Electronic Excited States from a Variance-Based Contracted Quantum Eigensolver

August 22, 2023

This is the demonstration jupyter notebook of "Electronic Excited States from a Variance-Based Contracted Quantum Eigensolver" by Yuchen Wang and David A. Mazziotti.

[1]:
```
#import qiskit, scipy, numpy
from qiskit import QuantumCircuit,QuantumRegister,ClassicalRegister
from qiskit import Aer,execute
import numpy as np
import itertools
from scipy.linalg import expm
```

[2]:
```
#Pauli matrices
sigma_x = np.array([[0, 1],  [1, 0]], dtype=complex)
sigma_y = np.array([[0, -1j],[1j, 0]], dtype=complex)
sigma_z = np.array([[1, 0],  [0, -1]], dtype=complex)
iden = np.array([[1, 0],  [0, 1]], dtype=complex)
#initialize quantum circuit
def init_qc(N):
    q = QuantumRegister(N)
    c = ClassicalRegister(N)
    qc = QuantumCircuit(q,c)
    return q,c,qc
#apply tensor product of a list of matrices from right to left
def tensors(list_of_mats):
    num_mats = len(list_of_mats)
    tensor_product = list_of_mats[0]
    i=0
    while i+1 < num_mats:
        tensor_product = np.kron(tensor_product,list_of_mats[i+1])
        i=i+1
    return tensor_product
#define the tensor product from Pauli string
def str2mat(pstr):
    matrix_list = []
    for char in pstr:
        if char == 'I':
            matrix_list.append(iden)
```

```
        elif char == 'X':
            matrix_list.append(sigma_x)
        elif char == 'Y':
            matrix_list.append(sigma_y)
        elif char == 'Z':
            matrix_list.append(sigma_z)
    return tensors(matrix_list)
#this returns a dictionary of Pauli String from any unitary operator
def HS(m, n):
    return (np.dot(m.conjugate().transpose(), n)).trace()
def decomposeNqubit(H, N=4):
    pmat = [iden, sigma_x, sigma_y, sigma_z]
    char = ['I', 'X', 'Y', 'Z']
    char_list = [''.join(i) for i in itertools.product(char, repeat = N)]
    pmat_list = [i for i in itertools.product(pmat, repeat = N)]
    decomposed_dict = {}
    for i in range(len(char_list)):
        a_ij = 1/(2**N) * HS(tensors(pmat_list[i]), H)
        if np.abs(a_ij) > 0.00000001:
            decomposed_dict[char_list[i]] = a_ij
    return decomposed_dict
```

[4]:
```
#binary dictionary of Pauli string
def z_to_binary(z_string, N=4):
    labels = ['0', '1']
    b_string = [''.join(i) for i in itertools.product(labels, repeat = N)]
    tensor_list = []
    b_dict = {}
    for i in z_string:
        if i == 'Z':
            tensor_list.append(np.real(sigma_z))
        elif i == 'I':
            tensor_list.append(np.real(iden))
    matrix = tensors(tensor_list)
    for i in range(len(b_string)):
        b_dict[b_string[i]] = matrix[i, i]
    return b_dict
#get z-basis string from statevector/qasm simulator
def measure_zbasis(qc, N=4):
    result = execute(qc, Aer.get_backend('statevector_simulator')).result()
    statevector = result.get_statevector()
    #this sums up all binary string
    b_coeff = np.zeros(2**N)
    b_label = ['0', '1']
    b_string = [''.join(i) for i in itertools.product(b_label, repeat = N)]
    for i in range(len(np.asarray(statevector))):
        b_coeff[i] = np.abs(statevector[i])**2
```

```
    #this sums up all z basis string
    z_coeff = np.zeros([2**N]) #z_coeff is what we want to calculate
    z_label = ['Z', 'I']
    z_string = [''.join(i) for i in itertools.product(z_label, repeat = N)]
    for k in (range(len(z_string))):
        dict_binary = z_to_binary(z_string[k])
        for j in range(len(b_string)):
            if b_string[j] in dict_binary:
                z_coeff[k] += b_coeff[j] * dict_binary[b_string[j]]
    return z_coeff, z_string
#change the basis to get X,Y value
def measure_abasis(qc, string, N=4):
    for pos,i in enumerate(string):
        if i == 'X' :
            qc.h(pos) #convert qubit to X
        elif i == 'Y' :
            qc.z(pos) #convert qubit to Y
            qc.s(pos)
            qc.h(pos)
    a_coeff, a_string = measure_zbasis(qc)
    for pos,i in enumerate(string):
        if i == 'X' or i == 'Y':
            for j in range(len(a_string)):
                if a_string[j][pos] == 'Z':
                    a_string[j] = a_string[j][:pos]+i+a_string[j][(pos+1):]
    return a_coeff, a_string
#measure the value w.r.t given dict
def measure_from_dict(qc, m_dict):
    e = 0
    for item in m_dict:
        qc_copy = qc.copy()
        a_coeff, a_string = measure_abasis(qc_copy, item)
        for i in range(len(a_string)):
            if item == a_string[i]:
                e += m_dict[item] * a_coeff[i]
    return e
#this applys pauli string to circuits
def apply_pauli_string(qc, pauli):
    for q, i in enumerate(pauli):
        if i == "X":
            qc.x(q)
        elif i == "Y":
            qc.rz(-np.pi / 2, q)
            qc.x(q)
            qc.rz(np.pi / 2, q)
        elif i == "Z":
            qc.rz(np.pi, q)
```

```
def pauli_string(qc, loc, sigma="x", inv=False):
    if sigma in ["Z", "z"]:
        pass
    elif sigma in ["X", "x"]:
        qc.h(q[loc])
    elif sigma in ["I", "i"]:
        pass
    elif sigma in ["Y", "y"]:  #
        if inv:  # Y -> Z
            qc.h(q[loc])
            qc.s(q[loc])
        else:  # Z -> Y
            qc.sdg(q[loc])
            qc.h(q[loc])
#this gadget performs exp(1j*val*pauli) over wavefunction
def pauli_gadget(qc, val, pauli):
    s, c = pauli, val
    scaling = -2.0
    if len(s) == 1:
        if s == "I":
            pass
        elif s == "X":
            qc.rx(val * scaling, q[0])
        elif s == "Y":
            qc.ry(val * scaling, q[0])
        elif s == "Z":
            qc.rz(val * scaling, q[0])
    else:
        pauliTerms = 0
        ind = []
        terms = []
        for n, i in enumerate(s):
            if not i in ["I"]:
                pauliTerms += 1
                ind.append(n)
                terms.append(i)
        if pauliTerms == 0:
            pass
        else:
            # basis
            for n, p in zip(ind, terms):
                pauli_string(qc, n, p)
            # exp cnot
            for n in range(0, pauliTerms - 1):
                qc.cx(q[ind[n]], q[ind[n + 1]])
            # parameter
            qc.rz(val * scaling, q[ind[-1]])
```

```
            # exp cnot
            for n in reversed(range(pauliTerms - 1)):
                qc.cx(q[ind[n]], q[ind[n + 1]])
            # inv. basis
            for n, p in zip(ind, terms):
                pauli_string(qc, n, p, inv=True)
#We import an example of the Jordan-Wigner mapped dictionary of H2 Hamiltonian␣
 ↪and verify the HF solution
p_dict_H2 = {"IIII": -0.53393635, "ZIII": 0.06727930, "IZII": 0.00665130,␣
 ↪"IIZI": 0.06727930,
"IIIZ": 0.00665130, "ZZII": 0.06501570, "ZIZI": 0.12736570, "XXXX": 0.06478462,
"YYXX": 0.06478462, "XXYY": 0.06478462, "YYYY": 0.06478462, "ZIIZ": 0.12980031,
"IZZI": 0.12980031, "IZIZ": 0.13366603, "IIZZ": 0.06501570}
for p_str in ["IYZX","ZYXI","XZYI"]:
    q,c,qc = init_qc(4)
    apply_pauli_string(qc, p_str)
    print (measure_from_dict(qc, p_dict_H2))
```

```
-0.78379264
-0.6653988599999999
-0.5412806400000001
```

[5]:
```
#Jordan-Wigner mapping
def JW_anni(pos, N=4):
    tensor_list = []
    for i in range(N):
        if i<pos :
            tensor_list.append(sigma_z)
        if i==pos :
            tensor_list.append((sigma_x+1j*sigma_y)/2)
        if i>pos :
            tensor_list.append(iden)
        anni_matrix = tensors(tensor_list)
    return anni_matrix
def JW_crea(pos, N=4):
    tensor_list = []
    for i in range(N):
        if i<pos :
            tensor_list.append(sigma_z)
        if i==pos :
            tensor_list.append((sigma_x-1j*sigma_y)/2)
        if i>pos :
            tensor_list.append(iden)
        crea_matrix = tensors(tensor_list)
    return crea_matrix
#the 2-rdm tensor is stored as a^{\dagger}_{i}a^{\dagger}_{j}a_{l}a_{k}
#the operator is appllied in the order of k,l,j,i
```

```
def rdmtomo(i,j,k,l):
    tomo = JW_anni(k)
    tomo = np.matmul(JW_anni(l), tomo)
    tomo = np.matmul(JW_crea(j), tomo)
    tomo = np.matmul(JW_crea(i), tomo)
    return tomo
#now we define the rdm decompose and sum
def tomo_decompose(N=4):
    tomo_string_real = []
    tomo_string_imag = []
    for i in range(N):
        for j in range(N):
            for k in range(N):
                for l in range(N):
                    tomo_dict = decomposeNqubit(rdmtomo(i,j,k,l), 4)
                    for item in tomo_dict:
                        if np.isreal(tomo_dict[item]):
                            if item not in tomo_string_real:
                                tomo_string_real.append(item)
                        else:
                            if item not in tomo_string_imag:
                                tomo_string_imag.append(item)
    return tomo_string_real, tomo_string_imag
def tomo_sum(mat, N=4):
    new_mat = np.zeros([2**N, 2**N], dtype = complex)
    for i in range(N):
        for j in range(N):
            for k in range(N):
                for l in range(N):
                    new_mat += mat[4*i+j,4*k+l] * rdmtomo(i,j,k,l)
    return new_mat
#this block measures the fermionic rdm from given circuit
def rdm2(qc, N=4):
    tomo_string_real, tomo_string_imag = tomo_decompose()
    tomo_string = tomo_string_real + tomo_string_imag
    tomo_values = []
    for string_to_measure in tomo_string:
        qc_copy = qc.copy()
        value_list_measured, string_list_measured = measure_abasis(qc_copy,␣
 ↪string_to_measure)
        #loop over the list
        for i in range(len(string_list_measured)):
            if string_list_measured[i] == string_to_measure:
                tomo_values.append(value_list_measured[i])
    rdm = np.zeros([2**N, 2**N], dtype = complex)
    for i in range(N):
        for j in range(N):
```

```
            for k in range(N):
                for l in range(N):
                    tomo_dict = decomposeNqubit(rdmtomo(i,j,k,l), 4)
                    for item in tomo_dict:
                        for m in range(len(tomo_string)):
                            if tomo_string[m] == item:
                                rdm[4*i+j,4*k+l] +=␣
↪tomo_values[m]*tomo_dict[item]
    return rdm
```

[6]:
```
#now we define the variance matrices
def var2_mat(p_dict, para, N=4):
    emat = np.zeros([2**N, 2**N])
    for item in p_dict:
        emat += p_dict[item] * str2mat(item).real
    vmat = emat - para*np.identity(16)
    v2mat = np.matmul(vmat, vmat)
    return v2mat
def propagate(qc, mat_dict):
    qc_copy = qc.copy()
    for item in mat_dict:
        pauli_gadget(qc_copy, mat_dict[item].real, item)
    rdm = rdm2(qc_copy)
    return rdm
def antihermitian(M):
    D = 0.5 * (M + np.conj(M.T))
    C = 0.5 * (M - np.conj(M.T))
    return C
```

[9]:
```
q,c,qc = init_qc(4)
apply_pauli_string(qc, "ZYXI")
j = 0
para_e = -0.5
#para_e = -0.7
while j < 10:
    print ("at iter ", j)
    para_e_iter = measure_from_dict(qc, p_dict_H2)
    print ("energy is ", para_e_iter)
    v2mat = var2_mat(p_dict_H2, para_e)
    v2_dict = decomposeNqubit(v2mat)
    var = measure_from_dict(qc, v2_dict)
    print ("variance is", var)
    rdmf = propagate(qc, v2_dict)
    v2_dict_inv = {}
    for i in v2_dict:
        v2_dict_inv[i] = -v2_dict[i]
    rdmb = propagate(qc, v2_dict_inv)
```

```
    F2_op = antihermitian(tomo_sum((rdmf - rdmb)/2j))
    qc.unitary(expm(F2_op),[3,2,1,0])
    v2mat_iter = var2_mat(p_dict_H2, para_e_iter)
    v2_dict_iter = decomposeNqubit(v2mat_iter)
    var_iter = measure_from_dict(qc, v2_dict_iter)
    if var_iter < var:
        para_e = para_e_iter
        var = var_iter
    j+=1
```

```
at iter  0
energy is  -0.6653988599999999
variance is (0.09450953470601+0j)
at iter  1
energy is  -0.5019743373780265
variance is (0.06715275181671043+0j)
at iter  2
energy is  -0.5019743373780265
variance is (0.040445177222490504+0j)
at iter  3
energy is  -0.42337420747142573
variance is (0.014754799806467864+0j)
at iter  4
energy is  -0.4082830074536329
variance is (0.0012719345040423519+0j)
at iter  5
energy is  -0.40626383655782394
variance is (5.868493439864508e-06+0j)
at iter  6
energy is  -0.4062632884474563
variance is (1.5073731472886598e-06+0j)
at iter  7
energy is  -0.40626346376863487
variance is (1.5982368628680854e-06+0j)
at iter  8
energy is  -0.40626345294860766
variance is (1.5926290196366377e-06+0j)
at iter  9
energy is  -0.40626345363440985
variance is (1.5929844509321445e-06+0j)
```

[ ]: