

What I've done is I've constructed the following dictionary to allow the user to choose the initial and final hamiltonians.

Initial: A basic operator whose ground state is the HF state.

There is something that needs fixing. $|001\rangle$ can be associated with IIZ , but $|011\rangle$ cannot be associated with IZZ . You'd need a minus sign. This is too crude an initial Hamiltonian, so a slightly wiser choice for the initial Hamiltonian could be the HF energy times this operator we've constructed.

So, this is wrong, and I'm about to fix it. We'll delve into how we defined the initial operator.

```

1      hamiltonian_methods = {
2          'initial': {
3              'qiskit_hf': {
4                  'description': 'Use Qiskit Hartree-Fock method to generate Initial Hamiltonian',
5                  'generate': lambda molecule, taper, freeze_core: MoleculeClass(molecule, taper, freeze_core).get_hartreefock_in_pauli()
6              },
7              'qiskit_hf_and_energy': {
8                  'description': 'Use Qiskit Hartree-Fock method to generate Initial Hamiltonian',
9                  'generate': lambda molecule, taper, freeze_core: MoleculeClass(molecule, taper, freeze_core).get_hartreefock_energy()* MoleculeClass(molecule, taper, f
10             },
11             'paper': {
12                 'description': 'Use Hartree-Fock from a specific paper (custom implementation)',
13                 'generate': lambda molecule, taper, freeze_core: IBM_LiH_initial
14             }
15         },
16         'final': {
17             'qiskit': {
18                 'description': 'Use final Hamiltonian from a paper-specific method',
19                 'generate': lambda molecule, taper, freeze_core: MoleculeClass(molecule, taper, freeze_core).get_qubit_operator()
20             },
21             'paper': {
22                 'description': 'Use Qiskit for final Hamiltonian (if different from initial)',
23                 'generate': lambda molecule, taper, freeze_core: IBM_LiH
24             }
25         }
26     }
27 
```

This is how we converted into Pauli:

```

1      def get_hartreefock_in_pauli(self):
2          # Get the number of spatial orbitals (i.e., the number of qubits in the mapping)
3          problem = self.electronic_structure_problem
4          # Get the Hartree-Fock state
5          hf_state = HartreeFock(problem.num_spatial_orbitals, problem.num_particles, JordanWignerMapper())
6          state_vector = Statevector(hf_state)
7          binary_string = state_vector.probabilities_dict()
8          binary_string = get_string_from_dict(binary_string)
9          pauli_op = create_z_operator_from_binary_string(binary_string)
10         return pauli_op

```

Now, we've defined an operator whose eigenvalue is zero for the HF state, and one otherwise:

```

1      def get_hartreefock_in_projector(self):
2          # Get the number of spatial orbitals (i.e., the number of qubits in the mapping)
3          problem = self.electronic_structure_problem
4          # Get the Hartree-Fock state
5          hf_state = HartreeFock(problem.num_spatial_orbitals, problem.num_particles, JordanWignerMapper())
6
7          state_vector = Statevector(hf_state)

```

```

8         state_vector.to_operator()
9         # Identity operator (same size as the projector, which is typically 2^n for n qubits)
10        identity = Operator(np.eye(projector.num_qubits))
11
12        # Identity minus projector
13        identity_minus_projector = identity - projector
14
15        return identity_minus_projector # This is the operator I - P

```

We've modified the dictionary:

```

1      hamiltonian_methods = {
2          'initial': {
3              'qiskit_hf': {
4                  'description': 'This is a simple Hartree-Fock Hamiltonian.',
5                  'generate': lambda molecule, taper, freeze_core: MoleculeClass(molecule, taper, freeze_core).get_hartreefock_in_projector()
6              },
7              'qiskit_hf_and_energy': {
8                  'description': 'This is a simple Hartree-Fock Hamiltonian, multiplied by the Hartree-fock energy.',
9                  'generate': lambda molecule, taper, freeze_core: MoleculeClass(molecule, taper, freeze_core).get_hartreefock_energy()* MoleculeClass(molecule, taper, freeze_core).get_hartreefock_in_projector()
10              },
11
12              'paper': {
13                  'description': 'Use Hartree-Fock from a specific paper (custom implementation)',
14                  'generate': lambda molecule, taper, freeze_core: IBM_LiH_initial
15              }
16          },
17          'final': {
18              'qiskit': {
19                  'description': 'Use final Hamiltonian from a paper-specific method',
20                  'generate': lambda molecule, taper, freeze_core: MoleculeClass(molecule, taper, freeze_core).get_qubit_operator()
21              },
22              'paper': {
23                  'description': 'Use Qiskit for final Hamiltonian (if different from initial)',
24                  'generate': lambda molecule, taper, freeze_core: IBM_LiH
25              }
26          }
27      }

```

I'm feeling this is a bit dubious- the Operator doesn't seem defined. So, I'm going to test it. The best way to test it is to ask the alternative run method to output the initial hamiltonian.

```

1      def get_hartreefock_in_projector(self):
2          # Get the number of spatial orbitals (i.e., the number of qubits in the mapping)
3          problem = self.electronic_structure_problem
4          # Get the Hartree-Fock state
5          hf_state = HartreeFock(problem.num_spatial_orbitals, problem.num_particles, JordanWignerMapper())
6          state_vector = Statevector(hf_state)
7          state_vector.to_operator()
8          # Identity operator (same size as the projector, which is typically 2^n for n qubits)
9          identity = Operator(np.eye(projector.num_qubits))
10         # Identity minus projector
11         identity_minus_projector = identity - projector
12         return identity_minus_projector # This is the operator I - P

```

Now, I've fixed a few major bugs with the code. I've simplified the projector. I will use sparse pauli to come up with I-projector.

```

1      def get_hartreefock_in_projector(self):
2          # Get the number of spatial orbitals (i.e., the number of qubits in the mapping)
3          problem = self.electronic_structure_problem
4          # Get the Hartree-Fock state

```

```

5 hf_state = HartreeFock(problem.num_spatial_orbitals,problem.num_particles,JordanWignerMapper())
6 state_vector= Statevector(hf_state)
7 projector=state_vector.to_operator()
8
9 return projector

```

We start with running the code, and that is the `alternative_run` bit.

```

1 from aqc_pqc import AQC_PQC
2 from hamiltonian import Hamiltonian
3 from quantum_circuit import QCir
4 import networkx as nx
5 import numpy as np
6 from brute_force import Brute_Force
7 from qaoa_circuit import QAOA_Circuit
8 from aqc_qaoa import AQC_PQC_QAOA
9 from Quantum_Chemistry import Moleculeclass,Solvebynumpy
10 from aavqe import *
11 from qiskit_algorithms.utils import algorithm_globals
12 from qiskit_aer import Aer
13 backend = Aer.get_backend('statevector_simulator')
14 seed = 3
15 number_of_qubits = 8
16 steps = 10#Choose number of steps to interpolate from initial to final Hamiltonian
17 connectivity = 'nearest-neighbors' #This is the connectivity of the non-parameterized gates in the Hardware-Efficient ansatz
18 single_qubit_gates = 'ry'
19 entanglement_gates = 'cz'
20 layers = 1
21 entanglement = 'linear'
22 hfstate=Moleculeclass(molecule,taper,freeze_core).get_hartreefock()
23 myaavqe=My_AAVQE(number_of_qubits,steps,layers,single_qubit_gates,entanglement_gates,entanglement,'qiskit_hf','qiskit')
24 myaavqe.alternative_run()

```

Here we've initialised My AAVQE.

Line 6 pulls the appropriate operator given the definition. This needs to be checked- that the initial operator is defined correctly. It is used later in the method on line 31, and I got an error on line 34. The method on line 31 takes angles and observables as arguments, each of which gets fed through the `alternative_run` method on line 39. Specifically, on line 42.

```

1 def __init__(self, number_of_qubits, steps, layers, single_qubit_gates, entanglement_gates, entanglement,initial_hamiltonian,target_hamiltonian,initial_state=None):
2     self.number_of_qubits = number_of_qubits
3     self.initial_state=initial_state
4     self.steps = steps
5     self.string_initial_hamiltonian=initial_hamiltonian
6     self.initial_hamiltonian=hamiltonian_methods['initial'][initial_hamiltonian]['generate'](molecule,taper,freeze_core)
7     self.string_final_hamiltonian=target_hamiltonian
8     self.offset=0
9     self.layers = layers
10    self.single_qubit_gates = single_qubit_gates
11    self.entanglement_gates = entanglement_gates
12    self.entanglement = entanglement
13    if self.string_initial_hamiltonian == 'transverse':
14        X_tuples = []
15        for i in range(number_of_qubits):
16            X_tuples.append(('X', [i], -1))
17    self.initial_hamiltonian = SparsePauliOp.from_sparse_list([X_tuples], num_qubits = number_of_qubits)
18    elif self.string_initial_hamiltonian == 'paper':
19        self.initial_parameters=[0 for x in range(self.number_of_qubits*(self.layers+1))]
20    self.initial_parameters[6]=np.pi
21    self.initial_parameters[7]=np.pi
22    else:
23        self.initial_parameters=[0 for x in range(self.number_of_qubits*(self.layers+1))]
24        self.initial_parameters[8]=np.pi
25        self.initial_parameters[12]=np.pi

```

```

26     self.target_hamiltonian=hamiltonian_methods['final'][target_hamiltonian]['generate'](molecule, taper, freezeccore)
27     print(self.initial_hamiltonian)
28     self.qcir = TwoLocal(self.number_of_qubits, self.single_qubit_gates, self.entanglement_gates, self.entanglement, self.layers, initial_state= self.initial_state)
29     self.number_of_parameters = len(self.initial_parameters)
30
31     def get_expectation_value(self, angles, observable):
32         estimator = StatevectorEstimator()
33         pub = (self.qcir, observable, angles)
34         job = estimator.run([pub])
35         result = job.result()[0]
36         expectation_value = result.data.evs
37         return np.real(expectation_value)
38
39     def alternative_run(self):
40         lambdas = [i for i in np.linspace(0, 1, self.steps+1)][1:]
41         optimal_thetas = self.initial_parameters.copy()
42         instantaneous_expectation_value=self.get_expectation_value(optimal_thetas,self.initial_hamiltonian)
43         initial_ground_state=self.minimum_eigenvalue(self.initial_hamiltonian)
44         energies_aavqe = [instantaneous_expectation_value]
45         energies_exact = [initial_ground_state]
46         print(f'We start with the optimal angles of the initial hamiltonian: {optimal_thetas}')
47         for lamda in lambdas:
48             print('\n')
49             hamiltonian = self.get_instantaneous_hamiltonian(lamda)
50             minimization_object = optimize.minimize(self.get_expectation_value, x0=optimal_thetas, args=(hamiltonian), method='SLSQP')
51             optimal_thetas = minimization_object.x
52             print(f'We are working on {lamda} where the current optimal point is {optimal_thetas}')
53             self.offset=0
54             inst_exp_value = self.get_expectation_value(optimal_thetas, hamiltonian) - lamda*self.offset
55             energies_aavqe.append(inst_exp_value)
56             energies_exact.append(self.minimum_eigenvalue(hamiltonian) - lamda*self.offset)
57             print(f'and the instantaneous expectation values is {inst_exp_value}')
58             print(f'and the true expectation value is {self.minimum_eigenvalue(hamiltonian) - lamda*self.offset}')
59             plt.plot(energies_aavqe,label='aavqe energy')
60             plt.plot(energies_exact,label='true energy')
61             plt.legend()
62             plt.xlabel('time')
63             plt.ylabel('energy (Ha)')
64             plt.title(f'{self.string_initial_hamiltonian} and {self.string_final_hamiltonian}')
65             plt.show()
66             return energies_aavqe

```

We start with running the code, and that is the alternative_run bit.

```

1     from aqc_pqc import AQC_PQC
2     from hamiltonian import Hamiltonian
3     from quantum_circuit import QCir
4     import networkx as nx
5     import numpy as np
6     from brute_force import Brute_Force
7     from qaoa_circuit import QAOA_Circuit
8     from aqc_qaoa import AQC_PQC_QAOA
9     from Quantum_Chemistry import Moleculeclass,Solvebynumpy
10    from aavqe import *
11    from qiskit_algorithms.utils import algorithm_globals
12    from qiskit_aer import Aer
13    backend = Aer.get_backend('statevector_simulator')
14    seed = 3
15    number_of_qubits = 8
16    steps = 10#Choose number of steps to interpolate from initial to final Hamiltonian
17    connectivity = 'nearest-neighbors' #This is the connectivity of the non-parameterized gates in the Hardware-Efficient ansatz
18    single_qubit_gates = 'ry'
19    entanglement_gates = 'cz'
20    layers = 1
21    entanglement = 'linear'
22    hfstate=Moleculeclass(molecule,taper,freezeccore).get_hartreefock()
23    myaavqe=My_AAVQE(number_of_qubits,steps,layers,single_qubit_gates,entanglement_gates,entanglement,'qiskit_hf','qiskit')
24    myaavqe.alternative_run()

```

Here we've initialised My AAVQE.

Line 6 pulls the appropriate operator given the definition. This needs to be checked- that the initial operator is defined correctly. It is used later in the method on line 31, and I got an error on line 34. The method on line 31 takes angles and observables as arguments, each of which gets fed through the `alternative_run` method on line 39. Specifically, on line 42.

```

1     def __init__(self, number_of_qubits, steps, layers, single_qubit_gates, entanglement_gates, entanglement, initial_hamiltonian, target_hamiltonian, initial_state=None):
2         self.number_of_qubits = number_of_qubits
3         self.initial_state = initial_state
4         self.steps = steps
5         self.string_initial_hamiltonian = initial_hamiltonian
6         self.initial_hamiltonian = hamiltonian_methods['initial'][initial_hamiltonian]['generate'](molecule, taper, freeze_core)
7         self.string_final_hamiltonian = target_hamiltonian
8         self.offset = 0
9         self.layers = layers
10        self.single_qubit_gates = single_qubit_gates
11        self.entanglement_gates = entanglement_gates
12        self.entanglement = entanglement
13        if self.string_initial_hamiltonian == 'transverse':
14            X_tuples = []
15            for i in range(number_of_qubits):
16                X_tuples.append(('X', [i], -1))
17            self.initial_hamiltonian = SparsePauliOp.from_sparse_list([X_tuples], num_qubits = number_of_qubits)
18        elif self.string_initial_hamiltonian == 'paper':
19            self.initial_parameters = [0 for x in range(self.number_of_qubits*(self.layers+1))]
20            self.initial_parameters[6] = np.pi
21            self.initial_parameters[7] = np.pi
22        else:
23            self.initial_parameters = [0 for x in range(self.number_of_qubits*(self.layers+1))]
24            self.initial_parameters[8] = np.pi
25            self.initial_parameters[12] = np.pi
26        self.target_hamiltonian = hamiltonian_methods['final'][target_hamiltonian]['generate'](molecule, taper, freeze_core)
27        print(self.initial_hamiltonian)
28        self.qcirq = TwoLocal(self.number_of_qubits, self.single_qubit_gates, self.entanglement_gates, self.entanglement, self.layers, initial_state = self.initial_state)
29        self.number_of_parameters = len(self.initial_parameters)
30
31    def get_expectation_value(self, angles, observable):
32        estimator = StatevectorEstimator()
33        pub = (self.qcirq, observable, angles)
34        job = estimator.run([pub])
35        result = job.result()[0]
36        expectation_value = result.data.evs
37        return np.real(expectation_value)
38
39    def alternative_run(self):
40        lambdas = [i for i in np.linspace(0, 1, self.steps+1)][1:]
41        optimal_thetas = self.initial_parameters.copy()
42        instantaneous_expectation_value = self.get_expectation_value(optimal_thetas, self.initial_hamiltonian)
43        initial_ground_state = self.minimum_eigenvalue(self.initial_hamiltonian)
44        energies_aavqe = [instantaneous_expectation_value]
45        energies_exact = [initial_ground_state]
46        print(f'We start with the optimal angles of the initial hamiltonian: {optimal_thetas}')
47        for lamda in lambdas:
48            print('\n')
49            hamiltonian = self.get_instantaneous_hamiltonian(lamda)
50            minimization_object = optimize.minimize(self.get_expectation_value, x0=optimal_thetas, args=(hamiltonian), method='SLSQP')
51            optimal_thetas = minimization_object.x
52            print(f'We are working on {lamda} where the current optimal point is {optimal_thetas}')
53            self.offset = 0
54            inst_exp_value = self.get_expectation_value(optimal_thetas, hamiltonian) - lamda*self.offset
55            energies_exact.append(inst_exp_value)
56            energies_aavqe.append(self.minimum_eigenvalue(hamiltonian) - lamda*self.offset)
57            print(f'and the instantaneous expectation values is {inst_exp_value}')
58            print(f'and the true expectation value is {self.minimum_eigenvalue(hamiltonian) - lamda*self.offset}')
59            plt.plot(energies_aavqe, label='aavqe energy')
60            plt.plot(energies_exact, label='true energy')
61            plt.legend()
62            plt.xlabel('time')
63            plt.ylabel('energy (Ha)')
64            plt.title(f'{self.string_initial_hamiltonian} and {self.string_final_hamiltonian}')
65            plt.show()
66        return energies_aavqe

```

Just to check that the projector is working correctly, and so that we are on track to getting the correct initial hamiltonian, we've run this code to check which element the Hamiltonian is projecting onto. It turns out the only non-zero element of the matrix is the (17,17)th element, which is what we expect. The Hartree fock state is indeed $|00010001\rangle$.

```

1      myaavqe=My_AAVQE(number_of_qubits,steps,layers,single_qubit_gates,entanglement_gates,entanglement,'qiskit_hf','qiskit',hfstate)
2      print(myaavqe.initial_hamiltonian)
3      H=myaavqe.initial_hamiltonian
4      # Assuming 'H' is your operator
5      matrix = H.to_matrix()
6      # Find non-zero elements in the matrix
7      non_zero_indices = np.nonzero(matrix) # Indices of non-zero elements
8      non_zero_values = matrix[non_zero_indices] # Non-zero values
9      print(hfstate)
10     # Print the non-zero elements and their positions
11     print("Non-zero elements:")
12     for idx, value in zip(zip(*non_zero_indices), non_zero_values):
13         print(f"Index: {idx}, Value: {value}")
14
15     # Assuming 'H' is your operator and has been converted to a matrix form
16     matrix = H.to_matrix()
17
18     # Compute eigenvalues and eigenvectors using numpy.linalg.eig
19     eigenvalues, eigenvectors = np.linalg.eig(matrix)
20
21     # Find the index of the minimum eigenvalue
22     min_index = np.argmin(eigenvalues)
23
24     # Get the corresponding minimum eigenvector
25     min_eigenvector = eigenvectors[:, min_index]
26
27     # Normalize the eigenvector (optional but recommended)
28     min_eigenvector = min_eigenvector / np.linalg.norm(min_eigenvector)
29
30     # Print the minimum eigenvalue and corresponding eigenvector
31     print(f"Minimum eigenvalue: {eigenvalues[min_index]}")
32     print(f"Minimum eigenvector: {min_eigenvector}")
33     #myaavqe.initial_hamiltonian()
34     #myaavqe.alternative_run()
35     #print(myaavqe.draw_latex())

```

Consider the following

```

1  def get_hartreefock_in_pauli(self):
2      # Get the number of spatial orbitals (i.e., the number of qubits in the mapping)
3      problem = self.electronic_structure_problem
4      # Get the Hartree-Fock state
5      hf_state = HartreeFock(problem.num_spatial_orbitals, problem.num_particles, JordanWignerMapper())
6
7      # Create the statevector for the Hartree-Fock state
8      state_vector = Statevector(hf_state)
9
10     # Get the probabilities dictionary for the state
11     binary_string = state_vector.probabilities_dict()
12
13     # Initialize an empty list to store the Pauli operator terms
14     Z_tuples = []
15
16     # Loop over each binary string in the probabilities dictionary
17     for bin_str, _ in binary_string.items():
18         # Loop over each bit in the binary string
19         for i, bit in enumerate(bin_str):
20             if bit == '0':
21                 Z_tuples.append(('Z', [i], -1)) # Append Z for bit 0
22             elif bit == '1':
23                 Z_tuples.append(('Z', [i], 1)) # Append Z for bit 1
24
25     # Convert the list of Pauli operators and coefficients into a SparsePauliOp
26     hamiltonian = SparsePauliOp.from_sparse_list([*Z_tuples], num_qubits = len(Z_tuples))
27     # print(f'the binary string is {binary_string}')
28     return hamiltonian

```
