Yasir_Mansour_HW3_VQE

June 15, 2024

1 H Molecule Homework Assignment

1.0.1 Quantum Software Development Journey: From Theory to Application with Classiq - Part 3

- Similarly to what we have done in class, in this exercise we will implement the VQE on H2 molecule.
- This time instead of using the built-in methods and functions (such as `Molecule` and `MoleculeProblem`) to difne and solve the problem, you will be provided with a two qubits Hamiltonian.

1.1 Submission

- Submit the completed Jupyter notebook and report via GitHub. Ensure all files are correctly named and organized.
- Use the Typeform link provided in the submission folder to confirm your submission.

1.2 Additional Resources

- Classiq Documentation(docs.classiq.io/latest/)
- The notebook from live session #3

1.3 Important Dates

- **Assignment Release:** 22.5.2024
- **Submission Deadline:** 3.6.2024 (7 A.M GMT+3)

Happy coding and good luck!

1.3.1 Part 1

Given the following Hamiltonian:

$$\hat{H} = -1.0523 \cdot (I \otimes I) + 0.3979 \cdot (I \otimes Z) - 0.3979 \cdot (Z \otimes I) - 0.0112 \cdot (Z \otimes Z) + 0.1809 \cdot (X \otimes X)$$

Complete the following code

[ ]:
```
!pip install classiq
```

[1]:
```
import classiq
```

[ ]:
```
classiq.authenticate()
```

[2]:
```
from typing import List, cast
from classiq import *
from classiq import Pauli, PauliTerm

#TODO: Complete Hamiltonian
HAMILTONIAN = QConstant("HAMILTONIAN", List[PauliTerm], [
    PauliTerm([Pauli.I, Pauli.I], -1.0523),
    PauliTerm([Pauli.I, Pauli.Z], 0.3979),
    PauliTerm([Pauli.Z, Pauli.I], -0.3979),
    PauliTerm([Pauli.Z, Pauli.Z], -0.0112),
    PauliTerm([Pauli.X, Pauli.X], 0.1809),
])
```

[3]:
```
@qfunc
def main(q: Output[QArray[QBit]], angles: CArray[CReal, 3]) -> None:
    # TODO: Create an ansatz which allows each qubit to have
    # arbitrary rotation

    allocate(2, q)
    U(angles[0], angles[1], angles[2], 0, q[0])
    U(angles[0], angles[1], angles[2], 0, q[1])
    #CX(q[0], q[1])

@cfunc
def cmain() -> None:
    res = vqe(
        hamiltonian=HAMILTONIAN,
        maximize=False,
        initial_point=[],
        optimizer=Optimizer.COBYLA,
        max_iteration=1000,
        tolerance=0.001,
        step_size=0,
        skip_compute_variance=False,
        alpha_cvar=1.0,
    )
    save({"result": res})

qmod = create_model(main, classical_execution_function=cmain)
#TODO: complete the line, use classical_execution_function
qprog = synthesize(qmod)
```

```
# show(qprog)
```

[5]:
```
execution = execute(qprog)
res = execution.result()
# execution.open_in_ide()
vqe_result = res[0].value
 #TODO: complete the line
```

[6]:
```
print(f"Optimal energy: {vqe_result.energy}")
print(f"Optimal parameters: {vqe_result.optimal_parameters}")
print(f"Eigenstate: {vqe_result.eigenstate}")
```

```
Optimal energy: -1.0681201171874999
Optimal parameters: {'angles_0': -3.429491869883728, 'angles_1':
-4.962285285210174, 'angles_2': 6.007229196693071}
Eigenstate: {'10': (0.13621559198564606+0j), '01': (0.13621559198564606+0j),
'11': (0.9812699042567239+0j)}
```

Optimal energy: -1.0711231445312501 Optimal parameters: {'angles_0': -3.0914206855935538, 'angles_1': -0.23729943557563232, 'angles_2': -2.5756826635214636} Eigenstate: {'01': (0.02209708691207961+0j), '11': (0.9997558295653994+0j)}

Does it similar to the `optimal energy` we calculated in class?
Does it similar to the `total energy` we calculated in class?

1.3.2 Part 2

Now, we want to have a more interesting ansatz in our `main`.
Add **one** line of code to the `main` function you created in Part 1 that creates **entanglement** between the two qubits.
Which gate should you use?

[7]:
```
@qfunc
def main(q: Output[QArray[QBit]], angles: CArray[CReal, 3]) -> None:
    # TODO: Create an ansatz which allows each qubit to have
    # arbitrary rotation

    allocate(2, q)
    U(angles[0], angles[1], angles[2], 0, q[0])
    U(angles[0], angles[1], angles[2], 0, q[1])
    CX(q[0], q[1])
    #H(q[0])
    #X(q[1])
    #CX(q[0], q[1])


@cfunc
def cmain() -> None:
    res = vqe(
```

```
        HAMILTONIAN, # TODO: complete the missing argument
        False,
        [],
        optimizer=Optimizer.COBYLA,
        max_iteration=1000,
        tolerance=0.001,
        step_size=0,
        skip_compute_variance=False,
        alpha_cvar=1.0,
    )
    save({"result": res})

qmod = create_model(main, classical_execution_function=cmain)
#TODO: complete the line, use classical_execution_function
qprog = synthesize(qmod)
# show(qprog)
```

[8]:
```
execution = execute(qprog)
res = execution.result()
# execution.open_in_ide()
vqe_result = res[0].value
#TODO: complete the line
```

[9]:
```
print(f"Optimal energy: {vqe_result.energy}")
print(f"Optimal parameters: {vqe_result.optimal_parameters}")
print(f"Eigenstate: {vqe_result.eigenstate}")
```

```
Optimal energy: -1.83589755859375
Optimal parameters: {'angles_0': -3.1968121237616858, 'angles_1':
4.415682206108902, 'angles_2': 5.319096413899962}
Eigenstate: {'11': (0.02209708691207961+0j), '10': (0.038273277230987154+0j),
'01': (0.9990229601966113+0j)}
```

Optimal energy: -1.8452896484374999 Optimal parameters: {‘angles_0’: -2.9812026284028255, ‘angles_1’: 0.8040137892002661, ‘angles_2’: 5.77426479151465}

Eigenstate: {‘11’: (0.08267972847076846+0j), ‘10’: (0.07967217989988726+0j), ‘01’: (0.9933863328282708+0j)}

Does it similar to the `optimal energy` we calculated in class?
Does it similar to the `total energy` we calculated in class?
What can we learn about the provided form this result Hamiltonian?

With entanglement one gets better results.