Screening Task 1

Alvin Moon

1 Introduction

This is the summary write-up for my response to the first screening task. The code for this task is written in Python using the QuTIP, scipy and numpy modules. The website for QuTIP is `http://qutip.org/`.

2 Plot

I'll illustrate the output of the code here. The state is the normalized sum of two random elementary tensors.

```
>>> psi1 = tensor(rand_ket(2), rand_ket(2),rand_ket(2),rand_ket(2))
>>> psi2 = tensor(rand_ket(2), rand_ket(2),rand_ket(2),rand_ket(2))
>>> psi = (psi1 + psi2)
>>> psi = psi / (psi.norm())
>>> psi.norm()
1.0
>>> psi
Quantum object: dims = [[2, 2, 2, 2], [1, 1, 1, 1]], shape = (16, 1), type = ket
Qobj data =
[[ 0.04921436+0.15037292j]
 [-0.04917617-0.02627526j]
 [-0.26653504+0.26833976j]
 [ 0.10321357-0.12096351j]
 [ 0.12194049-0.02494125j]
 [-0.01277009+0.00999079j]
 [ 0.16343802+0.32213059j]
 [-0.10278496-0.12737783j]
 [-0.03812158+0.31572518j]
 [-0.33347538-0.14086358j]
 [ 0.15715032+0.25940415j]
 [ 0.00384048+0.1100241j ]
 [ 0.31916975-0.17671234j]
 [ 0.12804888+0.31561003j]
 [ 0.08048657-0.01804298j]
 [ 0.03398412-0.19273902j]]
```

As an example of the output of this code with this choice of ψ, here is a graph where the x-axis is the number L of layers and the y-axis is the minimum distance ϵ from the random vector ψ. The optimization was done using numpy's minimizer, `numpy.optimize.minimize`. The graph was produced using `matplotlib`.

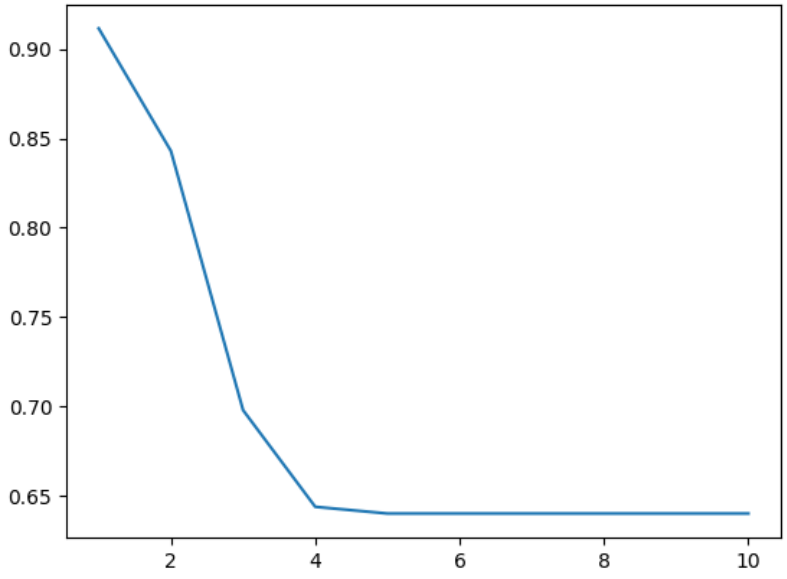

Figure 1: The x-axis is the number L of layers and the y-axis is the minimum distance ϵ from the random vector ψ to the set of $\psi(\theta)$.

3 Code descriptions

3.1 Gates

I defined three gates: C_z, R_x, R_z. R_x and R_z are rotation gates acting on a single cubit:

```
def Rx(theta):
    x = theta/2
    Rx = Qobj([ [math.cos(x), -complex(0,math.sin(x))],
    [-complex(0,math.sin(x)), math.cos(x)]] )

    return Rx

def Rz(theta):
    x = complex(0,theta/2)
    Rz = Qobj([ [cmath.exp(x), 0], [0, cmath.exp(x)]])

    return Rz
```

C_z is a controlled gate which applies σ^z to a certain subspace of $\mathbb{C}^2 \otimes \mathbb{C}^2$.

```
def Cz(x,y):
    P1 = Qobj([ [0,0], [0,1]])
    P0 = Qobj([ [1,0], [0,0]])
#Pn is the projection onto the nth basis vector, n=0,1
```

```
    if x==y:
        print('Sites should differ')
    elif min(x,y)<0:
        print('Sites must be in 0,1,2,3')
    elif max(x,y)>3:
        print('Sites must be in 0,1,2,3')
    else:
        T = Qobj([ [0,0], [0,0]])
        S = Qobj([ [0,0], [0,0]])
# T is, up to non-identity tensor factors, P1 \otimes sigma z
# S is, up to non-identity tensor factors, P0 \otimes identity
# Then CZ = T + S.
        for a in [0,1,2,3]:
            if a == 0:
                if a == x:
                    T = P1
                    S = P0
                elif a == y:
                    T = sigmaz()
                    S = identity(2)
                else:
                    T = identity(2)
                    S = identity(2)
            else:
                if a == x:
                    T = tensor(T, P1)
                    S = tensor(S, P0)
                elif a == y:
                    T = tensor(T, sigmaz())
                    S = tensor(S, identity(2))
                else:
                    T = tensor(T, identity(2))
                    S = tensor(S, identity(2))
    return T + S
```

3.2 Optimization code

This section contains the code used to optimize

$$\epsilon(\theta) = \|\psi - \psi(\theta)\|.$$

It is contained in the file `bulk.py`.

`evenBlock` and `oddBlock` are functions which implement the even and odd-labeled unitaries in the circuit diagram.

```
def evenBlock(theta):
    U = tensor(Rz(theta[0]), Rz(theta[1]), Rz(theta[2]), Rz(theta[3]))
    C1 = Cz(0,1)
    C2 = Cz(0,2)
    C3 = Cz(0,3)
```

```
    C4 = Cz(1,2)
    C5 = Cz(1,3)
    C6 = Cz(2,3)

    Even = C6 * C5 * C4 * C3 * C2 * C1 * U

    return Even


def oddBlock(theta):
    Odd = tensor(Rx(theta[0]), Rx(theta[1]), Rx(theta[2]), Rx(theta[3]))

    return Odd
```

`layer` is the product of an even and odd-labeled unitary. The inputs theta and phi are length 4 lists, and the first angle is the argument for oddBlock.

`preparedVector` prepares the vector $\psi(\theta)$ from $|-\rangle^{\otimes 4}$. Theta is length 8L. First 4L entries of Theta are even angles, last 4L are odd angles, and L is number of layers.

```
def layer(theta, phi):
    T = evenBlock(phi) * oddBlock(theta)

    return T


def preparedVector(L,Theta):
    xi = tensor(basis(2,0), basis(2,0), basis(2,0), basis(2,0))
    theta = Theta[0:4*L]
    phi = Theta[4*L:8*L]

    for j in range(0,4*L,4):
        xi = oddBlock([phi[j], phi[j+1], phi[j+2], phi[j+3]])*xi
        xi = evenBlock([theta[j],theta[j+1],theta[j+2],theta[j+3]])*xi

    return xi
```

`objFunction` is the objective function for the minimizer. `distanceMinimizer` minimizes $\epsilon(\theta) = \|\psi - \psi(\theta)\|$ with inputs ψ and L. The minimization is done using the built in optimization tool in `numpy`.

```
def objFunction(Theta,L,psi):
     x = preparedVector(L,Theta) - psi
     ep = x.norm()

     return ep
```

```
def distanceMinimizer(psi, L):
    d = []
    for j in range(1,L+1):
        y = scipy.optimize.minimize(objFunction, numpy.zeros(8*j),
        args=(j,psi),method = None, jac=None, hess=None, hessp=None, bounds = None, constraints=(),
        tol=None, callback=None, options=None).fun

        d.append(y)

    return d
```