Hybrid Quantum-Classical Algorithms: Variational Quantum Eigensolver (VQE) Algorithm

In [1]:
```
%matplotlib inline
# Importing standard Qiskit libraries and configuring account
from qiskit import QuantumCircuit, execute, Aer, IBMQ
from qiskit.compiler import transpile, assemble
from qiskit.tools.jupyter import *
from qiskit.visualization import *
# Loading your IBM Q account(s)
provider = IBMQ.load_account()
```

```
/opt/conda/lib/python3.7/site-packages/qiskit/providers/ibmq/ibmqfactory.py:192: UserWarning: Timestamps in IBMQ back
end properties, jobs, and job results are all now in local time instead of UTC.
  warnings.warn('Timestamps in IBMQ backend properties, jobs, and job results '
```

Variational Quantum Eigensolver (VQE)

As you may have heard during this summer school, we are still far away from building a fault-tolerant quantum computer- those with full error correction. The available quantum computers in the near term, called Noisy Intermediate Scale Quantum computers (NISQ), are too small to support error correction and have high levels of noise. Therefore, implementing long quantum circuits on such devices will not likely provide satisfyingly accurate results. A class of algorithms, called the variational algorithms, on the other hand are suited to be performed on a quantum computer since they can be implemented using shallow circuits. Here you are going to learn about one of these variational algorithms: "The variational quantum eigensolver (VQE)".

VQE is used to find the smallest/ minimum eigenvalue of some matrix H using a shallow circuit. This algorithm is a Hybrid algorithm as it divides the task of finding the smallest eigenvalue between quantum and classical processors. This is done by taking an advantage of the natural preparation of variational quantum states and measuring expectation values on quantum computers, which are difficult to do on classical computers due to the expoenential growth of the Hilbert space (quantum qubits state space) with system size (the number of qubits). Moreover, the algorithm uses classical processors to do tasks they are capable of performing easily. More specifically, VQE uses a classical optimizer to minimize the expectation value (you'll see in what follows what does this mean) obtained from the quantum computer by varying the variational qubits state.

VQE- Main idea

The main idea of the variational quantum eigensolver is implementing the quantum variational principle and Ritz method on both a quantum computer and a classical computer. The basic idea of these methods is to search for a trial qubits state (wavefunction) $|\psi\rangle$ for the problem, which depends on adjustable parameters $\vec{\theta} = \theta_1,\ \theta_2, . .$, that we call variational parameters.

The "expectation value" of H is given by:

$$\langle\psi(\vec{\theta})|H|\psi(\vec{\theta})\rangle$$

Here, $|\psi(\vec{\theta})\rangle$ is the qubits state which depends on the parameters set $\vec{\theta} = \theta_1,\ \theta_2, . .$ which for our case can be considered as a variational "column vector" which lives in a part of the whole vector space. Morever, $\langle\psi(\vec{\theta})| = (|\psi(\vec{\theta})\rangle)^\dagger$ can be considered as a row vector. The whole expression $\langle\psi(\vec{\theta})|H|\psi(\vec{\theta})\rangle$ gives a scalar that depends on the parameters $\vec{\theta} = \theta_1,\ \theta_2, .$. We'll denote this expression by $E(\vec{\theta})$:

$$E(\vec{\theta}) = \langle\psi(\vec{\theta})|H|\psi(\vec{\theta})\rangle.$$

If $|\psi(\vec{\theta})\rangle$ is an eigenvector or close to an eigenvector of the matrix H, then this expectation value gives the corresponding eigenvalue or something close to it.

According to the variational principle, this quantity always satisfies

$$E(\vec{\theta}) \geq E_0$$

where E_0 is the smallest eigenvalue.

In the VQE algorithm, we adjust the parameters $\vec{\theta} = \theta_1,\ \theta_2, . .$ untill this quantity $E(\vec{\theta})$ is minimized.

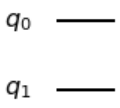
The search in the parameter space becomes unfeasible and very hard to do on a classical computer very rapidly as the computational resources required for preparing the quantum states on a classical computer grow exponentially as a function of the system size. On the other hand, the quantum machines operate on the Hilbert space, whose dimension grows also exponentially. If we have N qubits, then the dimension will be 2^N. This vast space of the quantum states can be used to tackle the exponential growth of the complexity of the optimization problems. The VQE exploits the advantage of a quantum computer to prepare a parametrized trial state, which is intractable by a classical computer, and performing measurements on this state to compute the parametrized expectation values. Also, the optimization of the parameters is done on a classical computer which can do this with the available hardware nowadays better than a quantum computer.

Sketch of the VQE algorithm

First Step: Performed on the quantum computer.

A simple starting state which is easy to be prepared on a quantum computer is initialized, e.g. the state $|000000\rangle$. So let's assume we have a system of 2 qubits, then in qiskit as we've seen in the previous tutorials, this can be done by:

In [1]:
```
##Example for an initialized state of the qubits -- All qubits in the state 0##
from qiskit import QuantumCircuit

qc_VQE=QuantumCircuit(2)

qc_VQE.draw('mpl')
```



Second Step: Performed on the quantum computer.

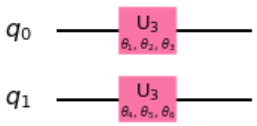
A parametrized state $\psi(\vec{\theta})$ that depends on a set of control parameters, $\vec{\theta} = \theta_1, \theta_2, .. \theta_m$ is prepared by applying what is often called a variational form, $U(\theta)$, which is a sequence of gates that depend on the set of parameters $\vec{\theta}$, on the initial state prepared in the first step.

For the 2-qubits example, we may pick the following variational form, composed of the following gates:

1) Performing a generic rotation on the first and second qubit using the $U3$ gates. 2) Entangling the two qubits using a CNOT gate. 3) Again, performing a generic rotation on the first and second qubit using the $U3$ gates.

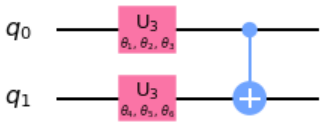
let's see the implementation of such circuit:

In [2]:
```
from qiskit.circuit import Parameter

params = [Parameter(r'$\theta_1$'), Parameter(r'$\theta_2$'), Parameter(r'$\theta_3$'),Parameter(r'$\theta_4$'), Param
eter(r'$\theta_5$'), Parameter(r'$\theta_6$')]

qc_VQE.u3(params[0],params[1],params[2],0)
qc_VQE.u3(params[3],params[4],params[5],1)
qc_VQE.draw()
```



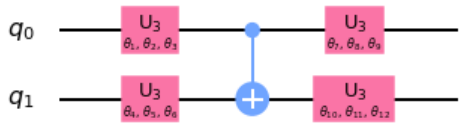
Now we'll entangle the two qubits using the CNOT gate

In [3]:
```
qc_VQE.cx(0,1)
qc_VQE.draw()
```



Now that we have entangled the qubits, we'll perform a generic rotation of both qubits once again

In [4]:
```
params2 = [Parameter(r'$\theta_7$'), Parameter(r'$\theta_8$'), Parameter(r'$\theta_9$'), Parameter(r'$\theta_{10}$'),
Parameter(r'$\theta_{11}$'), Parameter(r'$\theta_{12}$')]

qc_VQE.u3(params2[0],params2[1],params2[2],0)
qc_VQE.u3(params2[3],params2[4],params2[5],1)
qc_VQE.draw()
```



Summary so far:

We have started from the stae $|00\rangle$, where both the first and second qubits are in state $|0\rangle$, and the full system state is $|00\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$.

To obtain a trial state that covers part of/ the full Hilbert space (qubits state), we applied the following sequence of gate: $U3(\theta_1, \theta_2, \theta_3)_{q_0}$, $U3(\theta_4, \theta_5, \theta_6)_{q_1}$, $CNOT_{q_0,q_1}$, $U3(\theta_7, \theta_8, \theta_9)_{q_0}$ and $U3(\theta_{10}, \theta_{11}, \theta_{12})_{q_1}$. The obtained variational state prepared using such circuit is

$$|\psi(\vec{\theta})\rangle = U3(\theta_{10}, \theta_{11}, \theta_{12})_{q_1}\, U3(\theta_7, \theta_8, \theta_9)_{q_0}\, CNOT_{q_0,q_1}\, U3(\theta_4, \theta_5, \theta_6)_{q_1}\, U3(\theta_1, \theta_2, \theta_3)_{q_0}|00\rangle,$$

where $\vec{\theta} = (\theta_1, \theta_2, \ldots, \theta_1 2)$.

Third step: Performed on the quantum computer

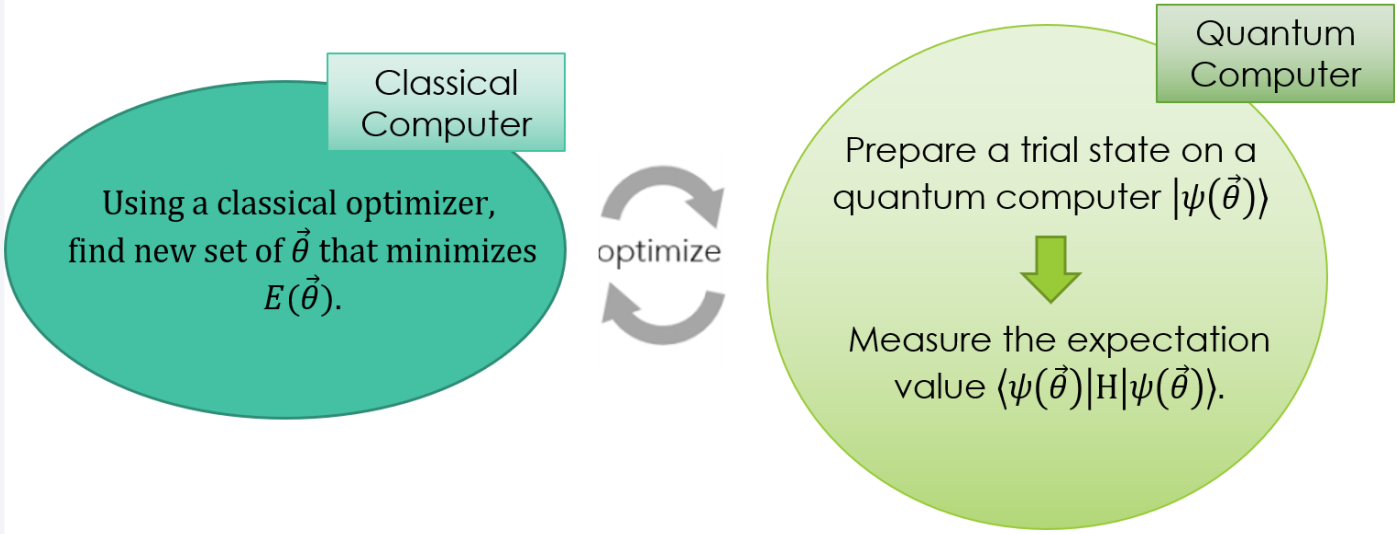
The expectation value of the problem matrix H that acts N qubits, $\langle\psi(\vec{\theta})|H|\psi(\vec{\theta})\rangle$ is computed on the quantum computer.

Last step: Performed on the classical computer

In this step, we feed the obtained result $E_g(\vec{\theta})$ to a classical optimizer. The classical optimizer's job is to tell us how can we change the parameters to obtain a lower energy. The outcome of the classical optimizer will be a new set of parameters $\vec{\theta}_{new}$ for which $E_g(\vec{\theta}_{new}) < E_g(\vec{\theta})$.

The feedback loop:

What will happen next? Now we will have a loop, the outcome obtained by the classical optimizers, $\vec{\theta}_{new}$ is fed to the quantum computer, a new state is prepared: $|\psi(\vec{\theta}_{new}\rangle$, then the expectation values in the new state $\langle\psi(\vec{\theta}_{new})|\sigma_i|\psi(\vec{\theta}_{new})\rangle$ are computed as explained in step #3. This is followed again by classically optimizing the parameters using a classical optimizer whose outcome is a new set of parameters $\vec{\theta}_{new2}$ for which $E_g(\vec{\theta}_{new2}) < E_g(\vec{\theta}_{new})$. The new obtained parameters are adjusted accordingly on the quantum computer, and this process, will be repeated, where we will have a feedback loop between a classical computer and a quantum co-processor, until the groundstate energy is obtained with a desired precision.



Implementation in Qiskit: Example- Ising model of two qubits system

In the previous steps, we considered a system of two qubits q_0 and q_1. We also picked a variational form for creating the trial state we need for the VQE algorithm, and also implemented it using Qiskit. However, the main problem we are trying to solve is finding the minimum eigenvalue of the Hamiltonian (groundstate energy). If so, then we'll need to know how does the Hamiltonian of the system look like. For this example, we will consider the following matrix:

$$H = Z_{q1} \otimes Z_{q_0},$$

where Z is the Z-Pauli operator, whose matrix representation in the computational basis is: $Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$. This matrix describes an object called the Ising Hamiltonian.

Our goal now is to use the VQE implemented on a quantum computer to compute the smallest eigenvalue of H.

The example we picked here is simple for which the groundstate energy will be easily computed by hand. So before using the VQE, let's pause for a moment, and try to

1) write the Hamiltonian matrix representation in the computational basis of the two qubits ($|00\rangle$, $|01\rangle$ $|10\rangle$ and $|11\rangle$).

2) Diagonalize the obtained Hamiltonian and compute the minimum eigenvalue E_g.

(Or do 2 directly).

Answer:

$$H = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This is a diagonal matrix, which is what we expect to have. The minimum eigenvalue of this matrix is -1.

Now let's see how we can use the VQE algorithm to compute the groundstate energy of the above Hamiltonian and check the result. This we will do on Qasm simulator and a real device.

Instead of implementing the full algorithm by ourselves, we will use the method VQE developed by the IBM team which is also a built-in function that can be imported from Qiskit.

So what arguments does VQE method take?

As explained in the documentaion page of VQE: https://qiskit.org/documentation/stubs/qiskit.aqua.algorithms.VQE.html (https://qiskit.org/documentation/stubs/qiskit.aqua.algorithms.VQE.html), VQE takes various parameters, her we will focus on the first 4 (the rest are optional):

Parameters:

- **operator** (`Union` [`OperatorBase` , `LegacyBaseOperator` , `None`]) – Qubit operator of the Observable
- **var_form** (`Union` [`QuantumCircuit` , `VariationalForm` , `None`]) – A parameterized circuit used as Ansatz for the wave function.
- **optimizer** (`Optional` [`Optimizer`]) – A classical optimizer.
- **initial_point** (`Optional` [`ndarray`]) – An optional initial point (i.e. initial parameter values) for the optimizer. If `None` then VQE will look to the variational form for a preferred point and if not will simply compute a random one.

What do we have here?

1. First we have the operator $\hat{O}$, for which we wish to find the parameters that minimize its expectation value. In our case, the operator we are considering is the Hamiltonian of the system, $\hat{O} = H$. For our case,

In [16]:
```
from qiskit.aqua.operators import Z,I
ising=Z^Z
```

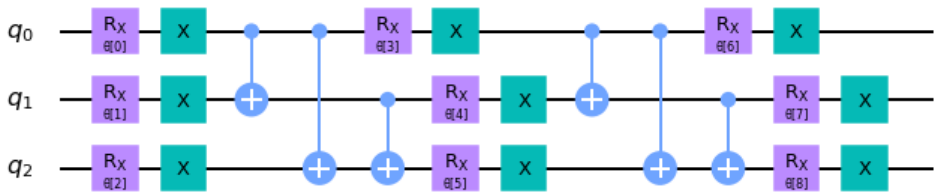
1. The variational form: For the variational circuit we may pick one of the available variational circuits in Qiskit, for example, the EfficientSU2. Using this method we can create a variational circuit which is composed of layers of SU(2) gates: e.g. Paulis ('x','y','z'), rotation gates ('rx','ry','rz') and between the layers, cx gates are applied in order to introduce entanglement between the qubits.
   You are allowed also to specify the entanglement structure. Assuming we have N qubits in the system, the available options are:

- 'full': each q_n is entangled using a CX-gate with q_m for $m = n+1, \ldots N$.
- 'linear':each q_n is entangled using a CX-gate with q_m for $m = n+1$.
- 'circular':similar to the 'linear' case, but also entangling the first and last qubit together using a CX-gate.
- 'sca': shifted circular alternating (see the example below for details).

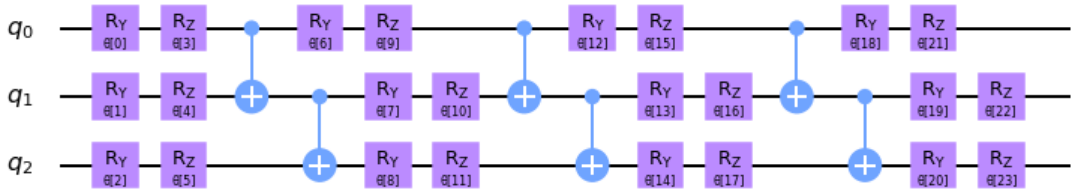
You may also provide a list of integer-pairs specifying the indices of qubits entangled with one another. The repitition number of the layers can be specified by setting the parameter 'reps'. If it is not specified, then by default it is set to 3.

To make this clear, let's take a look at the following five examples in which we will be considering a system composed of 3 qubits rather than 2-qubits for illustrating the different entanglement structure specifications.

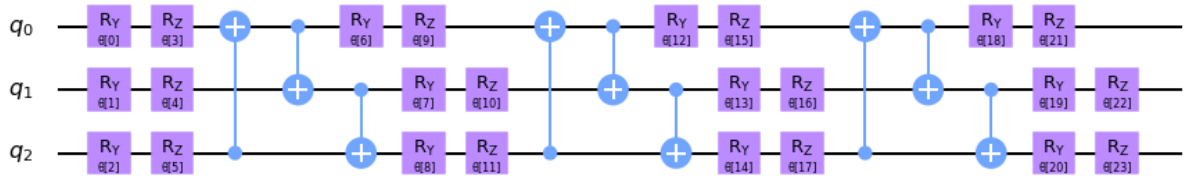
In [54]:
```
##### Layers of rx and x gates acting on each qubit with cx gates. The entanglement here is specified to be "full" ###
#####
from qiskit.circuit.library import EfficientSU2

ansatz = EfficientSU2(3, su2_gates=['rx','x'], entanglement='full', reps=2)
ansatz.draw('mpl')
```



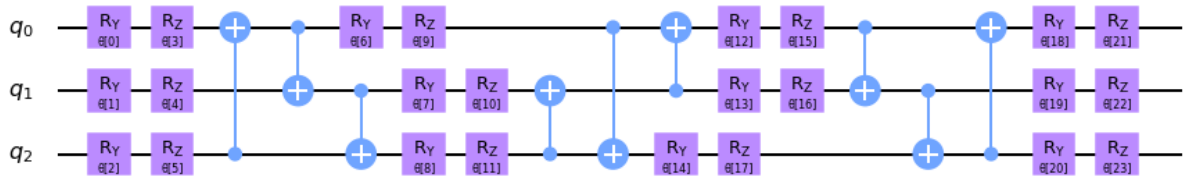
In [55]:
```
##### Layers of ry and rz gates (this is the default, no need to specify) acting on each qubit with cx gates. The enta
nglement here is specified to be "linear" ########
from qiskit.circuit.library import EfficientSU2

ansatz = EfficientSU2(3, entanglement='linear',reps=3)
ansatz.draw('mpl')
```



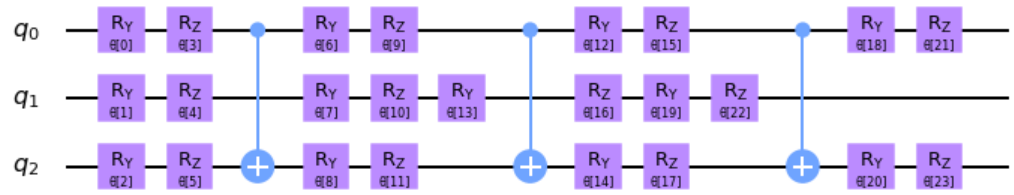
In [48]:
```
##### Layers of ry and rz gates (this is the default, no need to specify) acting on each qubit with cx gates. The enta
nglement here is specified to be "linear" ########
from qiskit.circuit.library import EfficientSU2

ansatz = EfficientSU2(3, entanglement='circular')
ansatz.draw('mpl')
```



In [49]:
```
##### Layers of ry and rz gates (this is the default, no need to specify) acting on each qubit with cx gates. The enta
nglement here is specified to be "linear" ########
from qiskit.circuit.library import EfficientSU2

ansatz = EfficientSU2(3, entanglement='sca')
ansatz.draw('mpl')
```

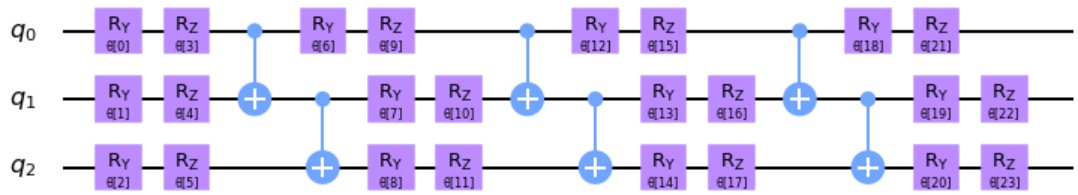


In [42]:
```
##### Layers of ry and rz gates (this is the default, no need to specify) acting on each qubit with cx gates. The enta
nglement here is specified to be "linear" ########
from qiskit.circuit.library import EfficientSU2

ansatz = EfficientSU2(3, entanglement=[[0,2]])
ansatz.draw('mpl')
```



Let's go back to the 2-qubit gates, and consider the following ansatz

In [56]:
```
from qiskit.circuit.library import EfficientSU2

ansatz1 = EfficientSU2(2, entanglement='linear',)
ansatz.draw('mpl')
```



or we can create our own parametrized circuit, as we did here, that can be used as our variational form. For now we will choose the one we already created:

In [57]:
```
ansatz2=qc_VQE
```

1. The calssical optimizer: This parameter can be one of the following optimizers that can be imported from qiskit.aqua.components.optimizers: COBYLA (Constrained Optimization By Linear Approximation optimizer), SPSA (the Simultaneous Perturbation Stochastic Approximation optimizer), SLSQP (Sequential Least SQuares Programming optimizer)..etc. The full list of optimizers in qiskit you may find in the following link: https://qiskit.org/documentation/apidoc/qiskit.aqua.components.optimizers.html (https://qiskit.org/documentation/apidoc/qiskit.aqua.components.optimizers.html). For example, here we'll use the SPSA (For more details about this optimizer, please see https://qiskit.org/documentation/stubs/qiskit.aqua.components.optimizers.SPSA.html (https://qiskit.org/documentation/stubs/qiskit.aqua.components.optimizers.SPSA.html)):

In [6]:
```
from qiskit.aqua.components.optimizers import SPSA
optimizer=SPSA()
```

In [58]:
```
from qiskit.aqua.algorithms import VQE
vqe=VQE(ising,ansatz1,optimizer)

result=vqe.run(Aer.get_backend("statevector_simulator"))
print('Minimal energy  at:', result.eigenvalue)
```

```
Minimal energy  at: (-0.9999948663937912+0j)
```

In [59]:
```
vqe=VQE(ising,ansatz2,optimizer)

result=vqe.run(Aer.get_backend("statevector_simulator"))
print('Minimal energy  at:', result.eigenvalue)
```

```
Minimal energy  at: (-0.9999951779651061+0j)
```

Hackathon Exercise #1

Read the supplementary material explaining the variational quantum eigensolver algorithm (VQE) and its implementation in qiskit.

Challenge: Find using the VQE algorithm the best approximation for the least energy of the matrix representing the following quantum operator

$$H = Z \otimes Z \otimes X \otimes Y \otimes I \otimes I + I \otimes Z \otimes Z \otimes X \otimes Y \otimes I$$

$$I \otimes I \otimes Z \otimes Z \otimes X \otimes Y + 1.2(X \otimes I \otimes I \otimes I \otimes I \otimes I) +$$

$$I \otimes X \otimes I \otimes I \otimes I \otimes I + 0.8(I \otimes I \otimes X \otimes I \otimes I \otimes I) +$$

$$0.6(I \otimes I \otimes I \otimes X \otimes I \otimes I) + 0.4(I \otimes I \otimes I \otimes I \otimes X \otimes I) + 0.2(I \otimes I \otimes I \otimes I \otimes I \otimes X)$$

$$= \sum_{i=0}^{5} h_i \cdot X_i + \sum_{i=0}^{2} Z_i \otimes Z_{i+1} \otimes X_{i+2} \otimes Y_{i+3}, \quad where\ h_i = [0.2, 0.4, 0.6, 0.8, 1, 1.2]$$

and X, Y, Z are the Pauli operators and I is the identity operator. You are allowed to use at most 10 variational gates for building the variational ansatz. The number of the non-variational gates you are allowed to use is not specified. The submitted solution should include the code implementing VQE for this model, a figure showing the ansatz you picked and the optimal eigenvalue you obtained. The quantum part of your code should be run first on a statevector_simulator until you reach a satisfying answer and then check what your code with the optimal variational circuit you obtained in this step gives you when you:

a) Run it on a qasm simulator with depolarizing noise. to do this you need to add the following lines to your code before calling VQE class.

In [ ]:
```
#depolarization noise- randomly flips each bit of the final output with probability p
from qiskit.providers.aer.noise import NoiseModel
from qiskit.aqua import QuantumInstance
from qiskit.providers.aer.noise.errors import pauli_error, depolarizing_error

def get_noise(p):

    error_meas = pauli_error([('X',p), ('I', 1 - p)])

    noise_model = NoiseModel()
    noise_model.add_all_qubit_quantum_error(error_meas, "measure")

    return noise_model

noise_model=get_noise(0.01) # probability for a bit flip=0.01
backend = Aer.get_backend("qasm_simulator")
quantum_instance = QuantumInstance(backend=backend,
                                   shots=1024,
                                   noise_model=noise_model)

vqe=VQE(  ) # add your operator, ansatz, optimizer
result=vqe.run(quantum_instance)
```

b) Run it on a real quantum computer that tries to simulate the code with a noise model generated based on the properties of the 'ibmq_16_melbourne' backend. To do this, please add the following lines to your code:

In [ ]:
```
provider = IBMQ.get_provider(hub='ibm-q')
backend = Aer.get_backend("qasm_simulator")
device = provider.get_backend("ibmq_16_melbourne")
noise_model = NoiseModel.from_backend(device.properties())
quantum_instance = QuantumInstance(backend=backend,
                                   shots=1000,
                                   noise_model=noise_model)
quantum_instance = QuantumInstance(backend=backend,
                                   shots=1024,
                                   noise_model=noise_model)

vqe=VQE(  ) # add your operator, ansatz, optimizer
result=vqe.run(quantum_instance)
```

You should be patient while executing the second part.. let it run and go grap some coffe and a snack and then come back and check if you got your results back!