Combining the contracted quantum eigensolver with the Rayleigh-Ritz variational principle for mixed states for the computation of quantum excited states

November 1, 2023

This is the demonstration jupyter notebook of "Combining the contracted quantum eigensolver with the Rayleigh-Ritz variational principle for mixed states for the computation of quantum excited states"

[ ]:
```
from qiskit import QuantumCircuit,QuantumRegister,ClassicalRegister,execute
from qiskit.visualization import array_to_latex as a2l
import numpy as np
import itertools
from qiskit.providers.fake_provider import FakeLagosV2

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)
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)
```

```
#define the basis change gates
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)
def execute_qc_list(qc_list, backend_name = FakeLagosV2, nshots=8192):
    result = execute(qc_list, backend = backend_name(), shots = nshots).result()
    counts = result.get_counts()
    return counts
#perform a measurement of all string combinations
def tomo_qc(qc):
    qc_list = []
    elements = ["I", "X", "Y"]
    tomo_list = [''.join(i) for i in itertools.product(elements, repeat = 2)]
    for j in range(len(tomo_list)):
        qc_copy = qc.copy()
        for pos,i in enumerate(tomo_list[j]):
            #print (i, pos)
            if i == 'X' :
                qc_copy.h(pos) #convert qubit to X
            elif i == 'Y' :
                qc_copy.z(pos) #convert qubit to Y
                qc_copy.s(pos)
                qc_copy.h(pos)
        qc_copy.measure([0,1], [0,1])
        qc_list.append(qc_copy)
    return tomo_list, qc_list
#compressed tomography
tomo_list = []
tomo_list.append(str2mat("XY")*1j)
tomo_list.append(str2mat("YX")*1j)
#binary dictionary of Pauli string in z basis
def z_to_binary(z_string, N=2):
    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
def proceed_counts(counts, N=2):
    #runs over all counts to get tol counts
    tol = 0
    for keys in counts:
        tol += counts[keys]
    z_label = ['Z', 'I']
    z_string = [''.join(i) for i in itertools.product(z_label, repeat = N)]
    z_dict = {}
    for i in z_string:
        b_val = 0
        binary_dict = z_to_binary(i)
        for key in binary_dict:
            for key2 in counts:
                if key == key2[-2:][::-1]:
                    b_val += binary_dict[key]*counts[key2]/tol
        z_dict[i] = b_val
    return z_dict
#given the circuit, return the dict with changed basis
def tomo_dict_generator(qc):
    tomo_str = {}
    tomo_list, qc_list = tomo_qc(qc)
    counts = execute_qc_list(qc_list)
    for i in range(len(counts)):
        z_dict = proceed_counts(counts[i])
        for item in z_dict:
            new_dict = {}
            new_str = []
            for j in range(2):
            #print (tomo_list[i][j])
                if item[j]=="Z" and tomo_list[i][j]=="X":
                    new_str.append("X")
                elif item[j]=="Z" and tomo_list[i][j]=="Y":
                    new_str.append("Y")
                else:
                    new_str.append(item[j])
            tomo_str["".join(new_str)]=z_dict[item]
    return tomo_str
#measure expectation value from pauli dictionary
def assemble_from_tomo(h_dict_2q, tomo_dict):
    e = 0
    for item in h_dict_2q:
        e += tomo_dict[item]*h_dict_2q[item]
    return e
#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=2):
    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
# Calculate the Hermitian part (H) and anti-Hermitian part (A)
def decompose_hermitian_antihermitian(matrix):
    H = 0.5 * (matrix + np.conj(matrix.T))
    A = 0.5 * (matrix - np.conj(matrix.T))
    return H, A
```

[130]:
```
#Hamiltonian
h_dict_2q = {"II": -0.28794507760149823, "XX":0.17900057606140662, "IZ":0.
 ↪4204556797828042,
                "ZI":0.4204556797828042,"ZZ":0.01150740217682722}
hmat = np.zeros([4,4], dtype=complex)
for i in h_dict_2q:
    hmat += str2mat(i) * h_dict_2q[i]

#initialize circuit based on weight
def init_qc(sv_in):
    q = QuantumRegister(4)
    c = ClassicalRegister(4)
    qc = QuantumCircuit(q,c)
    qc.initialize(sv_in, [0,1])
    qc.cx(0,2)
    qc.cx(1,3)
    return q,c,qc
sv_in = np.array([9,1,9,1])
sv_in = sv_in/np.linalg.norm(sv_in)
q,c,qc = initial_qc(sv_in)
print ("exact ensemble energy is", sum([sorted(sv_in)[::-1][i]**2*sorted(np.
 ↪linalg.eig(hmat)[0])[i] for i in range(4)]).real)

stepsize=0.1
iters = 0
total_dict = {}
while iters < 10:
    print ("Begin iteration", iters+1)
    qc_f = qc.copy()
    qc_b = qc.copy()
```

```
    #imaginary propagation
    for ops in h_dict_2q:
        pauli_gadget(qc_f, stepsize*h_dict_2q[ops].real, ops)
        pauli_gadget(qc_b, -stepsize*h_dict_2q[ops].real, ops)
    tomo_f = tomo_dict_generator(qc_f)
    tomo_b = tomo_dict_generator(qc_b)
    a2_mat = np.zeros([4,4], dtype=complex)
    for tomo in tomo_list:
        sq_dict = decomposeNqubit(tomo,2)
        a2f = assemble_from_tomo(sq_dict, tomo_f)
        a2b = assemble_from_tomo(sq_dict, tomo_b)
        a2 = (a2f-a2b)/2j/stepsize
        a2_mat += a2*tomo
    a2h, a2 = decompose_hermitian_antihermitian(a2_mat)
    a2_dict = decomposeNqubit(a2,2)
    in_iter_energy = []
    in_iter_epsilon = []
    #perform a fixed step line search
    for epsilon in np.arange(0, 1.0, 0.1):
        qc_v = qc.copy()
        for item in a2_dict:
            pauli_gadget(qc_v, (-1j*epsilon*a2_dict[item]).real, item)
        tomo_v = tomo_dict_generator(qc_v)
        in_iter_energy.append(assemble_from_tomo(h_dict_2q, tomo_v))
        in_iter_epsilon.append(epsilon)
    min_val = min(in_iter_energy)
    min_pos = [pos for pos, val in enumerate(in_iter_energy) if val == min_val]
    print ("  At iteration", iters+1, "the minimum ensemble energy is:", np.
↪round(min_val.real,8), "corresponding epsilon is:", np.
↪round(in_iter_epsilon[min_pos[0]],3))
    #apply to original circuit and compose
    q,c,qc = initial_qc(sv_in)
    for item in a2_dict:
        if item not in total_dict:
            total_dict[item]=in_iter_epsilon[min_pos[0]]*a2_dict[item]
        else:
            total_dict[item]+=in_iter_epsilon[min_pos[0]]*a2_dict[item]
    for item in total_dict:
        pauli_gadget(qc, (-1j*total_dict[item]).real, item)
    iters+=1
```

```
exact ensemble energy is -0.7946535433101121
Begin iteration 1
  At iteration 1 the minimum ensemble energy is: -0.04115976 corresponding
epsilon is: 0.9
Begin iteration 2
  At iteration 2 the minimum ensemble energy is: -0.72371967 corresponding
epsilon is: 0.9
```

```
Begin iteration 3
  At iteration 3 the minimum ensemble energy is: -0.74813793 corresponding
epsilon is: 0.3
Begin iteration 4
  At iteration 4 the minimum ensemble energy is: -0.76703332 corresponding
epsilon is: 0.7
Begin iteration 5
  At iteration 5 the minimum ensemble energy is: -0.77065849 corresponding
epsilon is: 0.0
Begin iteration 6
  At iteration 6 the minimum ensemble energy is: -0.760798 corresponding epsilon
is: 0.0
Begin iteration 7
  At iteration 7 the minimum ensemble energy is: -0.76670697 corresponding
epsilon is: 0.0
Begin iteration 8
  At iteration 8 the minimum ensemble energy is: -0.76308363 corresponding
epsilon is: 0.2
Begin iteration 9
  At iteration 9 the minimum ensemble energy is: -0.7605818 corresponding
epsilon is: 0.0
Begin iteration 10
  At iteration 10 the minimum ensemble energy is: -0.76561431 corresponding
epsilon is: 0.2
```

[ ]: