Simulating Molecules using Variational Quantum Eigensolver

Project Report Submitted

for Fulfillment of the Requirements

for the course PH312 (Mini-Project)

by

Gadgi Mihir Rajesh

(190121008)

under the guidance of

Prof. Tapan Mishra

to the

Department of Physics

Indian Institute of Technology, Guwahati

Guwahati - 781039, Assam

Contents

Chapter 1

Introduction

In simulating the chosen molecules, H_2 and LiH, I will try to implement VQE which stands for Variational Quantum Eigensolver.

1.1 Historical background

During the last decade, quantum computers matured quickly and began to realize Feynman's initial dream of a computing system that could simulate the laws of nature in a quantum way. A 2014 paper first authored by Alberto Peruzzo introduced the Variational Quantum Eigensolver (VQE), an algorithm meant for finding the ground state energy (lowest energy) of a molecule, with much shallower circuits than other approaches. And, in 2017, the IBM Quantum team used the VQE algorithm to simulate the ground state energy of the lithium hydride molecule.[3]

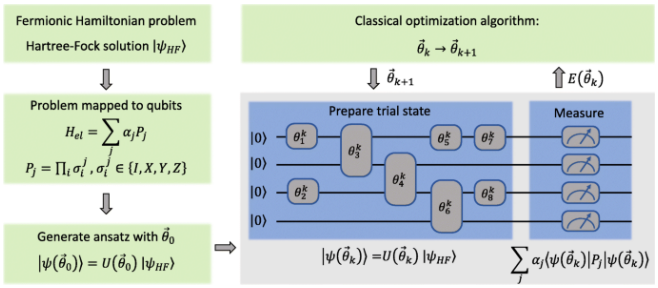
VQE's magic comes from outsourcing some of the problem's processing workload to a classical computer. The algorithm starts with a parameterized quantum circuit called an ansatz (a best guess) then finds the optimal parameters for this circuit using a classical optimizer. The VQE's advantage over classical algorithms comes from the fact that a quantum processing unit can represent and store the problem's exact wavefunction, an exponentially hard problem for a classical computer.[1]

1.2 Brief description of Variational Quantum Eigensolver

VQE is a hybrid algorithm. This means that its computational load is shared between the hardwares of Quantum Computer and the Classical Computer. We start by reasonably assuming the form of the target wave function. Commonly, this is done by representing a wave function in a basis of atom-centered Gaussian basis function.

Then we assume a trial wave function (ansatz) with adjustable parameters, then we design a quantum circuit which can realise this wave function. The ansatz parameters are varied and adjusted until we are able to minimise the expectation value of the Electronic Hamiltonian.

$$E \leq \frac{\langle \psi(\vec{\theta}) | \hat{H}_{electronic} | \psi(\vec{\theta}) \rangle}{\langle \psi(\vec{\theta}) | \psi(\vec{\theta}) \rangle}$$



In 1.1 we begin by adjusting the variabloe parameter θ. In the figure, the green color regions denote use of classical computing and the blue region shows areas where Quantum Computer is used to realise the trial wave function.

We begin the simulation by constructing the Fermionic Hamiltonian and finding mean-field solution $|\psi_{HF}\rangle$.

This is now mapped into Qubit Hamiltionian, which can be represented as :

$$H = \sum_j \alpha_j \prod_i \sigma_i^j$$

Then we chose the ansatz which represents the wave function and we initialize it with initial set of parameters, $\vec{\theta_0}$, and a trial state is then prepared on a quantum computer as a quantum circuit consisting of parametrized gates. We repeat this procedure many times until the convergence criterion is met.

So basically, at some n^{th} iteration, we compute the energy of by measuring every Hamiltoninan term $\langle (\vec{\theta_n}) | P_j | (\vec{\theta_n}) \rangle$ on the quantum computer and then we add them on classical computers. We need to

feed the energy, $E(\vec{\theta_k})$ in the classical algorithm which will then update parameters for the next step of optimization $\vec{\theta_{k+1}}$ according to the optimization algorithm we chose to consider.

Chapter 2

The Hamiltonian

2.1 Hartree-Fock Approximation

2.1.1 The Schrödinger equation of the Coulomb Hamiltonian

$$H = -\frac{\hbar^2}{2M_{\text{tot}}}\nabla_{\mathbf{X}}^2 + H' \quad \text{with} \quad H' = -\frac{\hbar^2}{2}\sum_{i=1}^{N_{\text{tot}}-1}\frac{1}{m_i}\nabla_i^2 + \frac{\hbar^2}{2M_{\text{tot}}}\sum_{i,j=1}^{N_{\text{tot}}-1}\nabla_i \cdot \nabla_j + V(\mathbf{t})$$

2.1.2 Fock Operator

The molecular Hamiltonian consists of an electron-electron repulsion term, this needs to involve the coordinates of both the electrons, so we need an approximation in order to avoid cumbersome calculations.

Under Hartree-Fock Approximation, the terms of the exact Hamiltonian (except nuclea- nuclear repulsion terms) are re-expressed as the sum of one-electron operators outlined below, for closed-shell atoms or molecules (with two electrons in each spatial orbital).[2]

$$\hat{F}[\{\phi_j\}](1) = \hat{H}^{\text{core}}(1) + \sum_{j=1}^{N/2}[2\hat{J}_j(1) - \hat{K}_j(1)]$$

here,

$$\hat{F}[\{\phi_j\}](1)$$

is the one electron Fock operator that is generated by ϕ_j, and

$$\hat{H}^{\text{core}}(1) = -\frac{1}{2}\nabla_1^2 - \sum_{\alpha}\frac{Z_\alpha}{r_{1\alpha}}$$

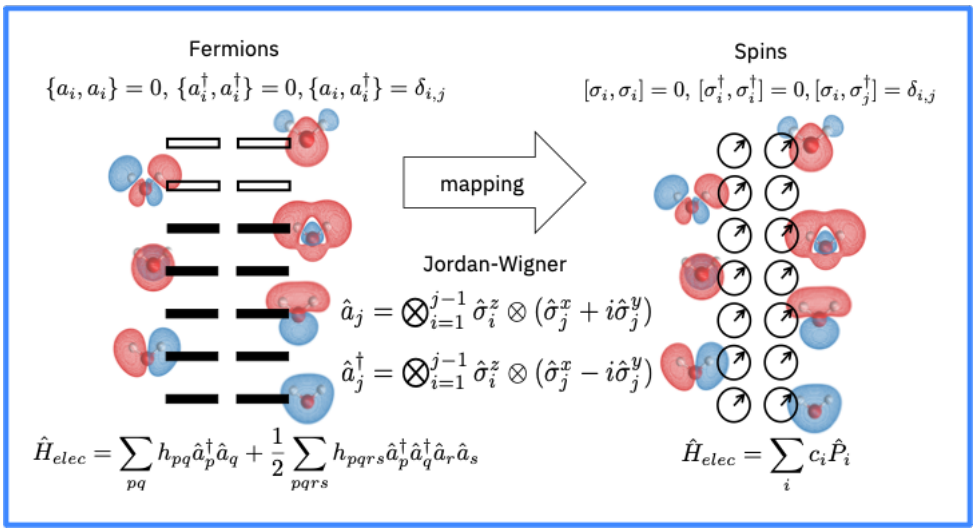
is the one electron core Hamiltonian. And $\hat{J}_j(1)$ is the Coulomb operator, defining the electron–electron repulsion energy due to each of the two electrons in the j-th orbital.[2] Finally, $\hat{K}_j(1)$ is the exchange operator, defining the electron exchange energy due to the antisymmetry of the total N-electron wave function.[2] This "exchange energy" operator $\hat{K}$ is simply an artifact of the Slater determinant. Finding the Hartree–Fock one-electron wave functions is now equivalent to solving the eigenfunction equation

$$\hat{F}(1)\phi_i(1) = \epsilon_i\phi_i(1)$$

where $\phi_i(1)$ are a set of one-electron wave functions, called the Hartree–Fock molecular orbitals.

2.2 Jordan - Wigner Transformation

Jordan- Wigner Transformation helps us map spin operators onto Fermionic creation and annhilation operators. An interesting fact being, that Jordan - Wigner Transformation doesn't distinguish between spin -1/2 particles and Fermions. The picture 2.1 explains the transformation/mapping.



2.3 Molecular Hamiltonian

We need to define and explain operators that we need in order to measure the energy of the given system.

$$\hat{H} = \sum_{rs} h_{rs} \hat{a}_r^\dagger \hat{a}_s + \frac{1}{2} \sum_{pqrs} g_{pqrs} \hat{a}_p^\dagger \hat{a}_q^\dagger \hat{a}_r \hat{a}_s + E_{NN}$$

where,

$$h_{pq} = \int \phi_p^*(r) \left(-\frac{1}{2}\nabla^2 - \sum_I \frac{Z_I}{R_I - r} \right) \phi_q(r)$$
$$g_{pqrs} = \int \frac{\phi_p^*(r_1)\, \phi_q^*(r_2)\, \phi_r(r_2)\, \phi_s(r_1)}{|r_1 - r_2|}$$

where the h_{rs} and g_{pqrs} are the one-/two-body integrals (using the Hartree-Fock method) and E_{NN} the nuclear repulsion energy. The one-body integrals represent the kinetic energy of the electrons and their interaction with nuclei. The two-body integrals represent the electron-electron interaction.
The $\hat{a}_r^\dagger, \hat{a}_r$ operators represent creation and annihilation of electron in spin-orbital r and require mappings to operators, to measure them on a quantum computer..

So, for every non-zero matrix element in the h_{rs} and g_{pqrs} tensors, we can construct corresponding Pauli string (tensor product of Pauli operators) with the following fermion-to-qubit transformation. For instance, in Jordan-Wigner mapping for an orbital $r = 3$, we obtain the following Pauli string:

$$\hat{a}_3^\dagger = \hat{\sigma}_z \otimes \hat{\sigma}_z \otimes \left(\frac{\hat{\sigma}_x - i\hat{\sigma}_y}{2} \right) \otimes 1 \otimes \cdots \otimes 1$$

where $\hat{\sigma}_x, \hat{\sigma}_y, \hat{\sigma}_z$ are the well-known Pauli operators. The tensor products of $\hat{\sigma}_z$ operators are placed to enforce the fermionic anti-commutation relations.

Chapter 3

Simulating H_2 molecule

3.1 Brief description of the simulation Process

3.1.1 Mapping

For the purpose of H_2 molecule, we will be using Jordan-Wigner Mapping (Transformation)(2.2) in order to map the fermions to Qubits.

3.1.2 Ansatz

For H_2 molecule, we will be using Two Local ansatz.
So, basically, the Two Local Ansatz consists of alternating rotation layers and entanglement layers. It is a paramatrized circuit that uses single qubit rotation gates across each individual qubit and two qubit entanglement gates placed in a ladder format which are typically CNOT/CCX gates.

The ccx gate can be represented as the following matrix transformation:

$$CCX = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

And, also, for giving a better insight, the rotation matrix $R_y(\theta)$, can be represented by the following

single qubit transformation:

$$R_y(\theta) = \begin{bmatrix} cos\left(\frac{\theta}{2}\right) & -sin\left(\frac{\theta}{2}\right) \\ sin\left(\frac{\theta}{2}\right) & cos\left(\frac{\theta}{2}\right) \end{bmatrix}$$

3.1.3 Classical Optimizer

COBYLA stands for Constrained Optimization By Linear Approximation.

Usually, COBYLA optimizer is used in cases where the derivative of the objective function is unknown.

It is a numerical optimization method.

[1]:
```
import numpy as np
from qiskit import QuantumCircuit, transpile, Aer, IBMQ
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
from qiskit.providers.aer import QasmSimulator
import matplotlib as mpl
```

```
<frozen importlib._bootstrap>:219: RuntimeWarning:
scipy._lib.messagestream.MessageStream size changed, may indicate binary
incompatibility. Expected 56 from C header, got 64 from PyObject
```

[2]:
```
from qiskit_nature.drivers import PySCFDriver
molecule = "H .0 .0 .0; H .0 .0 0.739"
driver = PySCFDriver(atom=molecule)
qmolecule = driver.run()
```

```
/tmp/ipykernel_452/1640696031.py:3: DeprecationWarning: The PySCFDriver class is
deprecated as of version 0.2.0 and will be removed no sooner than 3 months after
the release. Instead use the PySCFDriver class from
qiskit_nature.drivers.second_quantization.pyscfd.
  driver = PySCFDriver(atom=molecule)
```

[3]:
```
from qiskit_nature.problems.second_quantization.electronic import␣
↪ElectronicStructureProblem
problem = ElectronicStructureProblem(driver)
```

```
second_q_ops = problem.second_q_ops()
main_op = second_q_ops[0]
```

[4]:
```
from qiskit_nature.mappers.second_quantization import ParityMapper,␣
 ↪BravyiKitaevMapper, JordanWignerMapper
from qiskit_nature.converters.second_quantization.qubit_converter import␣
 ↪QubitConverter
mapper = JordanWignerMapper()
converter = QubitConverter(mapper=mapper, two_qubit_reduction=False)
num_particles = (problem.molecule_data_transformed.num_alpha,
             problem.molecule_data_transformed.num_beta)
qubit_op = converter.convert(main_op, num_particles=num_particles)
```

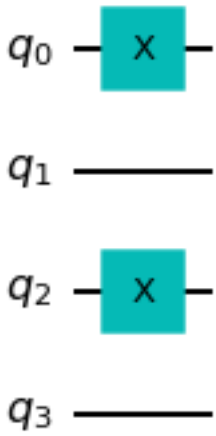
```
/tmp/ipykernel_452/962686619.py:5: DeprecationWarning: The
molecule_data_transformed property is deprecated as of version 0.2.0 and will be
removed no sooner than 3 months after the release. Instead use the
grouped_property_transformed property.
  num_particles = (problem.molecule_data_transformed.num_alpha,
```

Initial State We will start by considering $\psi = |0101\rangle$ as the initial state.

[5]:
```
from qiskit_nature.circuit.library import HartreeFock
num_particles = (problem.molecule_data_transformed.num_alpha,
             problem.molecule_data_transformed.num_beta)
num_spin_orbitals = 2 * problem.molecule_data_transformed.num_molecular_orbitals
init_state = HartreeFock(num_spin_orbitals, num_particles, converter)
```

[6]:
```
init_state.draw('mpl')
```

[6]:

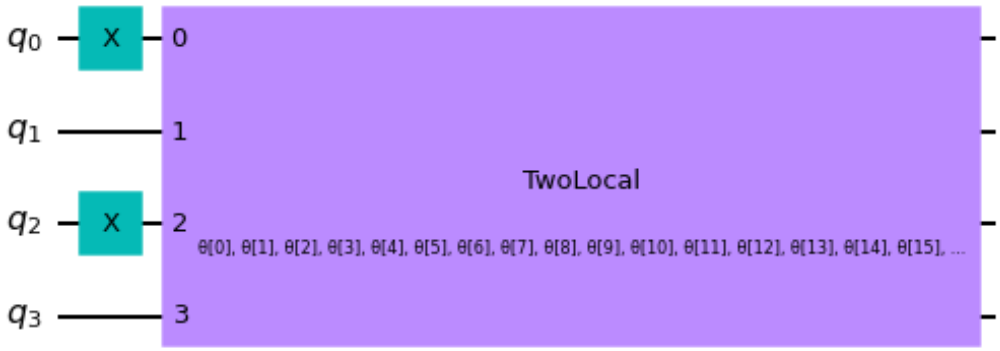


Ansatz We will use two-local ansatz as the default ansatz for this problem as it is readily available in the qiskit library and easy to compose.

[7]:
```
from qiskit.circuit.library import TwoLocal
from qiskit_nature.circuit.library import UCCSD, PUCCD, SUCCD
num_particles = (problem.molecule_data_transformed.num_alpha,
             problem.molecule_data_transformed.num_beta)
num_spin_orbitals = 2 * problem.molecule_data_transformed.num_molecular_orbitals
from qiskit.circuit import Parameter, QuantumCircuit, QuantumRegister
rotation_blocks = ['ry', 'rz']
entanglement_blocks = 'cx'
entanglement = 'full'
repetitions = 3
skip_final_rotation_layer = True
ansatz = TwoLocal(qubit_op.num_qubits, rotation_blocks, entanglement_blocks,␣
 ↪reps=repetitions,
                  entanglement=entanglement,␣
 ↪skip_final_rotation_layer=skip_final_rotation_layer)
# adding the initial state
ansatz.compose(init_state, front=True, inplace=True)
```

[8]:
```
ansatz.draw('mpl')
```

[8]:



[9]:
```
from qiskit import Aer
backend = Aer.get_backend('statevector_simulator')
```

[10]:
```
from qiskit.algorithms.optimizers import COBYLA, L_BFGS_B, SPSA, SLSQP
optimizer = COBYLA(maxiter=500)
```

Exact Energy Here I've shown some results for our chosen molecule, which are the exact values for different parameters of the molecule.

[11]:
```
from qiskit_nature.algorithms.ground_state_solvers.minimum_eigensolver_factories␣
↪import NumPyMinimumEigensolverFactory
from qiskit_nature.algorithms.ground_state_solvers import GroundStateEigensolver

def exact_diagonalizer(problem, converter):
    solver = NumPyMinimumEigensolverFactory()
    calc = GroundStateEigensolver(converter, solver)
    result = calc.solve(problem)
    return result

result_exact = exact_diagonalizer(problem, converter)
exact_energy = np.real(result_exact.eigenenergies[0])
print("Exact electronic energy", exact_energy)
print(result_exact)
```

```
Exact electronic energy -1.8533636186720357
=== GROUND STATE ENERGY ===

* Electronic ground state energy (Hartree): -1.853363618672
  - computed part:      -1.853363618672
~ Nuclear repulsion energy (Hartree): 0.716072003951
> Total ground state energy (Hartree): -1.137291614721

=== MEASURED OBSERVABLES ===

  0:  # Particles: 2.000 S: 0.000 S^2: 0.000 M: 0.000

=== DIPOLE MOMENTS ===

~ Nuclear dipole moment (a.u.): [0.0  0.0  1.39650761]

  0:
  * Electronic dipole moment (a.u.): [0.0  0.0  1.39650761]
    - computed part:      [0.0  0.0  1.39650761]
  > Dipole moment (a.u.): [0.0  0.0  0.0]  Total: 0.0
                  (debye): [0.0  0.0  0.00000001]  Total: 0.00000001
```

Calculating Energy for H_2 using VQE and finding errors in our calculation.

[12]:
```python
from qiskit.algorithms import VQE
from IPython.display import display, clear_output


def callback(eval_count, parameters, mean, std):
    display("Evaluation: {}, Energy: {}, Std: {}".format(eval_count, mean, std))
    clear_output(wait=True)
    counts.append(eval_count)
    values.append(mean)
    params.append(parameters)
    deviation.append(std)
```

```
counts = []
values = []
params = []
deviation = []
try:
    initial_point = [0.01] * len(ansatz.ordered_parameters)
except:
    initial_point = [0.01] * ansatz.num_parameters

algorithm = VQE(ansatz,
                optimizer=optimizer,
                quantum_instance=backend,
                callback=callback,
                initial_point=initial_point)

result = algorithm.compute_minimum_eigenvalue(qubit_op)

print(result)
```

```
{   'aux_operator_eigenvalues': None,
    'cost_function_evals': 500,
    'eigenstate': array([-8.01967738e-06+5.35021714e-05j,
6.78005082e-06+9.49056742e-05j,
        8.56961045e-06-1.44705532e-05j,  7.77754647e-05-4.74281620e-05j,
        1.57195558e-06-3.49200780e-07j,  9.48583307e-01+2.95900636e-01j,
        1.04027944e-03-1.80342805e-03j,  7.63572001e-08-1.42746754e-07j,
        1.86016557e-09+4.59114937e-08j, -9.39806181e-04+1.22919986e-03j,
       -1.07258614e-01-3.34862821e-02j, -1.36549702e-05-4.21335684e-06j,
       -8.40583917e-06+5.30799761e-06j, -7.70667653e-05+1.24317568e-04j,
       -9.80019416e-06-3.76518704e-06j, -1.00450272e-04-2.76953620e-05j]),
    'eigenvalue': (-1.85335941174982+0j),
    'optimal_parameters': {   ParameterVectorElement(Theta[23]): 0.3073020843708068,
                              ParameterVectorElement(Theta[22]): 0.5601039278118176,
                              ParameterVectorElement(Theta[21]): 0.8545808908288351,
                              ParameterVectorElement(Theta[20]):
```

-0.18680534907828777,
ParameterVectorElement(Theta[19]):
-0.0310403035987273,
ParameterVectorElement(Theta[18]):
0.026260747676010623,
ParameterVectorElement(Theta[17]): 3.1414137352538263,
ParameterVectorElement(Theta[16]):
0.003925607256799522,
ParameterVectorElement(Theta[15]):
0.16827786844713383,
ParameterVectorElement(Theta[14]):
-0.3511158485296716,
ParameterVectorElement(Theta[13]):
-0.2591436715750023,
ParameterVectorElement(Theta[12]): 1.0719645711475825,
ParameterVectorElement(Theta[11]):
0.01684157310432731,
ParameterVectorElement(Theta[10]):
-0.0006395852100755067,
ParameterVectorElement(Theta[9]):
-0.00020015939270742514,
ParameterVectorElement(Theta[8]):
-8.66125440230164e-05,
ParameterVectorElement(Theta[7]): 0.1154245162347783,
ParameterVectorElement(Theta[6]): 0.29069417955200433,
ParameterVectorElement(Theta[5]):
-0.06520651820449927,
ParameterVectorElement(Theta[2]):
0.025456784238420814,
ParameterVectorElement(Theta[1]):
2.8773182068247005e-05,
ParameterVectorElement(Theta[3]):
0.047721895731622874,
ParameterVectorElement(Theta[4]):
-0.06608680310943157,

```
                                ParameterVectorElement(Theta[0]):
0.22528230755418927},
    'optimal_point': array([ 2.25282308e-01,  2.87731821e-05,  2.54567842e-02,
4.77218957e-02,
       -6.60868031e-02, -6.52065182e-02,  2.90694180e-01,  1.15424516e-01,
       -8.66125440e-05, -2.00159393e-04, -6.39585210e-04,  1.68415731e-02,
        1.07196457e+00, -2.59143672e-01, -3.51115849e-01,  1.68277868e-01,
        3.92560726e-03,  3.14141374e+00,  2.62607477e-02, -3.10403036e-02,
       -1.86805349e-01,  8.54580891e-01,  5.60103928e-01,  3.07302084e-01]),
    'optimal_value': -1.85335941174982,
    'optimizer_evals': None,
    'optimizer_time': 2.6633033752441406}
```

[13]:
```
#calculating percentage error
E_th = -1.8533636186720357
E_vqe = -1.85335941174982


print("The percentage error in our calculation is " + str(np.abs(E_th-E_vqe)/E_th␣
 ↪*100)+" %")
```

```
The percentage error in our calculation is 0.00022698849666174086 %
```

Chapter 4

Simulating LiH molecule

Here, we will be using the same type of ansatz that was used for H_2 molecule.

4.0.1 Freeze Core Transformer

Freeze core transformer determines the "core" orbitals automatically and removes them or makes them inactive.
This is especially useful in cases where computation is heavy and we could reduce the number of qubits which in turn reduces the number of parameters that need to be individually computed which in turn reduces the computational load and makes the problem easier to solve.

The LiH molecule is symmetric along z-axis. The 1s of Li orbital has core electrons that can be frozen as they do not participate in bonding. In the ground state of this molecule, mostly the electrons from H 1s and Li 2s orbitals interact while mixing with 2 due to the shape and orientation of the molecule. We can use Freeze Core transformer to our advantage in order to eliminate these electrons.

[1]:
```
import numpy as np
#standard Qiskit libraries
from qiskit import QuantumCircuit, transpile, Aer, IBMQ
from qiskit.tools.jupyter import *
from qiskit.visualization import *
from ibm_quantum_widgets import *
```

```
from qiskit.providers.aer import QasmSimulator
```

```
<frozen importlib._bootstrap>:219: RuntimeWarning:
scipy._lib.messagestream.MessageStream size changed, may indicate binary
incompatibility. Expected 56 from C header, got 64 from PyObject
```

[2]:
```
from qiskit_nature.drivers import PySCFDriver
from qiskit_nature.transformers import FreezeCoreTransformer
molecule = 'Li 0.0 0.0 0.0; H 0.0 0.0 1.5474'
driver = PySCFDriver(atom=molecule)
qmolecule = driver.run()
```

```
/tmp/ipykernel_530/162381677.py:4: DeprecationWarning: The PySCFDriver class is
deprecated as of version 0.2.0 and will be removed no sooner than 3 months after
the release. Instead use the PySCFDriver class from
qiskit_nature.drivers.second_quantization.pyscfd.
  driver = PySCFDriver(atom=molecule)
```

[3]:
```
fct = [FreezeCoreTransformer(freeze_core=True, remove_orbitals=[3,4])]
```

```
/tmp/ipykernel_530/2902793591.py:1: DeprecationWarning: The
FreezeCoreTransformer class is deprecated as of version 0.2.0 and will be
removed no sooner than 3 months after the release. Instead use the
FreezeCoreTransformer class from
qiskit_nature.transformers.second_quantization.electronic as a direct
replacement.
  fct = [FreezeCoreTransformer(freeze_core=True, remove_orbitals=[3,4])]
```

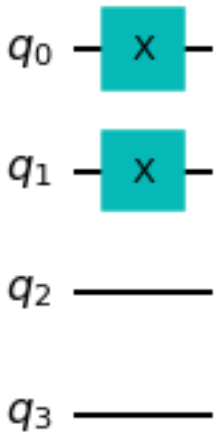
[4]:
```
from qiskit_nature.problems.second_quantization.electronic import␣
↪ElectronicStructureProblem
problem = ElectronicStructureProblem(driver, fct)
second_q_ops = problem.second_q_ops()

# Hamiltonian
main_op = second_q_ops[0]
from qiskit_nature.mappers.second_quantization import ParityMapper
```

```
from qiskit_nature.converters.second_quantization.qubit_converter import␣
 ↪QubitConverter


converter = QubitConverter(mapper=ParityMapper(), two_qubit_reduction=True)


# Mapping Fermions to Qubits
num_particles = (problem.molecule_data_transformed.num_alpha,
             problem.molecule_data_transformed.num_beta)
qubit_op = converter.convert(main_op, num_particles=num_particles)
from qiskit_nature.circuit.library import HartreeFock


num_particles = (problem.molecule_data_transformed.num_alpha,
             problem.molecule_data_transformed.num_beta)
num_spin_orbitals = 2 * problem.molecule_data_transformed.num_molecular_orbitals
init_state = HartreeFock(num_spin_orbitals, num_particles, converter)
```

```
/tmp/ipykernel_530/3100622809.py:13: DeprecationWarning: The
molecule_data_transformed property is deprecated as of version 0.2.0 and will be
removed no sooner than 3 months after the release. Instead use the
grouped_property_transformed property.
  num_particles = (problem.molecule_data_transformed.num_alpha,
```

[5]:
```
init_state.draw('mpl')
```

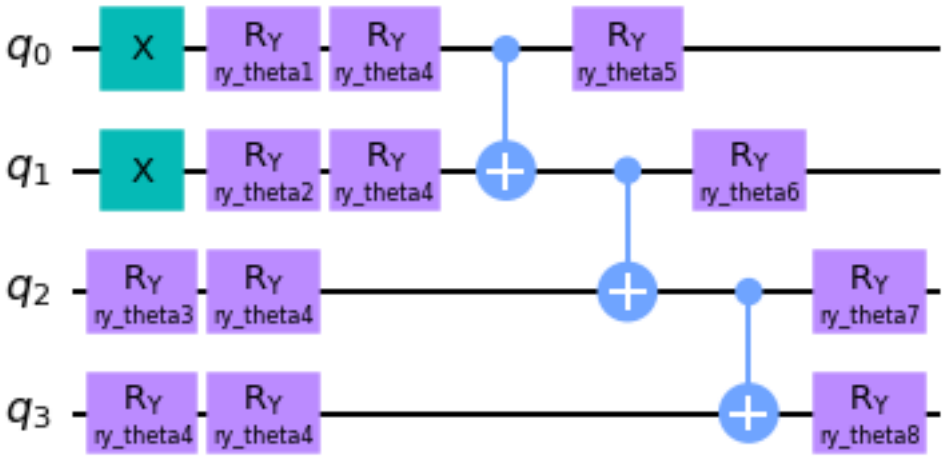
[5]:



[6]:
```
from qiskit.circuit.library import TwoLocal
from qiskit_nature.circuit.library import UCCSD, PUCCD, SUCCD


# Parameters for q-UCC antatze
num_particles = (problem.molecule_data_transformed.num_alpha,
             problem.molecule_data_transformed.num_beta)
num_spin_orbitals = 2 * problem.molecule_data_transformed.num_molecular_orbitals
n = qubit_op.num_qubits
qc = QuantumCircuit(qubit_op.num_qubits)
from qiskit.circuit import Parameter, QuantumCircuit, QuantumRegister
#the variational parameter
p=1
for i in range(n):
    theta = Parameter(f"ry_theta{p}" )
    qc.ry(theta, i)
    p += 1
qubit_label = 0
qc.ry(theta, range(n))
#qc.rz(theta, range(n))
for i in range(n-1):
    qc.cx(i, i+1)
for i in range(n):
    theta = Parameter(f"ry_theta{p}" )
    qc.ry(theta, i)
    p += 1
#qc.rz(theta, range(n))


# Add the initial state
ansatz = qc
ansatz.compose(init_state, front=True, inplace=True)
```

[7]:
```
ansatz.draw('mpl')
```

[7]:



[8]:
```
from qiskit_nature.algorithms.ground_state_solvers.minimum_eigensolver_factories␣
 ↪import NumPyMinimumEigensolverFactory
from qiskit_nature.algorithms.ground_state_solvers import GroundStateEigensolver
import numpy as np

def exact_diagonalizer(problem, converter):
    solver = NumPyMinimumEigensolverFactory()
    calc = GroundStateEigensolver(converter, solver)
    result = calc.solve(problem)
    return result

result_exact = exact_diagonalizer(problem, converter)
exact_energy = np.real(result_exact.eigenenergies[0])
print("Exact electronic energy", exact_energy)
print(result_exact)
```

```
Exact electronic energy -1.0887060157347386
=== GROUND STATE ENERGY ===

* Electronic ground state energy (Hartree): -8.907396311316
  - computed part:      -1.088706015735
```

```
  - FreezeCoreTransformer extracted energy part: -7.818690295581
~ Nuclear repulsion energy (Hartree): 1.025934879643
> Total ground state energy (Hartree): -7.881461431673

=== MEASURED OBSERVABLES ===

  0:  # Particles: 2.000 S: 0.000 S^2: 0.000 M: 0.000

=== DIPOLE MOMENTS ===

~ Nuclear dipole moment (a.u.): [0.0  0.0  2.92416221]

  0:
  * Electronic dipole moment (a.u.): [0.0  0.0  4.76300889]
    - computed part:      [0.0  0.0  4.76695575]
    - FreezeCoreTransformer extracted energy part: [0.0  0.0  -0.00394686]
  > Dipole moment (a.u.): [0.0  0.0  -1.83884668]  Total: 1.83884668
                  (debye): [0.0  0.0  -4.67388163]  Total: 4.67388163
```

So here, our desired solution/ final answer is -1.0887060157347386.

[9]:
```
from qiskit import Aer
backend = Aer.get_backend('statevector_simulator')
from qiskit.algorithms.optimizers import COBYLA, L_BFGS_B, SPSA, SLSQP
optimizer = COBYLA(maxiter=15000)
from qiskit.algorithms import VQE
from IPython.display import display, clear_output
def callback(eval_count, parameters, mean, std):
    display("Evaluation: {}, Energy: {}, Std: {}".format(eval_count, mean, std))
    clear_output(wait=True)
    counts.append(eval_count)
    values.append(mean)
    params.append(parameters)
    deviation.append(std)
```

```
counts = []
values = []
params = []
deviation = []
try:
    initial_point = [0.01] * len(ansatz.ordered_parameters)
except:
    initial_point = [0.01] * ansatz.num_parameters

algorithm = VQE(ansatz,
                optimizer=optimizer,
                quantum_instance=backend,
                callback=callback,
                initial_point=initial_point)

result = algorithm.compute_minimum_eigenvalue(qubit_op)

print(result)
```

```
{   'aux_operator_eigenvalues': None,
    'cost_function_evals': 953,
    'eigenstate': array([ 8.87906754e-04-3.76609208e-17j,
1.02394364e-04+4.82162587e-17j,
       -6.13421787e-04+3.76012282e-17j, -9.99998777e-01-1.70549165e-16j,
       -7.57739984e-07+3.27740140e-20j, -5.48348401e-05-4.86617658e-20j,
        5.48739278e-07-3.27184140e-20j,  8.15308456e-04+1.41689298e-19j,
       -2.88512155e-08+1.22422771e-21j, -4.57875743e-08-1.57254357e-21j,
        1.99518120e-08-1.22228447e-21j,  3.24639378e-05+5.53875094e-21j,
        6.88658841e-07-2.92097625e-20j,  8.12980815e-08+3.73966995e-20j,
       -4.75769674e-07+2.91634649e-20j, -7.75595898e-04-1.32277485e-19j]),
    'eigenvalue': (-1.0703581417404309+0j),
    'optimal_parameters': {   Parameter(ry_theta6): -1.5683228417176245,
                              Parameter(ry_theta7): -1.5724272563127628,
                              Parameter(ry_theta8): -1.570862519851208,
                              Parameter(ry_theta5): 0.663121580533992,
```

```
                                Parameter(ry_theta3): 0.7860642300485193,
                                Parameter(ry_theta4): 0.7846225936906919,
                                Parameter(ry_theta2): 0.7890537853257208,
                                Parameter(ry_theta1): -0.12027197208481556},
    'optimal_point': array([-0.12027197,  0.78905379,  0.78606423,  0.78462259,
0.66312158,
       -1.56832284, -1.57242726, -1.57086252]),
    'optimal_value': -1.0703581417404309,
    'optimizer_evals': None,
    'optimizer_time': 8.1324942111969}
```

And our final result is -1.0703581417404309.

Our result is not very accurate. The reason being that LiH is a complicated molecule for this particular problem. Also, the code written was run on a classical computer which simulated the results of a quantum computer. We can say that, the error is caused due to errors in simulation process and difficulty in optimising the parameters and errors caused by the classical optimizer.

Calculating Error in Results

[10]:
```
E_th = -1.0887060157347446
E_vqe = -1.0703581417404309
```

[11]:
```
print("The percentage error in our calculation is " + str(np.abs(E_th-E_vqe)/E_th␣
↪*100)+" %")
```

```
The percentage error in our calculation is 1.6852918721066412 %
```

Chapter 5

Conclusion

We could successfully simulate both, the Hydrogen molecule and the Lithium Hydride Molecule.

5.0.1 H_2 Molecule

The error in our calculation of Lowest Eigenvalue for H_2 using VQE was 0.00022698849666174086 percent.
This is a very small error. The reason being that H_2 is a simple molecule and it was computationally very easy to optimize the parameters to reach convergence.

5.0.2 LiH Molecule

The error in our calculation of Lowest Eigenvalue for LiH using VQE was 1.6852918721066412 percent.
The error is quite significant when we consider the fact that sometimes we need precise values for research purposes.
Some possible reasons that might cause this error are already discussed.

Bibliography

[1] A. Kandla. Hardware-efficient variational quantum eigensolver for small molecules and quantum magnets. *Nature Communications*, 549.7671:242-246, 2017.

[2] I. N. Levine. *Quantum Chemistry*. Englewood Cliffs, New Jersey: Prentice Hall., 4 edition, 1991.

[3] A. Peruzzo. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*, 5(1):1-7, 2014.