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.

```
        hamiltonian_methods = {
                'initial': {
                        'qiskit_hf': {
                                'description': 'Use Qiskit Hartree-Fock method to generate Initial Hamiltonian',
                                'generate': lambda molecule, taper, freezecore: Moleculeclass(molecule, taper, freezecore).get_hartreefock_in_pauli()
                        },
                        'qiskit_hf_and_energy': {
                                'description': 'Use Qiskit Hartree-Fock method to generate Initial Hamiltonian',
                                'generate': lambda molecule, taper, freezecore: Moleculeclass(molecule, taper, freezecore).get_hartreefock_energy()* Moleculeclass(molecule, taper, f
                        },

                        'paper': {
                                'description': 'Use Hartree-Fock from a specific paper (custom implementation)',
                                'generate': lambda molecule, taper, freezecore:IBM_LiH_initial
                        }
                },
                'final': {
                        'qiskit': {
                                'description': 'Use final Hamiltonian from a paper-specific method',
                                'generate': lambda molecule, taper, freezecore: Moleculeclass(molecule, taper,freezecore).get_qubit_operator()
                        },
                        'paper': {
                                'description': 'Use Qiskit for final Hamiltonian (if different from initial)',
                                'generate': lambda molecule, taper, freezecore: IBM_LiH
                        }
                }
        }
```

This is how we converted into Pauli:

```
        def get_hartreefock_in_pauli(self):
                # Get the number of spatial orbitals (i.e., the number of qubits in the mapping)
                problem= self.electronic_structure_problem
                # Get the Hartree-Fock state
                hf_state = HartreeFock(problem.num_spatial_orbitals,problem.num_particles,JordanWignerMapper())
                state_vector= Statevector(hf_state)
                binary_string=state_vector.probabilities_dict()
                binary_string= get_string_from_dict(binary_string)
                pauli_op= create_z_operator_from_binary_string(binary_string)
                return pauli_op
```

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

```
        hamiltonian_methods = {
                'initial': {
                        'qiskit_hf': {
                                'description': 'This is a simple Hartree-Fock Hamiltonian.',
                                'generate': lambda molecule, taper, freezecore: Moleculeclass(molecule, taper, freezecore).get_hartreefock_in_projector()
                        },
                        'qiskit_hf_and_energy': {
```

```
                'description': 'This is a simple Hartree-Fock Hamiltonian, multiplied by the Hartree-fock energy.',
                'generate': lambda molecule, taper, freezecore: Moleculeclass(molecule, taper, freezecore).get_hartreefock_energy()* Moleculeclass(molecule, taper, f
            },

            'paper': {
                'description': 'Use Hartree-Fock from a specific paper (custom implementation)',
                'generate': lambda molecule, taper, freezecore:IBM_LiH_initial
            }
        },
        'final': {
            'qiskit': {
                'description': 'Use final Hamiltonian from a paper-specific method',
                'generate': lambda molecule, taper, freezecore: Moleculeclass(molecule, taper,freezecore).get_qubit_operator()
            },
            'paper': {
                'description': 'Use Qiskit for final Hamiltonian (if different from initial)',
                'generate': lambda molecule, taper, freezecore: IBM_LiH
            }
        }
    }
```

We've modified the dictionary:

```
    hamiltonian_methods = {
        'initial': {
            'qiskit_hf': {
                'description': 'This is a simple Hartree-Fock Hamiltonian.',
                'generate': lambda molecule, taper, freezecore: Moleculeclass(molecule, taper, freezecore).get_hartreefock_in_projector()
            },
            'qiskit_hf_and_energy': {
                'description': 'This is a simple Hartree-Fock Hamiltonian, multiplied by the Hartree-fock energy.',
                'generate': lambda molecule, taper, freezecore: Moleculeclass(molecule, taper, freezecore).get_hartreefock_energy()* Moleculeclass(molecule, taper, f
            },

            'paper': {
                'description': 'Use Hartree-Fock from a specific paper (custom implementation)',
                'generate': lambda molecule, taper, freezecore:IBM_LiH_initial
            }
        },
        'final': {
            'qiskit': {
                'description': 'Use final Hamiltonian from a paper-specific method',
                'generate': lambda molecule, taper, freezecore: Moleculeclass(molecule, taper,freezecore).get_qubit_operator()
            },
            'paper': {
                'description': 'Use Qiskit for final Hamiltonian (if different from initial)',
                'generate': lambda molecule, taper, freezecore: IBM_LiH
            }
        }
    }
```

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.

```
    def get_hartreefock_in_projector(self):
        # Get the number of spatial orbitals (i.e., the number of qubits in the mapping)
        problem= self.electronic_structure_problem
        # Get the Hartree-Fock state
        hf_state = HartreeFock(problem.num_spatial_orbitals,problem.num_particles,JordanWignerMapper())
        state_vector= Statevector(hf_state)
        state_vector.to_operator()
        # Identity operator (same size as the projector, which is typically 2^n for n qubits)
        identity = Operator(np.eye(projector.num_qubits))
        # Identity minus projector
        identity_minus_projector = identity - projector
        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.

```
        def get_hartreefock_in_projector(self):
                # Get the number of spatial orbitals (i.e., the number of qubits in the mapping)
                problem= self.electronic_structure_problem
                # Get the Hartree-Fock state
                hf_state = HartreeFock(problem.num_spatial_orbitals,problem.num_particles,JordanWignerMapper())

                state_vector= Statevector(hf_state)
                state_vector.to_operator()
                # Identity operator (same size as the projector, which is typically 2^n for n qubits)
                identity = Operator(np.eye(projector.num_qubits))

                # Identity minus projector
                identity_minus_projector = identity - projector

                return identity_minus_projector  # This is the operator I - P
```

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

```
        def __init__(self, number_of_qubits, steps, layers, single_qubit_gates, entanglement_gates, entanglement,initial_hamiltonian,target_hamiltonian,initial_state=None):
        self.number_of_qubits = number_of_qubits
        self.initial_state=initial_state
        self.steps = steps
        self.string_initial_hamiltonian=initial_hamiltonian
        self.initial_hamiltonian=hamiltonian_methods['initial'][initial_hamiltonian]['generate'](molecule,taper,freezecore)
        self.string_final_hamiltonian=target_hamiltonian
        self.offset=0
        self.layers = layers
        self.single_qubit_gates = single_qubit_gates
        self.entanglement_gates = entanglement_gates
        self.entanglement = entanglement
        if self.string_initial_hamiltonian == 'transverse':
        X_tuples = []
        for i in range(number_of_qubits):
        X_tuples.append(('X', [i], -1))
        self.initial_hamiltonian = SparsePauliOp.from_sparse_list([*X_tuples], num_qubits = number_of_qubits)
        elif  self.string_initial_hamiltonian == 'paper':
        self.initial_parameters=[0 for x in range(self.number_of_qubits*(self.layers+1))]
        self.initial_parameters[6]=np.pi
        self.initial_parameters[7]=np.pi
        else:
        self.initial_parameters=[0 for x in range(self.number_of_qubits*(self.layers+1))]
        self.initial_parameters[8]=np.pi
        self.initial_parameters[12]=np.pi
        self.target_hamiltonian=hamiltonian_methods['final'][target_hamiltonian]['generate'](molecule, taper, freezecore)
        print(self.initial_hamiltonian)
        self.qcir = TwoLocal(self.number_of_qubits, self.single_qubit_gates, self.entanglement_gates, self.entanglement, self.layers,initial_state= self.initial_state)
        self.number_of_parameters = len(self.initial_parameters)

        def get_expectation_value(self, angles, observable):
        estimator = StatevectorEstimator()
        pub = (self.qcir, observable, angles)
        job = estimator.run([pub])
        result = job.result()[0]
        expectation_value = result.data.evs
        return np.real(expectation_value)

        def alternative_run(self):
        lambdas = [i for i in np.linspace(0, 1, self.steps+1)][1:]
        optimal_thetas = self.initial_parameters.copy()
        instantaneous_expectation_value=self.get_expectation_value(optimal_thetas,self.initial_hamiltonian)
        initial_ground_state=self.minimum_eigenvalue(self.initial_hamiltonian)
        energies_aavqe = [instantaneous_expectation_value]
        energies_exact = [initial_ground_state]
        print(f'We start with the optimal angles of the initial hamiltonian: {optimal_thetas}')
        for lamda in lambdas:
        print('\n')
        hamiltonian = self.get_instantaneous_hamiltonian(lamda)
```

```
        minimization_object = optimize.minimize(self.get_expectation_value, x0=optimal_thetas, args=(hamiltonian), method='SLSQP')
        optimal_thetas = minimization_object.x
        print(f'We are working on {lamda} where the current optimal point is {optimal_thetas}')
        self.offset=0
        inst_exp_value = self.get_expectation_value(optimal_thetas, hamiltonian) - lamda*self.offset
        energies_aavqe.append(inst_exp_value)
        energies_exact.append(self.minimum_eigenvalue(hamiltonian) - lamda*self.offset)
        print(f'and the instantaneous expectation values is {inst_exp_value}')
        print(f'and the true expectation value is {self.minimum_eigenvalue(hamiltonian) - lamda*self.offset}')
        plt.plot(energies_aavqe,label='aavqe energy')
        plt.plot(energies_exact,label='true energy')
        plt.legend()
        plt.xlabel('time')
        plt.ylabel('energy (Ha)')
        plt.title(f'{self.string_initial_hamiltonian} and {self.string_final_hamiltonian}')
        plt.show()
        return energies_aavqe
```

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,freezecore)
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, freezecore)
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]
```

```
        energies_exact = [initial_ground_state]
        print(f'We start with the optimal angles of the initial hamiltonian: {optimal_thetas}')
        for lamda in lambdas:
        print('\n')
        hamiltonian = self.get_instantaneous_hamiltonian(lamda)
        minimization_object = optimize.minimize(self.get_expectation_value, x0=optimal_thetas, args=(hamiltonian), method='SLSQP')
        optimal_thetas = minimization_object.x
        print(f'We are working on {lamda} where the current optimal point is {optimal_thetas}')
        self.offset=0
        inst_exp_value = self.get_expectation_value(optimal_thetas, hamiltonian) - lamda*self.offset
        energies_aavqe.append(inst_exp_value)
        energies_exact.append(self.minimum_eigenvalue(hamiltonian) - lamda*self.offset)
        print(f'and the instantaneous expectation values is {inst_exp_value}')
        print(f'and the true expectation value is {self.minimum_eigenvalue(hamiltonian) - lamda*self.offset}')
        plt.plot(energies_aavqe,label='aavqe energy')
        plt.plot(energies_exact,label='true energy')
        plt.legend()
        plt.xlabel('time')
        plt.ylabel('energy (Ha)')
        plt.title(f'{self.string_initial_hamiltonian} and {self.string_final_hamiltonian}')
        plt.show()
        return energies_aavqe
```

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

```
        def __init__(self, number_of_qubits, steps, layers, single_qubit_gates, entanglement_gates, entanglement,initial_hamiltonian,target_hamiltonian,initial_state=None):
        self.number_of_qubits = number_of_qubits
        self.initial_state=initial_state
        self.steps = steps
        self.string_initial_hamiltonian=initial_hamiltonian
        self.initial_hamiltonian=hamiltonian_methods['initial'][initial_hamiltonian]['generate'](molecule,taper,freezecore)
        self.string_final_hamiltonian=target_hamiltonian
        self.offset=0
        self.layers = layers
        self.single_qubit_gates = single_qubit_gates
        self.entanglement_gates = entanglement_gates
        self.entanglement = entanglement
        if self.string_initial_hamiltonian == 'transverse':
        X_tuples = []
        for i in range(number_of_qubits):
        X_tuples.append(('X', [i], -1))
        self.initial_hamiltonian = SparsePauliOp.from_sparse_list([*X_tuples], num_qubits = number_of_qubits)
        elif  self.string_initial_hamiltonian == 'paper':
        self.initial_parameters=[0 for x in range(self.number_of_qubits*(self.layers+1))]
        self.initial_parameters[6]=np.pi
        self.initial_parameters[7]=np.pi
        else:
        self.initial_parameters=[0 for x in range(self.number_of_qubits*(self.layers+1))]
        self.initial_parameters[8]=np.pi
        self.initial_parameters[12]=np.pi
        self.target_hamiltonian=hamiltonian_methods['final'][target_hamiltonian]['generate'](molecule, taper, freezecore)
        print(self.initial_hamiltonian)
        self.qcir = TwoLocal(self.number_of_qubits, self.single_qubit_gates, self.entanglement_gates, self.entanglement, self.layers,initial_state= self.initial_state)
        self.number_of_parameters = len(self.initial_parameters)

        def get_expectation_value(self, angles, observable):
        estimator = StatevectorEstimator()
        pub = (self.qcir, observable, angles)
        job = estimator.run([pub])
        result = job.result()[0]
        expectation_value = result.data.evs
        return np.real(expectation_value)

        def alternative_run(self):
        lambdas = [i for i in np.linspace(0, 1, self.steps+1)][1:]
        optimal_thetas = self.initial_parameters.copy()
        instantaneous_expectation_value=self.get_expectation_value(optimal_thetas,self.initial_hamiltonian)
        initial_ground_state=self.minimum_eigenvalue(self.initial_hamiltonian)
        energies_aavqe = [instantaneous_expectation_value]
        energies_exact = [initial_ground_state]
        print(f'We start with the optimal angles of the initial hamiltonian: {optimal_thetas}')
        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
```

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,freezecore)
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, freezecore)
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)
```

```
        initial_ground_state=self.minimum_eigenvalue(self.initial_hamiltonian)
        energies_aavqe = [instantaneous_expectation_value]
        energies_exact = [initial_ground_state]
        print(f'We start with the optimal angles of the initial hamiltonian: {optimal_thetas}')
        for lamda in lambdas:
        print('\n')
        hamiltonian = self.get_instantaneous_hamiltonian(lamda)
        minimization_object = optimize.minimize(self.get_expectation_value, x0=optimal_thetas, args=(hamiltonian), method='SLSQP')
        optimal_thetas = minimization_object.x
        print(f'We are working on {lamda} where the current optimal point is {optimal_thetas}')
        self.offset=0
        inst_exp_value = self.get_expectation_value(optimal_thetas, hamiltonian) - lamda*self.offset
        energies_aavqe.append(inst_exp_value)
        energies_exact.append(self.minimum_eigenvalue(hamiltonian) - lamda*self.offset)
        print(f'and the instantaneous expectation values is {inst_exp_value}')
        print(f'and the true expectation value is {self.minimum_eigenvalue(hamiltonian) - lamda*self.offset}')
        plt.plot(energies_aavqe,label='aavqe energy')
        plt.plot(energies_exact,label='true energy')
        plt.legend()
        plt.xlabel('time')
        plt.ylabel('energy (Ha)')
        plt.title(f'{self.string_initial_hamiltonian} and {self.string_final_hamiltonian}')
        plt.show()
        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 $|00010001\rangle$.

```
        def get_hartreefock_in_projector(self):
        # Get the number of spatial orbitals (i.e., the number of qubits in the mapping)
        problem= self.electronic_structure_problem
        # Get the Hartree-Fock state
        hf_state = HartreeFock(problem.num_spatial_orbitals,problem.num_particles,JordanWignerMapper())
        state_vector= Statevector(hf_state)
        projector=state_vector.to_operator()

        return projector
```