# Task4

September 27, 2020

[2]:
```
%matplotlib inline
# Importing standard Qiskit libraries and configuring account
from qiskit import QuantumCircuit, execute, Aer, IBMQ
from qiskit.compiler import transpile, assemble
from qiskit.tools.jupyter import *
from qiskit.visualization import *
# Loading your IBM Q account(s)
provider = IBMQ.load_account()
```

```
/opt/conda/lib/python3.7/site-packages/qiskit/providers/ibmq/ibmqfactory.py:192:
UserWarning: Timestamps in IBMQ backend properties, jobs, and job results are
all now in local time instead of UTC.
  warnings.warn('Timestamps in IBMQ backend properties, jobs, and job results '
```

[3]:
```
import numpy as np
from qiskit import BasicAer
from qiskit.aqua.components.optimizers import COBYLA
```

[4]:
```
# given hamiltonian
H = np.array([
    [1, 0, 0, 0],
    [0, 0,-1, 0],
    [0,-1, 0, 0],
    [0, 0, 0, 1]
], dtype=np.complex128)
# ground truths
eigvals,eigvects = np.linalg.eig(H)
target_eigval = min(eigvals).real
target_eigvect = eigvects[:,eigvals == min(eigvals)]
print(f'all eigenvalues: {eigvals}')
print(f'actual min eigenvalue: {target_eigval}')
print('actual min eigenvector: ')
print(eigvects[:,eigvals == min(eigvals)])
```

```
all eigenvalues: [ 1.+0.j -1.+0.j  1.+0.j  1.+0.j]
actual min eigenvalue: -0.9999999999999999
actual min eigenvector:
[[0.        +0.j]
```

```
[0.70710678+0.j]
[0.70710678+0.j]
[0.        +0.j]]
```

[5]:
```python
# pauli decomposition for 2x2 matrix
def pauli_decomp(H):
    pauli = { 'I': [[1,0],[0,1]],
              'X': [[0,1],[1,0]],
              'Y': [[0,-1j],[1j,0]],
              'Z': [[1,0],[0,-1]],}
    # coeff = 1/4 * tr{ (P1   P2) H }
    if H.shape[0] == 4:
        all_coeffs = { P1+P2: np.trace( np.kron(pauli[P1],pauli[P2]) @ H ).real / 4
                                for P1 in pauli for P2 in pauli }
    # filter only non-zeros
    return { PP: coeff for PP,coeff in all_coeffs.items() if abs(coeff) > 0 }
pauli_decomp(H)
```

[5]:
```
{'II': 0.5, 'XX': -0.5, 'YY': -0.5, 'ZZ': 0.5}
```

[6]:
```python
# create ansatz circuit
def ansatz(theta):
    qc = QuantumCircuit(2)
    # H   I
    qc.h(0)
    # CX
    qc.cx(0,1)
    # RX   I
    qc.rx(theta,0)
    return qc

# returns a vqe circuit depending on the measurement basis
def vqe_circuit(basis, params):
    qc = ansatz(*params)
    # transform to comupatational (Z) basis
    for i,b in enumerate(basis):
        if b == 'X':
            qc.ry(-np.pi/2, i)
        elif b == 'Y':
            qc.rx(np.pi/2, i)
    # measurement at the end
    qc.measure_all()
    return qc
```

[7]:
```python
# calculate expectation of a pauli matrix
def pauli_expect(counts, shots):
```

```python
    # if basis has even parity (e.g. |11>,|00>), sign = +1, else -1
    sign = lambda basis: 1 if basis.count('1') % 2 == 0 else -1
    # function to calculate actual expectations of the pauli matrix
    return sum( sign(basis) * counts.get(basis,0) for basis in counts ) / shots

# calculate expectation of a hamiltonian using VQE
def expectation( params,
                        backend = BasicAer.get_backend('qasm_simulator'),
                        shots = 2**13):
    # decompose hamiltonian into pauli matrices
    decomposed = pauli_decomp(H)
    # no vqe circuit needed for identity gate
    coeff_id = decomposed.pop('I' * int(np.log2(H.shape[0])), 0)

    # execute all the circuits to get counts
    all_qcs = [vqe_circuit(pauli_term, params) for pauli_term in decomposed]
    job = execute(all_qcs, backend, shots = shots)
    all_counts = job.result().get_counts()
    # multiply by corresponding co-efficients to get module expectations
    module_expects = [ coeff * pauli_expect(counts, shots) for counts, coeff
                         in zip(all_counts, decomposed.values()) ]
    return coeff_id + sum(module_expects)
```

[8]:
```python
optimizer = COBYLA(maxiter=500, tol=1e-4)
params = np.random.rand(1)
ret = optimizer.optimize(len(params), expectation, initial_point=params)
```

[9]:
```python
pred_eigval = ret[1]
pred_params = [ret[0].tolist()] if ret[0].shape == () else ret[0].tolist()
ansatz_job = execute(ansatz(*pred_params),
                BasicAer.get_backend('statevector_simulator'))
pred_eigvect = np.array([ansatz_job.result().get_statevector()]).T

np.set_printoptions(suppress=True, precision=3)
print(f'actual min eigenvalue: {target_eigval:.4f}')
print(f'predicted min eigenvalue: {pred_eigval:.4f}')
print(f'error: {abs(1 - pred_eigval/target_eigval) * 100 : .2f}%\n')
print('actual min eigenvector: ')
print(target_eigvect,'\n')
print('predicted min eigenvector: ')
print(pred_eigvect/np.vdot(target_eigvect,pred_eigvect), '\n')
print(f'absolute inner product: {abs(np.vdot(target_eigvect,pred_eigvect))}')
```

```
actual min eigenvalue: -1.0000
predicted min eigenvalue: -1.0000
error:  0.00%
```

```
actual min eigenvector:
[[0.   +0.j]
 [0.707+0.j]
 [0.707+0.j]
 [0.   +0.j]]

predicted min eigenvector:
[[0.   +0.003j]
 [0.707+0.j   ]
 [0.707-0.j   ]
 [0.   +0.003j]]

absolute inner product: 0.9999925758938519
```