an argument
120     V = AbstractGate("V", [], arity=2, matrix_generator=mat)
121     VD = AbstractGate("VD", [], arity=2, matrix_generator=mat_dagger)
122
123
124     # The Sleathor Weinfurter reduction with 3 qubits is the following :
125     # - apply the V gate on the 1st qbit
126     # - apply a CNOT gate. The X is on the 2nd qbit, the target is the 1st
127     # - apply a VD gate on the 2nd qbit
128     # - apply a CNOT gate. The X is on the 2nd qbit, the target is the 1st
129     # - apply the V gate on the 2nd qbit
130     # - apply a CNOT gate. The X is on the 3rd qbit, the target is the 2nd
131     # - apply a VD gate on the 3rd qbit
132     # - apply a CNOT gate. The X is on the 3rd qbit, the target is the 1st
133     # - apply the V gate on the 3rd qbit
134     # - apply a CNOT gate. The X is on the 3rd qbit, the target is the 2nd
135     # - apply a VD gate on the 3rd qbit
136     # - apply a CNOT gate. The X is on the 3rd qbit, the target is the 1st
137     # - apply the V gate on the 3rd qbit
138
139     # Building the circuit
140     prog.apply(V(),qbits[0],qbits[3])
141     prog.apply(CNOT, qbits[0], qbits[1])
142     prog.apply(VD(), qbits[1], qbits[3])
143     prog.apply(CNOT, qbits[0], qbits[1])

```

```

144 prog.apply(V(), qbits[1], qbits[3])
145 prog.apply(CNOT, qbits[1], qbits[2])
146 prog.apply(VD(), qbits[2], qbits[3])
147 prog.apply(CNOT, qbits[0], qbits[2])
148 prog.apply(V(), qbits[2], qbits[3])
149 prog.apply(CNOT, qbits[1], qbits[2])
150 prog.apply(VD(), qbits[2], qbits[3])
151 prog.apply(CNOT, qbits[0], qbits[2])
152 prog.apply(V(), qbits[2], qbits[3])
153
154
155 # Running the circuit
156 circuit = prog.to_circ()
157 # circuit.display()
158
159 mypylinalgqpu = get_default_qpu()
160
161 job = circuit.to_job()
162 result = mypylinalgqpu.submit(job)
163
164 # Plotting the results and the percentage of each state :w!
165 l = len(result)
166 states = ['']*l
167 probabilities= [0]*l
168
169 i=0
170 for sample in result:
171     print("State", sample.state, "with amplitude",
172           sample.amplitude, "and probability",
173           round(sample.probability*100,2), "%")
174     states[i] = str(sample.state)
175     probabilities[i] = round(sample.probability*100,2)
176     i = i+1
177 plt.bar(states, probabilities, color='skyblue')
178 plt.xlabel('States')
179 plt.ylabel('Probabilities')
180 plt.show()
181
182
183 ## To test if the X squared gate works,
184 # if the first qbit is one, the second qbit should switch
185 # otherwise, the second qbit should stay the same
186 def testing_X_sq2_sq4(state, th_gate):
187
188     # The test is only for the squared gate and the fourth gate
189     if th_gate != 2 and th_gate != 4:
190         print("The th_gate must be 2 or 4")
191         return
192
193     prog = Program()
194     qbits = prog.qalloc(2)
195
196     # Creating the gate depending on the th_gate
197     if th_gate == 2:
198         V = AbstractGate("V", [], arity=2, matrix_generator=X_squared_gate)
199     else:
200         V = AbstractGate("V", [], arity=2, matrix_generator=X_fourth_gate)
201
202
203     # Creating the initial state
204     for i in range(len(state)):

```

```

205     if state[i] == 1:
206         prog.apply(X, qbits[i])
207
208     # Applying the gate th_gate times should always return the X gate
209     for i in range(th_gate):
210         prog.apply(V(), qbits[0], qbits[1])
211
212     # Running the circuit
213     circuit = prog.to_circ()
214     # circuit.display()
215
216     mypylinalgqpu = get_default_qpu()
217
218     job = circuit.to_job()
219     result = mypylinalgqpu.submit(job)
220
221     # printing the percentage of each state
222     for sample in result:
223         print("State", sample.state, ", probability",
224               round(sample.probability*100,2), "%")
225
226
227 def main():
228
229     # Testing the X squared gate
230     for state in creating_2n_states(2):
231         print("Testing the X squared gate with the state : ", state)
232         testing_X_sq_sq4(state, 2)
233         # looks ok
234
235     print("-----")
236
237     # Testing the X fourth gate
238     for state in creating_2n_states(2):
239         print("Testing the X fourth gate with the state : ", state)
240         testing_X_sq_sq4(state, 4)
241         # looks ok
242     print("-----")
243
244     # Testing the Sleathor Weinfurter reduction with 2 qubits
245     for state in creating_2n_states(2):
246         print("Testing the Sleathor Weinfurter reduction with the state : ", state)
247         Sleathor_Weinfurter_2qubits_X(X_squared_gate, X_squared_gate_dagger, state)
248
249     print("-----")
250
251     # Testing the Sleathor Weinfurter reduction with 3 qubits
252     for state in creating_2n_states(3):
253         print("Testing the Sleathor Weinfurter reduction with the state : ", state)
254         Sleathor_Weinfurter_3qubits_X(X_fourth_gate, X_fourth_gate_dagger, state)
255
256     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) contrôlé : 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) contrôlé : 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 contrôlée, nous utilisons la propriété par bloc de la matrice afin de la créer récursivement, en effet, nous avons que

$$C_x NOT = \begin{pmatrix} I & 0 \\ 0 & CX \end{pmatrix} \quad (15)$$

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 qubits et 3 qubits 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 qubits. Je n'ai donc pas pu continuer mon implémentation.


```

1 #####
2
3 # Lapu Matthias
4
5 # The goal is to create a controlled gate on n qbits, using NOT that
6 # controls multiple qbits.
7 #####
8
9 from math import *
10 import numpy as np
11 from qat.qpus import PyLinalg
12 from qat.lang.AQASM import AbstractGate
13 from qat.lang.AQASM import *
14 import matplotlib.pyplot as plt
15 from scipy.linalg import sqrtm
16
17
18 # We need to create the X squared gate, because we apply it to 2 qubits
19 # we need it to be a 4x4 matrix
20 def X_squared_gate():
21     return np.array([[1, 0, 0, 0],
22                     [0, 1, 0, 0],
23                     [0, 0, np.sqrt(2)/2, -1j*np.sqrt(2)/2],
24                     [0, 0, -1j*np.sqrt(2)/2, np.sqrt(2)/2]])
25
26 # We will also need the dagger of the X squared gate
27 # it's the adjoint of the matrix
28 def X_squared_gate_dagger():
29     return X_squared_gate().conj().T
30
31 # To build the controlled gate using 3qbits, we need the fourth root
32 # of the X gate
33 def X_fourth_gate():
34     return sqrtm(X_squared_gate())
35
36 def X_fourth_gate_dagger():
37     return X_fourth_gate().conj().T
38
39 ## This function will create the CNOT matrix
40 # Reminder : the X gate matrix is : [[0,1],[1,0]]
41 # The identity matrix is : [[1,0],[0,1]]
42 # The CX matrix can be written as a block matrix like this :
43 # [[I, 0], [0, X]]
44 def CNOT_matrix():
45     # we need to create 2 matrix of size (n*n), the identity matrix
46     # and the X gate matrix
47     # Identity matrix
48     I = np.identity(2)
49     Zero = np.zeros((2,2))
50     X = np.flip(np.identity(2),0)
51
52     # Creating the CNOT matrix
53     CNOT = np.block([[I, Zero],
54                     [Zero, X]])
55     return CNOT
56
57 ## The toffoli gate, which is a controlled not gate on 3 qbits can be
58 # written as a block matrix like this :
59 # [[I, 0], [0, CX]] with CX the CNOT matrix written above
60 # Thus, we can write a recursive function that will create a CNOT gate

```

```

61 # on n qbits
62 # n=3 should return the Toffoli gate
63 def recursive_CNOT(n):
64     if n == 2:
65         return CNOT_matrix()
66     else:
67         return np.block([[np.identity(2**(n-1)), np.zeros((2**(n-1),2**(n-1)))],
68                          [np.zeros((2**(n-1),2**(n-1))), recursive_CNOT(n-1)]])
69
70
71 ## Build the Sleathor Weinfurter reduction with 2 qbits :w!
72 def Sleathor_Weinfurter_2qubits_X(prog,mat,mat_dagger,starting_qbit:int):
73
74     V = AbstractGate("V", [], arity=2, matrix_generator=mat)
75     VD = AbstractGate("VD", [], arity=2, matrix_generator=mat_dagger)
76
77     final_qbit = starting_qbit+3 # we must hop one of the qbit
78     # Building the circuit
79     prog.apply(V(), qbits[starting_qbit+1], qbits[final_qbit])
80     prog.apply(CNOT, qbits[starting_qbit], qbits[starting_qbit+1])
81     prog.apply(VD(), qbits[starting_qbit+1], qbits[final_qbit])
82     prog.apply(CNOT, qbits[starting_qbit], qbits[starting_qbit+1])
83     prog.apply(V(), qbits[starting_qbit], qbits[final_qbit])
84
85
86 ## Build the Sleathor Weinfurter reduction with 3 qbits
87 # explained in Sleathor_Weinfurter_reduction.py
88 def Sleathor_Weinfurter_3qubits_X(prog,mat,mat_dagger,
89                                   starting_qbit:int,):
90
91     # the starting qbit is (from the top) where the firts qbit available
92     # for the scope of the reduction is located
93
94     # Then, we create the V and V dagger gates using the function
95     # passed as an argument
96     V = AbstractGate("V", [], arity=2, matrix_generator=mat)
97     VD = AbstractGate("VD", [], arity=2, matrix_generator=mat_dagger)
98
99     final_qbit = starting_qbit+4 # Building the circuit
100
101     # Building the circuit
102     prog.apply(V(),qbits[starting_qbit],qbits[final_qbit])
103     prog.apply(CNOT, qbits[starting_qbit], qbits[starting_qbit+1])
104     prog.apply(VD(), qbits[starting_qbit+1], qbits[final_qbit])
105     prog.apply(CNOT, qbits[starting_qbit], qbits[starting_qbit+1])
106     prog.apply(V(), qbits[starting_qbit+1], qbits[final_qbit])
107     prog.apply(CNOT, qbits[starting_qbit+1], qbits[starting_qbit+2])
108     prog.apply(VD(), qbits[starting_qbit+2], qbits[final_qbit])
109     prog.apply(CNOT, qbits[starting_qbit], qbits[starting_qbit+2])
110     prog.apply(V(), qbits[starting_qbit+2], qbits[final_qbit])
111     prog.apply(CNOT, qbits[starting_qbit+1], qbits[starting_qbit+2])
112     prog.apply(VD(), qbits[starting_qbit+2], qbits[final_qbit])
113     prog.apply(CNOT, qbits[starting_qbit], qbits[starting_qbit+2])
114     prog.apply(V(), qbits[starting_qbit+2], qbits[final_qbit])
115
116
117
118
119 # To create a controlled gate on m qbits,
120 # we need a recursive CNOT to apply to m-2 qbits located at the m-1 qbit
121 # (so just 1 before the last qbit)

```

```

122 # We also need the V controlled gate on m-2 qbits located at the m qbit
123
124 def controlled_gate_n_qbits_X(m,mat,squared_mat,mat_dagger,squared_mat_dagger):
125     prog = Program()
126     qbits = prog.qalloc(m)
127
128     # We need to create the controlled CNOT gate on m-2 qbits
129
130     CCNOT = AbstractGate("CCNOT", [], arity=(m-1),
131                          matrix_generator=recursive_CNOT(m-2))
132
133     # We also need to create the V gate and the V dagger gate
134     V = AbstractGate("V", [], arity=2, matrix_generator=mat)
135     VD = AbstractGate("VD", [], arity=2, matrix_generator=mat_dagger)
136
137     # Then we apply the V gate on the m-2 and m-1 qbits
138
139     prog.apply(V(), qbits[m-2], qbits[m-1])
140
141     # the list of qbits that the cnot gate will control
142     qbits_to_control = [qbits[i] for i in range(m-2)]
143
144     # Then we apply the CNOT gate on the m-2 qbits to control the m-3 qbit
145     # The syntax error comes from here, I did not find a way to fix it
146     # The documentation only talks about gate with a clear number of qbits
147     # I did not find any example of a gate that controls variable qbits
148     prog.apply(CCNOT,qbits_to_control, qbits[m-1])
149     prog.apply(VD(), qbits[m-2], qbits[m-1])
150     prog.apply(CCNOT,qbits_to_control, qbits[m-1])
151
152     # The only thing we need is to create the V controlled gate on m-2 qbits
153     # It can be seen as a U gate, in this case the V and VD gate of this one
154     # would be the fourth root of the X gate
155
156     if m == 4:
157         Sleathor_Weinfurter_2qubits_X(prog,squared_mat,squared_mat_dagger,m-1)
158     elif m == 5:
159         Sleathor_Weinfurter_3qubits_X(prog,squared_mat,squared_mat_dagger,m-2)
160     else:
161         controlled_gate_n_qbits_X(m-1,mat,squared_mat,
162                                   mat_dagger,squared_mat_dagger)
163
164     # Running the circuit
165     circuit = prog.to_circ()
166     circuit.display()
167
168     mypylinalgqpu = get_default_qpu()
169
170     job = circuit.to_job()
171     result = mypylinalgqpu.submit(job)
172
173     # Plotting the results and the percentage of each state :w!
174     l = len(result)
175     states = ['']*l
176     probabilities= [0]*l
177
178     i=0
179     for sample in result:
180         print("State",sample.state,"with amplitude",
181               sample.amplitude,"and probability",
182               round(sample.probability*100,2),"%")

```

```

183     states[i] = str(sample.state)
184     probabilities[i] = round(sample.probability*100,2)
185     i = i+1
186     plt.bar(states, probabilities, color='skyblue')
187     plt.xlabel('States')
188     plt.ylabel('Probabilities')
189     plt.show()
190
191     controlled_gate_n_qbits_X(5,X_squared_gate,X_fourth_gate,
192                               X_squared_gate_dagger,X_fourth_gate_dagger)

```