Quantum Simulation of Kitaev Model on Different Lattices with HVA

Zhanpeng Fu, Zhongling Lu, Hantian Zhu

Zhiyuan College, Shanghai Jiaotong University, Shanghai, China

(16 January 2022)

Variational quantum eigensolver (VQE) is a applicable way to calculate energy eigenvalues when using a quantum computer with noise. Hence, we use VQE to simulate the quantum properties of quantum spin liquid. We choose the Kitaev model using Hamiltonian variational ansatz (HVA) for simulation.

1 Introduction

In the context of quantum computation there is reason to believe the quantum simulation of spin systems may offer early results in the search for beyond classical computations of real scientific interest. Quantum spin liquid is such a kind of novel matter in condensed matter physics that can be simulated using quantum algorithms. It was theoretically predicted due to the existence of quantum frustration. Raised by A.Kitaev[1], the Kitaev model is one of the toy models that could bring quantum spin liquid state into emergence and it may be used in establishing the system of topological quantum computing. Nonetheless, the quantum spin liquid state was recently deliberately designed experimentally and have been observed on a programmable quantum simulator[2], suggesting the new hope of topological quantum computing and the strong power of quantum simulation.

There are many different ways to simulate the Kitaev model, one of which is the commonly used variational quantum eigensolver (VQE) algorithm. To meet the demand of different tasks, various ansatzs are developed and utilized in different scenarios. One particular ansatz of interest is the Hamiltonian variational ansatz (HVA). This ansatz was inspired by the great ideas of quantum approximate optimization algorithm (QAOA) and adiabatic quantum computation. Previous work has suggested the HVA may be more robust and therefore easier to optimize[3].

In this work, we focus on the Kitaev model on two different lattices: honeycomb lattice and square-octagon lattice, using VQE algorithm with HVA to simulate and calculate the ground state energy and mean magnetic moment of the Kitaev model on honeycomb lattice and square-octagon-lattice[4]. We calculated the ground states under dozens of configurations and showed the phase transition with respect to magnetic moment.

2 Background

The Kitaev model of coupled spin-1/2s on a honeycomb lattice is exactly solvable, which represents a prototypical QSL example of quantum spin liquids[2]. However, the problem can't be analytically achieved considering the presence of a local magnetic field. The

field-free Kitaev honeycomb Hamiltonian is

$$\mathcal{H}_0 = J_x \sum_{(i,j)\in x} X_i X_j + J_y \sum_{(i,j)\in y} Y_i Y_j + J_z \sum_{(i,j)\in z} Z_i Z_j \quad (1)$$

Where $J_{x,y,z} < 0$ is the coefficient of spin interaction corresponding to antiferromagnetism, X_i, Y_i, Z_i respectively represent the Pauli matrix $\sigma_x, \sigma_y, \sigma_z$ on the i-th lattice point, and the nearest—neighbor bonds are divided into three categories due to the high degree of anisotropy, namely set x, y, z is showned in 1, corresponding to the spin coupling in this direction.

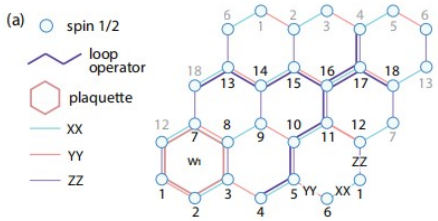


Figure 1: Spin coupling in honeycomb lattice[2]

In order to study its phase transition process under the external field, the energy term of the external field is added to the original Hamiltonian

$$\mathcal{H} = \mathcal{H}_0 + \sum_i (h_x X_i + h_y Y_i + h_z Z_i) \quad (2)$$

The Kitaev model on the square-octagon lattice was first studied by Yang et al. using Kitaev' s original approach.The antiferromagnetic Kitaev model in the Squre-octagon-lattice has the same form of Hamiltonian with which in honeycomb lattice.

The set x, y, z is showned in 2, corresponding to the spin coupling in this direction.

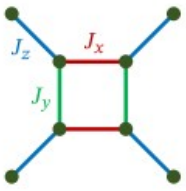


Figure 2: Anisotropic spin coupling in square-octagon lattice[4]

We consider the case where the coupling coefficients are isotropic, i.e. $J_x = J_y = J_z$. As shown in 3(a) and 3(b), when the external field along the z direction increases, the spin regime will continuously transition from the KSL state to an trival polarization phase polarized by the z-direction magnetic field.

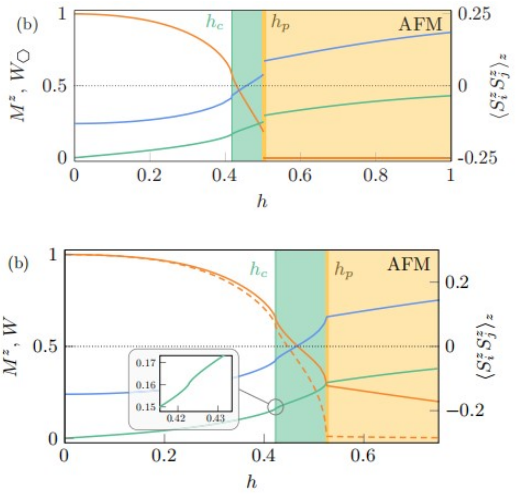


Figure 3: Previous results for the Kitaev model on the (up) honeycomb lattice and (down) square-octagon lattice under a magnetic field in [001] direction for AFM couplings[6]

It can also be seen from the phase diagram that there is an intermediate phase in the spin system in the field strength range of approximately 0.1 between h_c and h_p in both lattices, but due to our accuracy limitations, it is impossible to scan the entire interval with a small field strength range, so our results will only show the phase transition between the KSL and the polarized phase.

3 Methodology

HVA utilizes the ideas of both QAOA and quantum annealing algorithms. In quantum annealing we suppose it is easy to prepare a ground state ψ_I of H_0. Then, assuming that no gap closes, it is possible to adiabatically evolve from ψ_I to the ground state of H_1. If we break the annealing into short time steps dt and evolve for a total time T , this annealing is a sequence of (T/dt) different unitary rotations by Hamiltonian in-

terpolating between H_0 and H_1. This could be implemented on a quantum computer using a Trotter-Suzuki method which further decomposes this sequence into a sequence of unitary rotations by individual terms in the Hamiltonian.

Now consider an HVA with L layers, suppose that the whole unitary matrix corresponding to HVA is $U(\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ is the optimization parameters. Then

$$U(\boldsymbol{\theta}) = U(\boldsymbol{\theta_L})U(\boldsymbol{\theta_{L-1}})\cdots U(\boldsymbol{\theta_1}) \tag{3}$$

Each layer has six optimization parameters, which correspond to the time term in the time evolution operator corresponding to each part of Hamiltonian, which is very similar to QAOA. Which means

$$\begin{aligned} U(\boldsymbol{\theta_n}) = & e^{-i\theta_{n,6}\sum_j Z_j} e^{-i\theta_{n,5}\sum_{(i,j)\in z} Z_i Z_j} \\ & \times e^{-i\theta_{n,4}\sum_j Y_j} e^{-i\theta_{n,3}\sum_{(i,j)\in y} Y_i Y_j} \\ & \times e^{-i\theta_{n,2}\sum_j X_j} e^{-i\theta_{n,1}\sum_{(i,j)\in x} X_i X_j} \end{aligned} \tag{4}$$

The terms as X_i, Y_i, Z_i can be achieved by the single gates $Rx(2\theta_{n,2}), Ry(2\theta_{n,4}), Rz(2\theta_{n,6})$ acting on the i-th qubit. The term $e^{-i\theta_{n,1}X_iX_j}$ can be achieved by

$$(H\otimes H)CNOT_{i,j}(I\otimes Rz(2\theta_{n,1}))CNOT_{i,j}(H\otimes H) \tag{5}$$

The term $e^{-i\theta_{n,3}Y_iY_j}$ can be achieved by

$$\begin{aligned} & (Rx(\frac{\pi}{2})\otimes Rx(\frac{\pi}{2}))CNOT_{i,j}(I\otimes Rz(2\theta_{n,3})) \\ & CNOT_{i,j}(Rx(\frac{\pi}{2})\otimes Rx(\frac{\pi}{2})) \end{aligned} \tag{6}$$

The term $e^{-i\theta_{n,5}Z_iZ_j}$ can be achieved by

$$CNOT_{i,j}(I\otimes Rz(2\theta_{n,5}))CNOT_{i,j} \tag{7}$$

Where the $CNOT_{i,j}$ represents a CNOT gate with qubit i as target qubit and qubit j as control qubit. The main body of the HVA can be constructed by this way. The idea of the measurement and optimization part is basically the same as that of classical VQE. BOBYQA is chosen as classical optimizer to speed up the convergence.

In order to speed up the convergence speed, we also add additional optimization parameters before the main body of HVA. These optimization parameters are selected at the beginning to make the system energy as low as possible, so that the convergence result is better. For example, when the external field is small, we can rotate the spins with $\frac{\pi}{2}$ around the x-axis to the xy plane, and then rotate the odd-numbered spins in the set x around the z-axis by $\frac{\pi}{4}$, and the even-numbered spins around the z-axis by $-\frac{3\pi}{4}$, so that the nearest neighbor spins in the set x, y are opposite. When the external field is small, we can rotate the spin around the x-axis by a certain angle, such as $\frac{\pi}{4}$, and the same thereafter.

4 Results

4.1 Kitaev honeycomb lattice

We use 12 qubits and three layers to describe the honeycomb Kitaev model. The grid point labeling and lattice periodic structure are shown in the fig4.

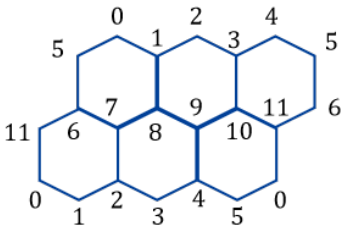


Figure 4: The grid point labeling and lattice periodic structure

Select the energy $E = \langle\psi|\,\mathcal{H}\,|\psi\rangle$ as the optimization function and output the average magnetic moment $M_z = \frac{\langle\psi|\sigma_z^{\otimes N}|\psi\rangle}{N}$ of the system in the z direction at the same time. The sequence of quantum gates constructed according to HVA and the additional optimization parameters mentioned above is shown in fig5.

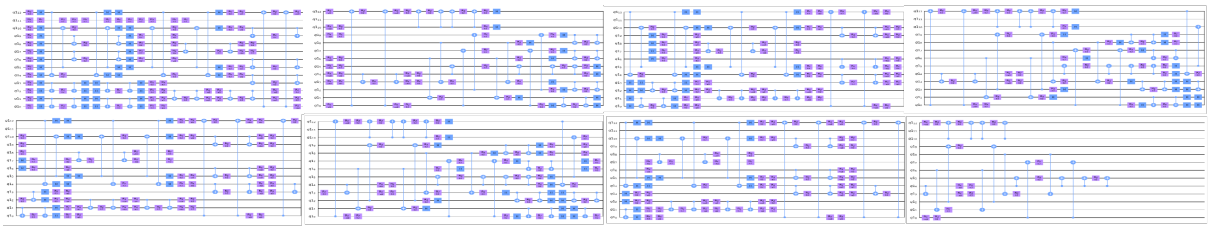
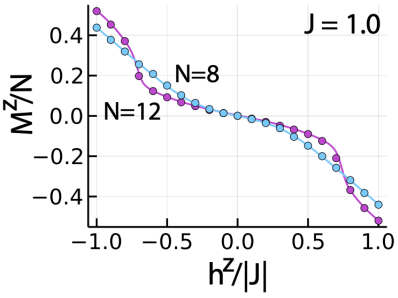
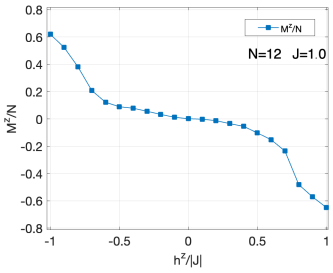


Figure 5: Quantum gate sequence corresponding to honeycomb lattice simulation

Take the coupling coefficient satisfying$J_x = J_y = J_z = -1$, scan the external field in the z direction from -1 to 1 with a step size of 0.1, and obtain the average magnetic moment in the z direction in the system ground state. We obtain the result as shown in the fig6(b).



(a) Previous work



(b) Ours

Figure 6: Comparison of mean magnetic moment with previous work[5]

The results we obtained were very similar to those given by other quantum simulation methods in the cited article. When the external field in the z direction is about +-0.5, the change trend of the magnetic moment of the system has a significant change, corresponding to the system transition from the KSL phase to the trivial spin-polarized phase. It is also basically in the transition magnetic field interval (h_c, h_p) obtained by the mean field method in the fig3(a).We choose a specific example to research the convergence condition of the iteration.The relation between energy and time of iteration in honeycomb simulation can be described by the figure below.We can see that the energy is convergent.After carrying on the iteration for 60 times the energy tends to be stable.

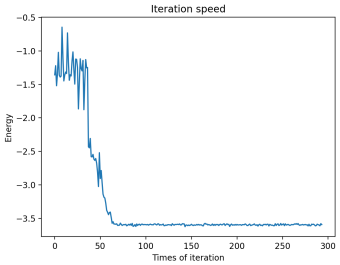
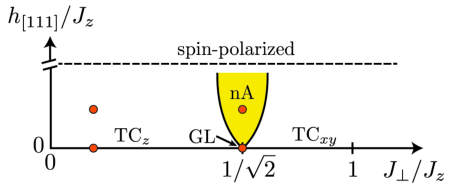


Figure 7: honeycomb iteration speed

4.2 Square-octagon lattice

In the context of square-octagon lattice, we tested the HVA algorithm under 4 given configurations in paper[4](see8) and calculated the energy of each ground state, showing high accuracy except for the 4th configuration($GL_z + h$).



Label	J_x	J_y	J_z	h_x	h_y	h_z	$E_g(N=8)$	$E_g(N=16)$
TC$_z$	0.1	0.1	1	0	0	0	-4.0100	-8.0250
TC$_z$+h	0.1	0.1	1	$\frac{0.05}{\sqrt{3}}$	$\frac{0.05}{\sqrt{3}}$	$\frac{0.05}{\sqrt{3}}$	-4.2476	-8.5002
GL	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$	1	0	0	0	-4.4721	-9.3002
GL+h	$\frac{1}{\sqrt{2}}$	$\frac{1}{\sqrt{2}}$	1	$\frac{0.05}{\sqrt{3}}$	$\frac{0.05}{\sqrt{3}}$	$\frac{0.05}{\sqrt{3}}$	-4.7011	-9.7008

Figure 8: Four ground state under different configurations

	Exact value	Calculated	Z moments
TC_z	-4.01	**-4.00936**	0.998
$TC_z + h$	-4.2476	**-4.2425**	-0.9987
GL_z	-4.4721	**-4.4614**	0.959
$GL_z + h$	-4.7011	**-4.3675**	-0.2923
Polarized state	NULL	**-32.0478**	1

Table 1: Our results on ground states above

Considering the periodic lattice, we simulated the square-octagon lattice with 8 qubits. The grid point labeling and lattice periodic structure are shown in fig 10. The sequence of quantum gates constructed according to HVA and the additional optimization parameters mentioned above is shown in fig9.

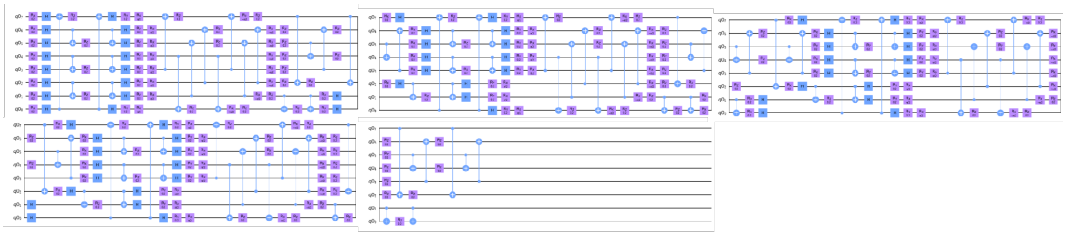


Figure 9: Quantum gate sequence corresponding to honeycomb lattice simulation

Take the coupling coefficient as $J_x = J_y = J_z = -1$, scan the external field in the z direction from -1 to 1 with a step size of 0.1, and obtain the average magnetic moment in the z direction in the system ground state.

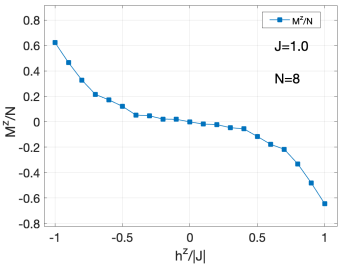


Figure 11: Mean magnetic moment in square-octagon lattice

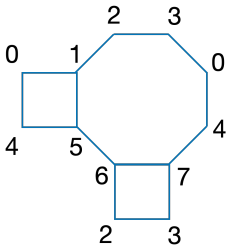


Figure 10: The grid point labeling and lattice periodic structure of square-octagon lattice

Although there is no direct benchmark for this result, it can be found that the turning point of the magnetic moment with the applied magnetic field is around $h_z = 0.5$, which is in the interval of the turning magnetic field (h_c, h_p) calculated by the mean field. It also corresponds to the transition of the system from the KSL phase to the spin-polarized phase.We choose a specific example to research the convergence condition

of the iteration.The relation between energy and time of iteration in octagon simulation can be described by the figure below.We can see that the energy is convergent .After carrying on the iteration for 40 times the energy tends to be stable.

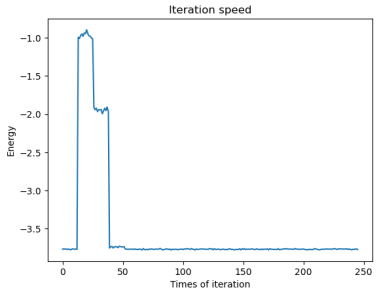


Figure 12: octagon iteration speed

5 Discussion & Conclusion

There is still something more to discuss for our simulation on quantum spin liquid.First,our simulation program is based on python and numpy , so the program runs very slowly.Hence we optimize our program and find a better place for running it such as AWS.Second,the previous edition of our program needs a complex process to set starting parameters.So we add a automatic gate flip code to simplify the setting part.Third,our simulation result is in good agreement with the results of the original work.But we still can not solve the problem that different starting parameters cause different output,for which we can not state the distinct reason for choosing the final starting parameter.Forth,our program concludes fewer layers than the ordinary work.If more layers' program can run substantially better and how better it is is of our consider.Fewer layers' program runs much quicker than others,and if we can set a appropriate series of starting parameters,we can get a perfect result as what we shown above.Our conclusion is that VQE and HVA can well simulate mean magnetic moment in square-octagon lattice,ground state under different configurations for quantum spin liquid.And the Kitaev model suits quantum spin system well.

References

[1] Kitaev, A. (2006). Anyons in an exactly solved model and beyond. Annals of Physics, 321(1), 2-111.

[2] Semeghini, G., Levine, H., Keesling, A., Ebadi, S., Wang, T. T., Bluvstein, D., ... & Lukin, M. D. (2021). Probing Topological Spin Liquids on a Programmable Quantum Simulator. arXiv preprint arXiv:2104.04119.

[3] J. R. McClean, S. Boixo, V. N. Smelyanskiy, R. Babbush, and H. Neven, Nat. Commun. 9, 4812 (2018)

[4] Li, A. C., Alam, M. S., Iadecola, T., Jahin, A., Kurkcuoglu, D. M., Li, R., ... & Tubman, N. M. (2021). Benchmarking variational quantum eigensolvers for the square-octagon-lattice Kitaev model. arXiv preprint arXiv:2108.13375.

[5] Bespalova, T. A., & Kyriienko, O. (2021). Quantum simulation and ground state preparation for the honeycomb Kitaev model. arXiv preprint arXiv:2109.13883.

[6] Christoph Berke, Simon Trebst, and Ciarán Hickey, Field stability of Majorana spin liquids in antiferromagnetic Kitaev models Phys. Rev. B 101, 214442 (2020)

A Appendix

A.1 Code of honeycomb lattice

```
import numpy as np
```

```
from qiskit import *
import pybobyqa

pi = np.pi
runtime_count = 0

def parametric_state(par):
    """
    one layer construct
    :return: Ansatz-target state
    """
    q = QuantumRegister(qubit_number)
    c = ClassicalRegister(qubit_number)
    circuit = QuantumCircuit(q, c)
    number = int((par.size)/6)
    theta = np.arccos(1 / 3 ** 0.5)

    #for m in range(nx):
        #circuit.ry(pi / 2, xp[2 * m])
        #circuit.ry(3 * pi / 2, xp[2 * m+1])
    #for m in range(ny):
        #circuit.rx(-pi / 2, yp[2 * m])
        #circuit.x(yp[2 * m + 1])
        #circuit.rx(-pi / 2, yp[2 * m + 1])
    #for m in range(nz):
        #circuit.x(zp[2 * m + 1])
    #for m in range(nz):
        #circuit.rx(pi/2, zp[2*m])
        #circuit.rz(pi/4, zp[2*m])
        #circuit.rx(-pi/2, zp[2*m+1])
        #circuit.rz(pi/4, zp[2*m+1])
    for m in range(qubit_number):
        circuit.rx(pi/2+par[6*number], m)
    for n in range(ny):
        circuit.rz(-pi/4+par[6*number+1], yp[2*n])
        circuit.rz(3*pi/4+par[6*number+2], yp[2*n+1])
    for n in range(nz):
        circuit.rx(par[6*number+3], zp[2*n])
        circuit.rx(par[6*number+4], zp[2*n+1])
    #Anatz_2基于哈密顿量的变分参数设定：
    for j in range(number):
        """
```

```
        The hamiltonian for X part
        """
        circuit.rx(2 * par[6 * j], range(qubit_number))
        for px in range(nx):
            circuit.h(xp[2*px])
            circuit.h(xp[2*px+1])
            circuit.cnot(xp[2*px+1], xp[2*px])
            circuit.rz(2 * par[(6 * j) + 1], xp[2*px])
            circuit.cnot(xp[2 * px + 1], xp[2 * px])
            circuit.h(xp[2 * px + 1])
            circuit.h(xp[2 * px])
        """
        The hamiltonian for Y part
        """
        circuit.ry(2 * par[(6 * j) + 2], range(qubit_number))
        for py in range(ny):
            circuit.rx(pi/2, yp[2*py])
            circuit.rx(pi/2, yp[2*py+1])
            circuit.cnot(yp[2*py+1], yp[2*py])
            circuit.rz(2 * par[(6 * j) + 3], yp[2*py])
            circuit.cnot(yp[2 * py + 1], yp[2 * py])
            circuit.rx(-pi / 2, yp[2 * py + 1])
            circuit.rx(-pi / 2, yp[2 * py])
        """
        The hamiltonian for Z part
        """
        circuit.rz(2 * par[(6 * j) + 4], range(qubit_number))
        for pz in range(nz):
            circuit.cnot(zp[2*pz+1], zp[2*pz])
            circuit.rz(2 * par[(6 * j) + 5], zp[2*pz])
            circuit.cnot(zp[2*pz+1], zp[2*pz])
        """
        The hamiltonian of
        """
    return circuit


def XX(par, i, j):
    """
    Gives expectation value of XX
    sub-hamiltonian from measurement
    on parametric state
```

```
    :return: expectation value of XX
    """
    circuit = parametric_state(par)
    ########## Transformation on XX ##########
    q = circuit.qregs[0]
    c = circuit.cregs[0]
    circuit.ry(-np.pi / 2, q[i])
    circuit.ry(-np.pi / 2, q[j])
    circuit.measure(q, c)
    ############ XX measurement ##############
    exp_XX = double_measurement(circuit, i, j)
    return exp_XX


def YY(par, i, j):
    """
    Gives expectation value of YY
    sub-hamiltonian from measurement
    on parametric state.
    :param theta: angle in radian
    :return: expectation value of YY
    """
    circuit = parametric_state(par)
    ########## Transformation on YY #########
    q = circuit.qregs[0]
    c = circuit.cregs[0]
    circuit.rx(np.pi / 2, q[i])
    circuit.rx(np.pi / 2, q[j])
    circuit.measure(q, c)
    ############ YY Measurement ###########
    exp_YY = double_measurement(circuit, i, j)
    return exp_YY


def ZZ(par, i, j):
    """
    Gives expectation value of ZZ
    sub-hamiltonian from measurement
    on parametric state.
    :return: expectation value of ZZ
    """
    circuit = parametric_state(par)
```

```
    ##########################################
    q = circuit.qregs[0]
    c = circuit.cregs[0]
    circuit.measure(q, c)
    ###########  ZZ measurement ################
    exp_ZZ = double_measurement(circuit, i, j)
    return exp_ZZ


def X(par, i):
    """
    Gives expectation value of X
    sub-hamiltonian from measurement
    on parametric state.
    :return: expectation value of X
    """
    circuit = parametric_state(par)
    q = circuit.qregs[0]
    c = circuit.cregs[0]
    circuit.ry(-np.pi / 2, q[i])
    circuit.measure(q, c)
    exp_X = single_measurement(circuit, i)
    return exp_X


def Y(par, i):
    """
    Gives expectation value of Y
    sub-hamiltonian from measurement
    on parametric state.
    :return: expectation value of Y
    """
    circuit = parametric_state(par)
    q = circuit.qregs[0]
    c = circuit.cregs[0]
    circuit.rx(np.pi / 2, q[i])
    circuit.measure(q, c)
    exp_Y = single_measurement(circuit, i)
    return exp_Y


def Z(par, i):
```

```
    """
    Gives expectation value of Z
    sub-hamiltonian from measurement
    on parametric state.
    :return: expectation value of Z
    """
    circuit = parametric_state(par)
    q = circuit.qregs[0]
    c = circuit.cregs[0]
    circuit.measure(q, c)
    exp_Z = single_measurement(circuit, i)
    return exp_Z


def vqe(par):  # ---------- creates ansatz measures and performs addition to get the
    """
    Contain the complete Hamiltonian
    :return: expectation value of whole hamiltonian
    """
    Jx = -1
    Jy = -1
    Jz = -1
    hx = 0
    hy = 0
    hz = -1
    E = 0
    Mz = 0
    Mx = 0
    plaquette = 0
    for jx in range(nx):
        E = E - Jx * (XX(par, xp[2*jx], xp[2*jx+1]))
    for jy in range(ny):
        E = E - Jy * (YY(par, yp[2*jy], yp[2*jy+1]))
    for jz in range(nz):
        E = E - Jz * (ZZ(par, zp[2*jz], zp[2*jz+1]))
    for i in range(qubit_number):
        E = E + hx * X(par, i) + hy * Y(par, i) + hz * Z(par, i)
    for j in range(qubit_number):
        Mz = Mz + Z(par, j)
    for j in range(qubit_number):
        Mx = Mx + X(par, j)
    #for i in range(1, qubit_number - 1, 2):
```

```
        #loop = X(par, i - 1) * Z(par, i) * Y(par, (i + 1) - 6 * ((i + 1) % 6 == 0)) * X(par,
        #plaquette += loop
    #plaquette = abs(plaquette / qubit_number * 2)

    Mz = Mz / qubit_number
    Mx = Mx / qubit_number
    global runtime_count
    runtime_count += 1
    if runtime_count % 5 == 0:
        print('Optimization time:{}'.format(runtime_count))
        print('Present Curcuit Parameters:{}'.format(par))
        print('Current Energy:{}'.format(E))
        print('normalized total magnetization in z direction:{}'.format(Mz))
        print('normalized total magnetization in x direction:{}'.format(Mx))
        #print('Plaquette:{}'.format(plaquette))
    return E

def doukey_check(my_dict: dict, my_key: str, i, j):
    """
    If key is missing returns 0
    otherwise the corresponding value.
    :param my_dict: the dictionary
    :param my_key: key (string)
    :return: 0 or value corresponding to key
    """
    response = 0
    for everyKey in my_dict:
        if everyKey[qubit_number - 1 - i] == my_key[qubit_number - 1 - i] and everyKey[qubit_
            response = response + my_dict[everyKey]
    return response

def sigkey_check(my_dict:dict, my_key:str, i):
    response = 0
    for everyKey in my_dict:
        if everyKey[qubit_number - 1 - i] == my_key[qubit_number - 1 - i]:
            response = response + my_dict[everyKey]
    return response

def double_measurement(circuit, i, j):  # ———————————— Takes a quantum circuit and perf
    """
    Takes the quantum circuit as
    input to perform measurement
```

```
    :param circuit: quantumm circuit
    :return: expectation value
    """
    shots = 5000
    backend = BasicAer.get_backend('qasm_simulator')
    job = execute(circuit, backend, shots=shots)
    result = job.result()
    counts = result.get_counts()
    p11 = ''
    p00 = ''
    for t in range(qubit_number):
        if t == qubit_number - 1 - i or t == qubit_number - 1 - j:
            p11 = p11 + '1'
        else:
            p11 = p11 + '0'
        p00 = p00 + '0'
    p01 = ''
    for t in range(qubit_number):
        if t == qubit_number - 1 - j:
            p01 = p01 + '1'
        else:
            p01 = p01 + '0'
    p10 = ''
    for t in range(qubit_number):
        if t == qubit_number - 1 - i:
            p10 = p10 + '1'
        else:
            p10 = p10 + '0'
    n00 = doukey_check(counts, p00, i, j)
    n01 = doukey_check(counts, p01, i, j)
    n10 = doukey_check(counts, p10, i, j)
    n11 = doukey_check(counts, p11, i, j)
    expectation_value = ((n00 + n11) - (n01 + n10)) / shots
    return expectation_value

def single_measurement(circuit, i):
    """
        Takes the quantum circuit as
        input to perform measurement
        :param circuit: quantumm circuit
        :return: expectation value
        """
```

```
    shots = 5000
    backend = BasicAer.get_backend('qasm_simulator')
    job = execute(circuit, backend, shots=shots)
    result = job.result()
    counts = result.get_counts()
    p1 = ''
    p2 = ''
    for t in range(qubit_number):
        if t == qubit_number - 1 - i:
            p1 = p1 + '1'
        else:
            p1 = p1 + '0'
        p2 = p2 + '0'
    n1 = sigkey_check(counts, p1, i)
    n0 = sigkey_check(counts, p2, i)
    expectation_value = (n0 - n1) / shots
    return expectation_value

qubit_number = 12

#晶格结构及配对,周期性边界条件

xp = np.array([1, 2, 3, 4, 7, 8, 9, 10, 11, 6, 5, 0])
yp = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
zp = np.array([2, 7, 4, 9, 0, 11, 8, 1, 10, 3, 6, 5])
nx = int((xp.size)/2)
ny = int((yp.size)/2)
nz = int((zp.size)/2)
#优化参数初值
par_array = np.array([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0, 0, 0, 0, 0])
result = pybobyqa.solve(vqe, par_array)
print(result)
```

A.2 Code of square-octagon lattice

```
import numpy as np
from qiskit import *
import pybobyqa

pi = np.pi
runtime_count = 0
```

```
def parametric_state(par):
    """
    one layer construct
    :return: Ansatz-target state
    """
    q = QuantumRegister(qubit_number)
    c = ClassicalRegister(qubit_number)
    circuit = QuantumCircuit(q, c)
    number = int((par.size)/6)
    theta = np.arccos(1 / 3 ** 0.5)

    #for m in range(nx):
        #circuit.ry(pi / 2, xp[2 * m])
        #circuit.ry(3 * pi / 2, xp[2 * m+1])
    #for m in range(ny):
        #circuit.rx(-pi / 2, yp[2 * m])
        #circuit.x(yp[2 * m + 1])
        #circuit.rx(-pi / 2, yp[2 * m + 1])
    #for m in range(nz):
        #circuit.x(zp[2 * m + 1])
    #for m in range(nz):
        #circuit.rx(pi/2, zp[2*m])
        #circuit.rz(pi/4, zp[2*m])
        #circuit.rx(-pi/2, zp[2*m+1])
        #circuit.rz(pi/4, zp[2*m+1])
    for m in range(qubit_number):
        circuit.rx(pi/2+par[6*number], m)
    for n in range(ny):
        circuit.rz(-pi/4+par[6*number+1], yp[2*n])
        circuit.rz(3*pi/4+par[6*number+2], yp[2*n+1])
    for n in range(nz):
        circuit.rx(par[6*number+3], zp[2*n])
        circuit.rx(par[6*number+4], zp[2*n+1])
    #Anatz_2基于哈密顿量的变分参数设定：
    for j in range(number):
        """
        The hamiltonian for X part
        """
        circuit.rx(2 * par[6 * j], range(qubit_number))
        for px in range(nx):
            circuit.h(xp[2*px])
            circuit.h(xp[2*px+1])
```

```
            circuit.cnot(xp[2*px+1], xp[2*px])
            circuit.rz(2 * par[(6 * j) + 1], xp[2*px])
            circuit.cnot(xp[2 * px + 1], xp[2 * px])
            circuit.h(xp[2 * px + 1])
            circuit.h(xp[2 * px])
        """
        The hamiltonian for Y part
        """
        circuit.ry(2 * par[(6 * j) + 2], range(qubit_number))
        for py in range(ny):
            circuit.rx(pi/2, yp[2*py])
            circuit.rx(pi/2, yp[2*py+1])
            circuit.cnot(yp[2*py+1], yp[2*py])
            circuit.rz(2 * par[(6 * j) + 3], yp[2*py])
            circuit.cnot(yp[2 * py + 1], yp[2 * py])
            circuit.rx(-pi / 2, yp[2 * py + 1])
            circuit.rx(-pi / 2, yp[2 * py])
        """
        The hamiltonian for Z part
        """
        circuit.rz(2 * par[(6 * j) + 4], range(qubit_number))
        for pz in range(nz):
            circuit.cnot(zp[2*pz+1], zp[2*pz])
            circuit.rz(2 * par[(6 * j) + 5], zp[2*pz])
            circuit.cnot(zp[2*pz+1], zp[2*pz])
        """
        The hamiltonian of
        """
    return circuit


def XX(par, i, j):
    """
    Gives expectation value of XX
    sub-hamiltonian from measurement
    on parametric state
    :return: expectation value of XX
    """
    circuit = parametric_state(par)
    ########## Transformation on XX ##########
    q = circuit.qregs[0]
    c = circuit.cregs[0]
```

```
    circuit.ry(-np.pi / 2, q[i])
    circuit.ry(-np.pi / 2, q[j])
    circuit.measure(q, c)
    ############ XX measurement ##############
    exp_XX = double_measurement(circuit, i, j)
    return exp_XX


def YY(par, i, j):
    """
    Gives expectation value of YY
    sub-hamiltonian from measurement
    on parametric state.
    :param theta: angle in radian
    :return: expectation value of YY
    """
    circuit = parametric_state(par)
    ########## Transformation on YY #########
    q = circuit.qregs[0]
    c = circuit.cregs[0]
    circuit.rx(np.pi / 2, q[i])
    circuit.rx(np.pi / 2, q[j])
    circuit.measure(q, c)
    ############ YY Measurement ###########
    exp_YY = double_measurement(circuit, i, j)
    return exp_YY


def ZZ(par, i, j):
    """
    Gives expectation value of ZZ
    sub-hamiltonian from measurement
    on parametric state.
    :return: expectation value of ZZ
    """
    circuit = parametric_state(par)
    ########################################
    q = circuit.qregs[0]
    c = circuit.cregs[0]
    circuit.measure(q, c)
    ########## ZZ measurement ##############
    exp_ZZ = double_measurement(circuit, i, j)
```

```
    return exp_ZZ


def X(par, i):
    """
    Gives expectation value of X
    sub-hamiltonian from measurement
    on parametric state.
    :return: expectation value of X
    """
    circuit = parametric_state(par)
    q = circuit.qregs[0]
    c = circuit.cregs[0]
    circuit.ry(-np.pi / 2, q[i])
    circuit.measure(q, c)
    exp_X = single_measurement(circuit, i)
    return exp_X


def Y(par, i):
    """
    Gives expectation value of Y
    sub-hamiltonian from measurement
    on parametric state.
    :return: expectation value of Y
    """
    circuit = parametric_state(par)
    q = circuit.qregs[0]
    c = circuit.cregs[0]
    circuit.rx(np.pi / 2, q[i])
    circuit.measure(q, c)
    exp_Y = single_measurement(circuit, i)
    return exp_Y


def Z(par, i):
    """
    Gives expectation value of Z
    sub-hamiltonian from measurement
    on parametric state.
    :return: expectation value of Z
    """
```

```
    circuit = parametric_state(par)
    q = circuit.qregs[0]
    c = circuit.cregs[0]
    circuit.measure(q, c)
    exp_Z = single_measurement(circuit, i)
    return exp_Z


def vqe(par):  # ———————————— creates ansatz measures and performs addition to get the
    """
    Contain the complete Hamiltonian
    :return: expectation value of whole hamiltonian
    """
    Jx = -1
    Jy = -1
    Jz = -1
    hx = 0
    hy = 0
    hz = -1
    E = 0
    Mz = 0
    Mx = 0
    plaquette = 0
    for jx in range(nx):
        E = E - Jx * (XX(par, xp[2*jx], xp[2*jx+1]))
    for jy in range(ny):
        E = E - Jy * (YY(par, yp[2*jy], yp[2*jy+1]))
    for jz in range(nz):
        E = E - Jz * (ZZ(par, zp[2*jz], zp[2*jz+1]))
    for i in range(qubit_number):
        E = E + hx * X(par, i) + hy * Y(par, i) + hz * Z(par, i)
    for j in range(qubit_number):
        Mz = Mz + Z(par, j)
    for j in range(qubit_number):
        Mx = Mx + X(par, j)
    #for i in range(1, qubit_number - 1, 2):
        #loop = X(par, i - 1) * Z(par, i) * Y(par, (i + 1) - 6 * ((i + 1) % 6 == 0)) * X(par,
        #plaquette += loop
    #plaquette = abs(plaquette / qubit_number * 2)

    Mz = Mz / qubit_number
    Mx = Mx / qubit_number
```

```
    global runtime_count
    runtime_count += 1
    if runtime_count % 5 == 0:
        print('Optimization time:{}'.format(runtime_count))
        print('Present Curcuit Parameters:{}'.format(par))
        print('Current Energy:{}'.format(E))
        print('normalized total magnetization in z direction:{}'.format(Mz))
        print('normalized total magnetization in x direction:{}'.format(Mx))
        #print('Plaquette:{}'.format(plaquette))
    return E

def doukey_check(my_dict: dict, my_key: str, i, j):
    """
    If key is missing returns 0
    otherwise the corresponding value.
    :param my_dict: the dictionary
    :param my_key: key (string)
    :return: 0 or value corresponding to key
    """
    response = 0
    for everyKey in my_dict:
        if everyKey[qubit_number - 1 - i] == my_key[qubit_number - 1 - i] and everyKey[qubit_
            response = response + my_dict[everyKey]
    return response

def sigkey_check(my_dict:dict, my_key:str, i):
    response = 0
    for everyKey in my_dict:
        if everyKey[qubit_number - 1 - i] == my_key[qubit_number - 1 - i]:
            response = response + my_dict[everyKey]
    return response

def double_measurement(circuit, i, j):  # ---------- Takes a quantum circuit and perf
    """
    Takes the quantum circuit as
    input to perform measurement
    :param circuit: quantumm circuit
    :return: expectation value
    """
    shots = 5000
    backend = BasicAer.get_backend('qasm_simulator')
    job = execute(circuit, backend, shots=shots)
```

```
    result = job.result()
    counts = result.get_counts()
    p11 = ''
    p00 = ''
    for t in range(qubit_number):
        if t == qubit_number - 1 - i or t == qubit_number - 1 - j:
            p11 = p11 + '1'
        else:
            p11 = p11 + '0'
        p00 = p00 + '0'
    p01 = ''
    for t in range(qubit_number):
        if t == qubit_number - 1 - j:
            p01 = p01 + '1'
        else:
            p01 = p01 + '0'
    p10 = ''
    for t in range(qubit_number):
        if t == qubit_number - 1 - i:
            p10 = p10 + '1'
        else:
            p10 = p10 + '0'
    n00 = doukey_check(counts, p00, i, j)
    n01 = doukey_check(counts, p01, i, j)
    n10 = doukey_check(counts, p10, i, j)
    n11 = doukey_check(counts, p11, i, j)
    expectation_value = ((n00 + n11) - (n01 + n10)) / shots
    return expectation_value

def single_measurement(circuit, i):
    """
        Takes the quantum circuit as
        input to perform measurement
        :param circuit: quantumm circuit
        :return: expectation value
        """
    shots = 5000
    backend = BasicAer.get_backend('qasm_simulator')
    job = execute(circuit, backend, shots=shots)
    result = job.result()
    counts = result.get_counts()
    p1 = ''
```

```
    p2 = ''
    for t in range(qubit_number):
        if t == qubit_number - 1 - i:
            p1 = p1 + '1'
        else:
            p1 = p1 + '0'
        p2 = p2 + '0'
    n1 = sigkey_check(counts, p1, i)
    n0 = sigkey_check(counts, p2, i)
    expectation_value = (n0 - n1) / shots
    return expectation_value

qubit_number = 8
#晶格结构及配对,周期性边界条件
xp = np.array([0, 1, 2, 3, 4, 5, 6, 7])
yp = np.array([0, 4, 1, 5, 6, 2, 7, 3])
zp = np.array([1, 2, 3, 0, 5, 6, 7, 4])
nx = int((xp.size)/2)
ny = int((yp.size)/2)
nz = int((zp.size)/2)
#优化参数初值
par_array = np.array([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0, 0, 0, 0, 0])
result = pybobyqa.solve(vqe, par_array)
print(result)
```