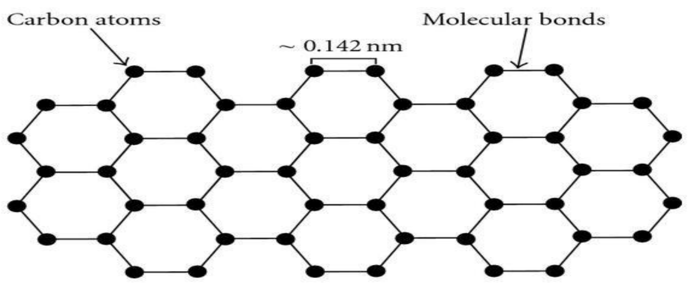
Task is to use the VQE algorithm to compute the ground state of a Heisenberg spin-1/2 system on a graphene lattice.



Heisenberg spin-1/2 model on the Graphene lattice :

$$H = \sum_{<ij>} X_i X_j + Y_i Y_j + Z_i Z_j$$

X, Y, Z are Pauli operators. Each spin will be represented by a single qubit.

A parametrized ansatz will be prepared and the quantum computer (simulator will be used) will compute the energy expectation value of Hamiltonian H acting on the parameterized ansatz for a given set of parameter values. A classical computer uses the measurement data from the quantum computer to determine how the parameters values should be adjusted to further minimize the energy. As the classical and the quantum computer loop through many iterations, they search the parameter space and converge to the approximate ground state.

The minimization of expectation value of Hamiltonian depends on choice of initial ansatz.

First, the graphene lattice is defined on a graph (24 qubits total) :

We will also need to specify how the spin-1/2 particles map to qubits. Using Qiskit nature's `LogarithmicMapper`, we can map each spin-1/2 site to a single qubit.

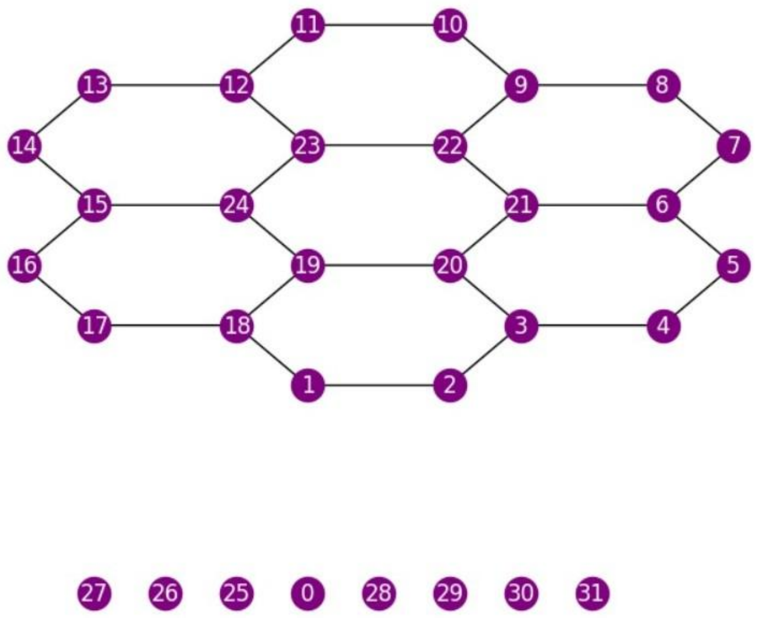
To execute VQE on real devices, it is necessary to "inflate" the Hamilontian from 24 qubits to ‘n’ qubits to match the device’s no of qubits, but any extra qubits will not participate in the energy expectation value.

48]:

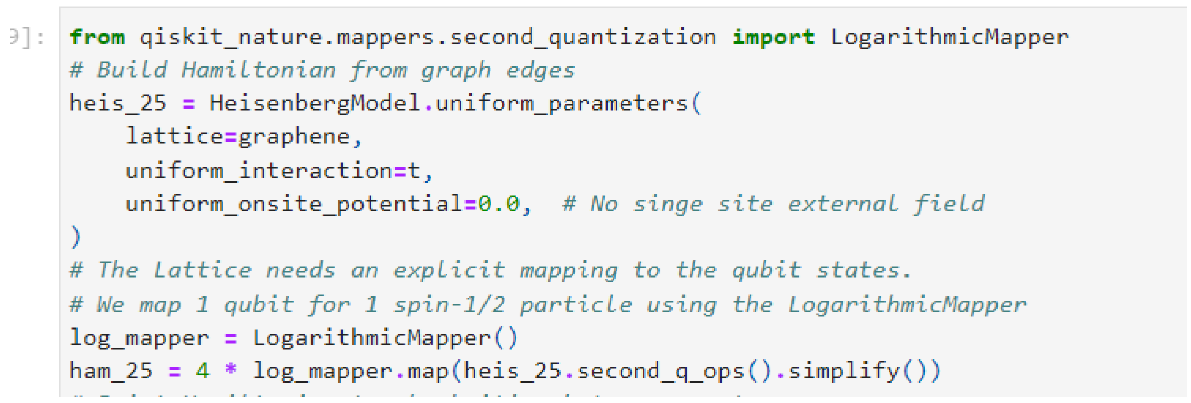
```python
import rustworkx as rx
from qiskit_nature.problems.second_quantization.lattice import Lattice
num_sites = 32
t = 1.0

graph = rx.PyGraph(multigraph=False)
graph.add_nodes_from(range(num_sites))
edge_list = [
    (1, 2, t),
    (2, 3, t),
    (3, 4, t),
    (4, 5, t),
    (5, 6, t),
    (6, 7, t),
    (7, 8, t),
    (8, 9, t),
    (9, 10, t),
    (10, 11, t),
    (11, 12, t),
    (12, 13, t),
    (13, 14, t),
    (14, 15, t),
    (15, 16, t),
    (16, 17, t),
    (17, 18, t),
    (1, 18, t),
    (18, 19, t),
    (19, 20, t),
    (20, 21, t),
    (21, 22, t),
    (22, 23, t),
    (23, 24, t),
    (24, 19, t),
    (20, 3, t),
    (21, 6, t),
    (22, 9, t),
    (23, 12, t),
    (24, 15, t),
]
```

```python
b=0.5
s=np.sqrt(0.75)

graphene_pos = {0:[0,-3], 1:[0,0], 2:[1,0], 3:[1+b,s], 4:[2+b,s], 5:[2+2*b,2*s], 6:[2+b,3*s],
              7:[2+2*b,4*s], 8:[2+b,5*s], 9:[1+b,5*s], 21:[1+b,3*s], 10:[1,6*s],
              22:[1,4*s], 20:[1,2*s], 11:[0,6*s], 23:[0,4*s], 19:[0,2*s], 18:[-b,s],
              24:[-b,3*s], 12:[-b,5*s], 13:[-(b+1),5*s], 15:[-(b+1),3*s], 17:[-(b+1),s],
              14:[-(2*b+1),4*s], 16:[-(2*b+1),2*s], 25:[-b,-3], 26:[-2*b,-3], 27:[-3*b,-3], 28:[b,-3], 29:[2*b,-3], 30:[3*b,-3], 31:[4*b,-3],}

"""
kagome_pos = {0:[0,-3], 1:[0,0], 2:[1,0], 3:[1+b,s], 4:[2+b,s], 5:[2+2*b,2*s], 6:[2+b,3*s],
              7:[2+2*b,4*s], 8:[2+b,5*s], 9:[1+b,5*s], 10:[1,6*s], 11:[0,6*s],18:[-b,s],
              12:[-b,5*s], 13:[-(b+1),5*s], 15:[-(b+1),3*s], 17:[-(b+1),s],
              14:[-(2*b+1),4*s], 16:[-(2*b+1),2*s]}
"""

# Generate graph from the list of edges
graph.add_edges_from(edge_list)

# Make a Lattice from graph
graphene = Lattice(graph)

graphene.draw(style={'with_labels':True, 'font_color':'white', 'node_color':'purple', 'pos':graphene_pos})
plt.show()
```

Mapped qubit lattice (ibmq_qasm_simulator (32 qubits)) :

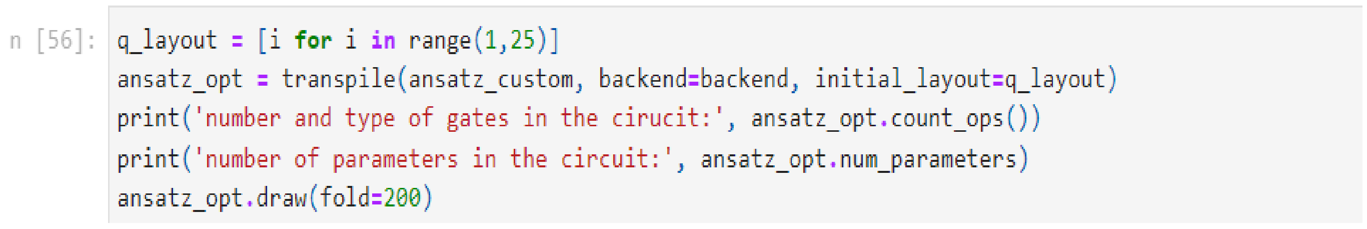


The Hamiltonian consisting of Pauli gates are created by the following (The HeisenbergModel is defined in the same way IsingModel is built.)


```
from qiskit_nature.mappers.second_quantization import LogarithmicMapper
# Build Hamiltonian from graph edges
heis_25 = HeisenbergModel.uniform_parameters(
    lattice=graphene,
    uniform_interaction=t,
    uniform_onsite_potential=0.0,  # No singe site external field
)
# The Lattice needs an explicit mapping to the qubit states.
# We map 1 qubit for 1 spin-1/2 particle using the LogarithmicMapper
log_mapper = LogarithmicMapper()
ham_25 = 4 * log_mapper.map(heis_25.second_q_ops().simplify())
```

Define a ansatz and matching qubit layout:

The ansatz can be defined over 24 qubits without needing to inflate it to 32 qubits. The transpiler will take care of that.


```
q_layout = [i for i in range(1,25)]
ansatz_opt = transpile(ansatz_custom, backend=backend, initial_layout=q_layout)
print('number and type of gates in the cirucit:', ansatz_opt.count_ops())
print('number of parameters in the circuit:', ansatz_opt.num_parameters)
ansatz_opt.draw(fold=200)
```

ansatz_custom : defined ansatz over 24 qubits ; ansatz_opt : inflated ansatz over 32 qubits ; backend : ibmq_qasm_simulator

I will use both pre-built `EfficientSU2()` to generate an ansatz and a custom built ansatz to compare ground state energies.

Choose a classical optimizer :

Along with choosing a classical optimizer, picking an initial set of parameters plays a significant role in VQE.
I have taken the SPSA optimizer.

It's convenient to bring the components of VQE together into a custom class which has been called `CustomVQE`.
Let's define a callback function which tells us what parameter values is the classical optimizer picking? How does the convergence to the ground state energy behave?

```
]: # Define a simple callback function
   intermediate_info_sim_backend = []
   def callback_sim(value):
           intermediate_info_sim_backend.append(value)
```

I have taken this from IBM Quantum's code samples for custom vqe function creation :
Basically the code below defines a function CustomVQE(estimator, circuit, optimizer, callback), which returns minimum eigenvalue on the command of :
CustomVQE(estimator, ansatz_opt, optimizer, callback=callback).compute_minimum_eigenvalue(Hamiltonian)

```
optimizer = SPSA(maxiter=75)
```

```
[58]: from qiskit.algorithms import MinimumEigensolver, VQEResult

      # Define a custome VQE class to orchestra the ansatz, classical optimizers,
      # initial point, callback, and final result
      class CustomVQE(MinimumEigensolver):

          def __init__(self, estimator, circuit, optimizer, callback=None):
              self._estimator = estimator
              self._circuit = circuit
              self._optimizer = optimizer
              self._callback = callback

          def compute_minimum_eigenvalue(self, operators, aux_operators=None):

              # Define objective function to classically minimize over
              def objective(x):
                  # Execute job with estimator primitive
                  job = self._estimator.run([self._circuit], [operators], [x])
                  # Get results from jobs
                  est_result = job.result()
                  # Get the measured energy value
                  value = est_result.values[0]
                  # Save result information using callback function
                  if self._callback is not None:
                      self._callback(value)
                  return value

              # Select an initial point for the ansatzs' parameters
              x0 = np.pi/4 * np.random.rand(self._circuit.num_parameters)

              # Run optimization
              res = self._optimizer.minimize(objective, x0=x0)

              # Populate VQE result
              result = VQEResult()
              result.cost_function_evals = res.nfev
              result.eigenvalue = res.fun
              result.optimal_parameters = res.x
              return result
```

Since the program is run on IBM Quantum cloud platform it must be wrapped in session, after that the below code will run the VQE and give us the computed minimum eigenvalue and the total time taken to compute it in seconds.

```
[64]: # Setup Estimator with session error handling reconnection work around
      start = time.time()
      with Session(service=service, backend=backend) as session:
          # Prepare extended primitive
          rt_estimator = RetryEstimator(session=session)
          # set up algorithm
          custom_vqe = CustomVQE(rt_estimator, ansatz_opt, optimizer, callback=callback_sim)
          # run algorithm
          result = custom_vqe.compute_minimum_eigenvalue(ham_25)
      end = time.time()
      print(f'execution time (s): {end - start:.2f}')
```

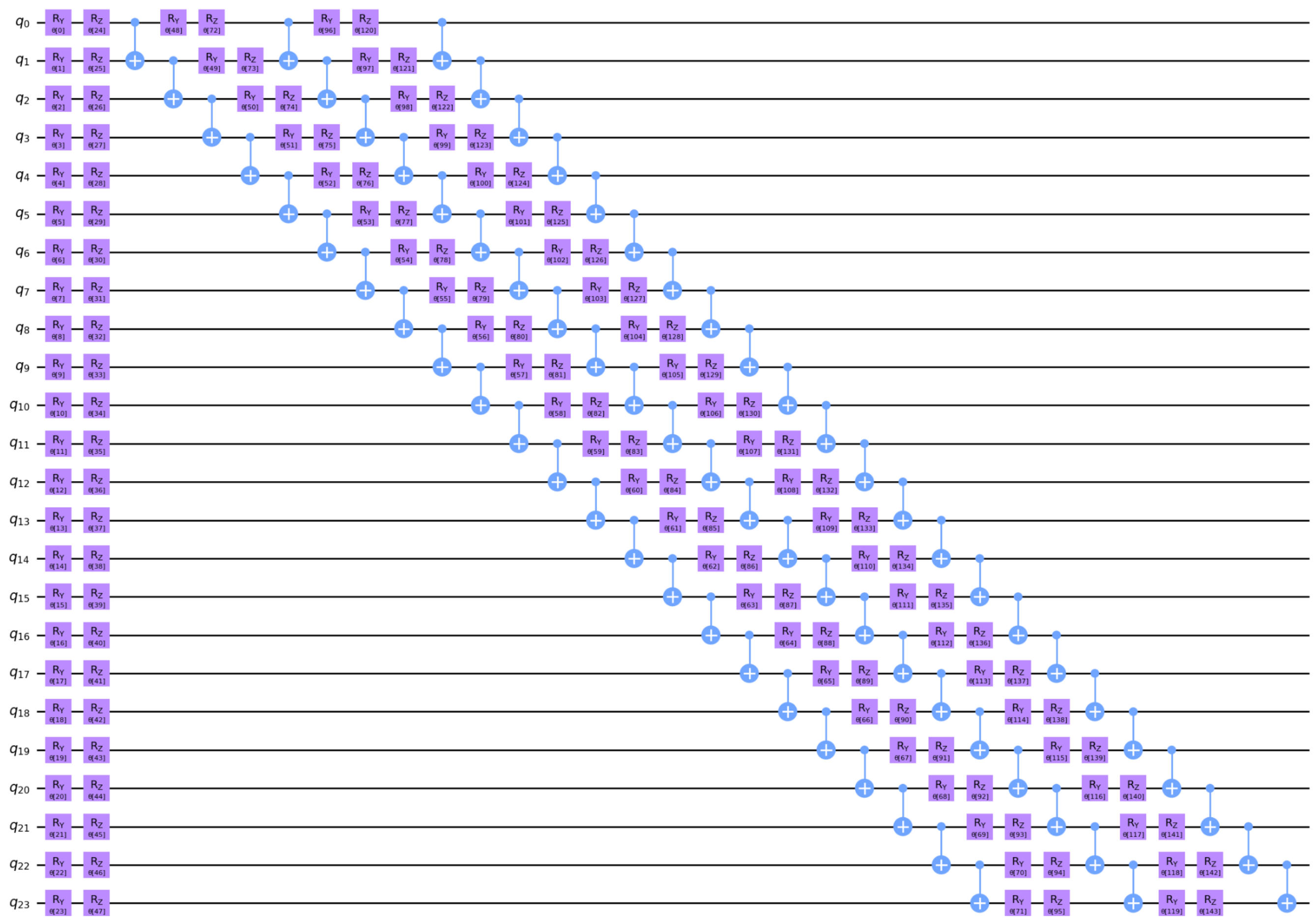
```
[65]: plt.plot(intermediate_info_sim_backend, color='purple', lw=2, label='Simulated VQE')
      plt.ylabel('Energy')
      plt.xlabel('Iterations')

      plt.legend()
      plt.grid()
      plt.show()
```

Now I will show the ansatz and the minimum energy with graph

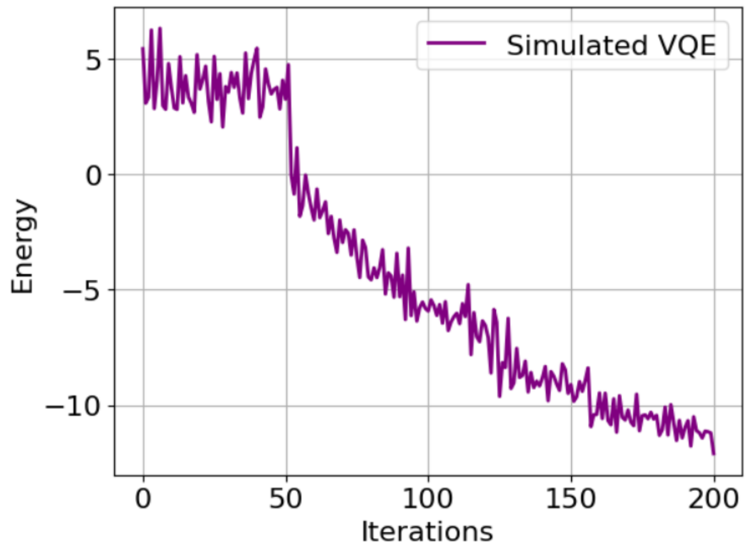
1)

```
[ ]: from qiskit.circuit.library import EfficientSU2
```

```
[ ]: # Construct ansatz from qiskit circuit library functions
     ansatz = EfficientSU2(24, entanglement='linear', reps=3, skip_final_rotation_layer=True).decompose()
     ansatz.draw(fold=300)
```



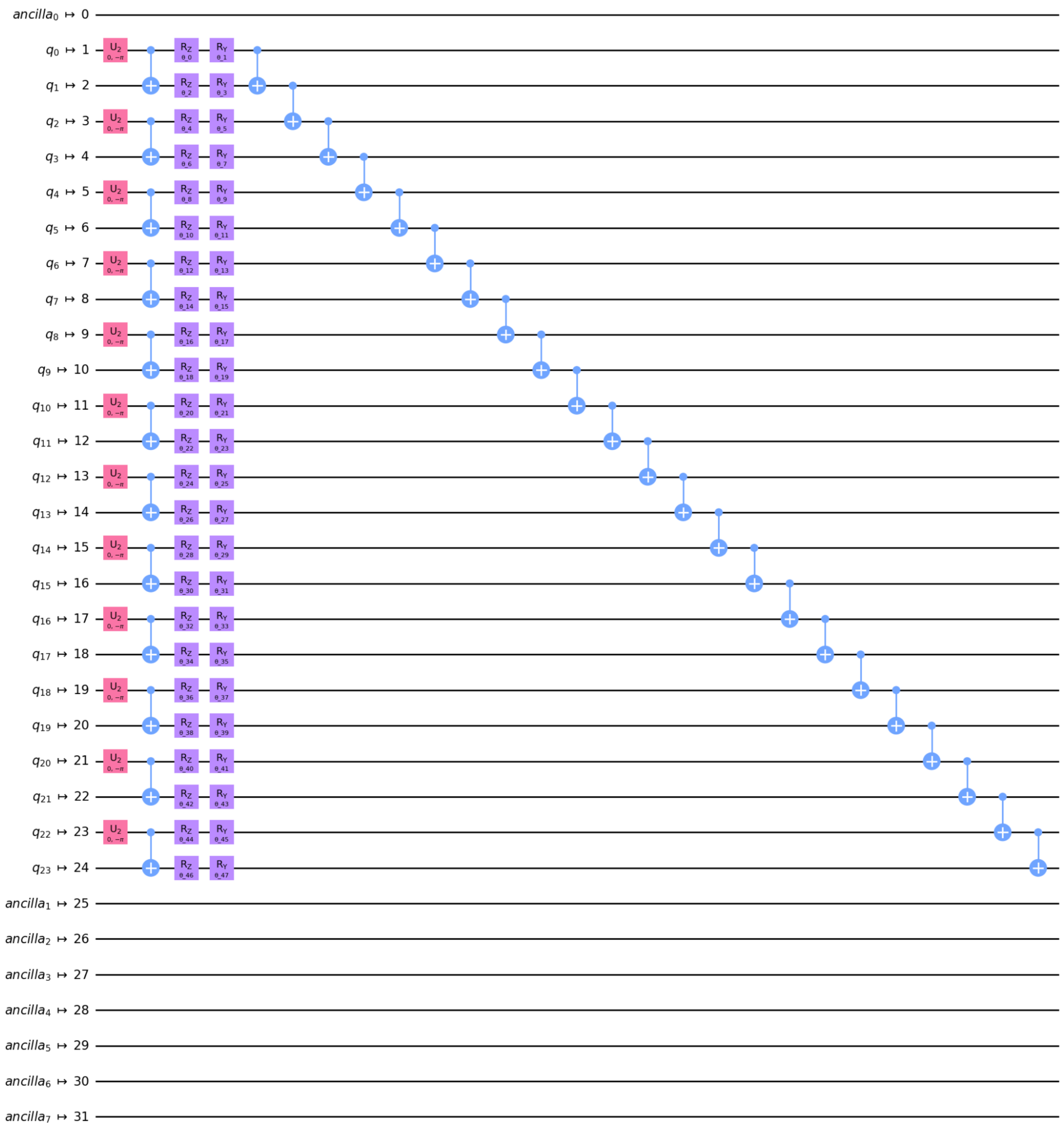
execution time (s): 3111.92
Computed ground state energy: -12.1080000000



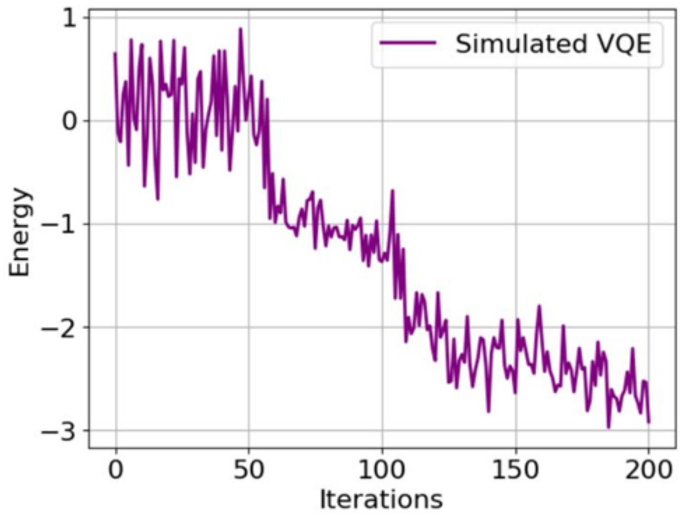
2)

]:
```python
from qiskit import QuantumCircuit, transpile
from qiskit.circuit import Parameter

from qiskit import IBMQ
```

]:
```python
# Build a custom ansatz from scratch
ansatz_custom = QuantumCircuit(24)
# build initial state
ansatz_custom.h(range(0, 24, 2))
ansatz_custom.cx(range(0, 23, 2), range(1, 24, 2))
# First layer
j = 0
for i in range(24):
    ansatz_custom.rz(Parameter('θ_' + str(j)), i)
    j += 1
    ansatz_custom.ry(Parameter('θ_' + str(j)), i)
    j += 1
ansatz_custom.cx(range(0, 23), range(1, 24))

ansatz_custom.draw(fold=250)
```



```
execution time (s): 1898.06
Computed ground state energy: -2.9150000000
```



While trying to compute eigenvalue using NumPyEigensolver for using it as a reference line, each time the kernel died within 2hrs, so I could not compute the eigenvalue.