https://learning.quantum.ibm.com/tutorial/variational-quantum-eigensolve

```
hamiltonian = SparsePauliOp.from_list(
    [("YZ", 0.3980), ("ZI", -0.3980), ("ZZ", -0.0113), ("XX", 0.1810)]
)
```

**A matrix representing the total energy of a system. Rather than the entire matrix it's defined "sparsely" by multiple Pauli operators (matrices) acting on 2 qubits at a time; the string component of the tuple. The Pauli operators represent the spin state (up/down) of a single qubit and combine it into a tensor product; combining the two matrices produces a third matrice representing their combined system. The floating point number is the coefficient of these combined systems, representing their strength in this Hamiltonian.**

```
ansatz = EfficientSU2(hamiltonian.num_qubits)
ansatz.decompose().draw("mpl", style="iqp")
```

**The ansatz will be the circuit that is our trial state throughout the experiment; the goal being to find the lowest energy (ground state) of the Hamiltonian. The circuit constructed typically involves alternating layers of single-qubit rotations (Ry and Rx) and entanglement gates (CNOT) acting on pairs of qubits, like our combined Pauli operators which were also acting on pairs of qubits. The method .decompose() breaks down the circuit into the basic gates that make it up and the .draw() method visualizes it in Matplotlib.**

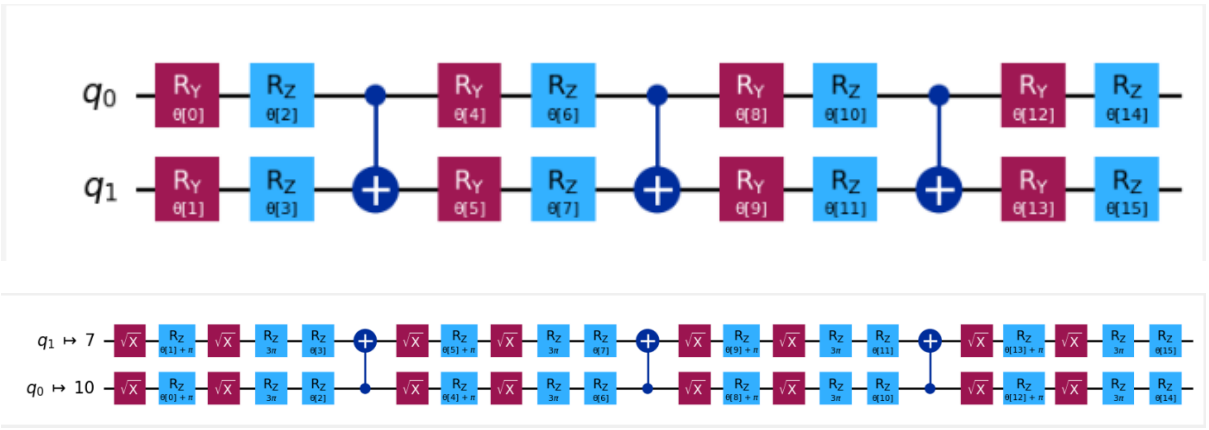
```
target = backend.target
pm = generate_preset_pass_manager(target=target, optimization_level=3)
ansatz_isa = pm.run(ansatz)
```

**"target" will be the quantum hardware on which the ansatz will be run. "pm" creates a pass manager; this is a manager for a series of instructions (passes). It takes the "target", the quantum hardware to be run on as input. This allows it to create a set of instructions tailored to the chosen quantum hardware capabilities. The optimization_level parameter controls the aggressiveness of the optimization process in the pass manager, with higher levels creating more complex but more efficient circuits. pm then takes the ansatz circuit as input**

**and applies the optimization passes defined in pm based on the quantum hardware (target). This modified circuit is then stored in a new variable called "ansatz_isa". Effectively the ansatz circuit is modified to be transpiled (source code translated from one language to another) and stored in a new variable.**

ansatz_isa.draw(output="mpl", idle_wires=False, style="iqp")

**This draws the circuit which we can see in comparison to the original one:**




hamiltonian_isa = hamiltonian.apply_layout(layout=ansatz_isa.layout)

**The Hamiltonian is then transformed to be compatible with the quantum hardware.**

def cost_func(params, ansatz, hamiltonian, estimator):

"""Return estimate of energy from estimator

Parameters:

params (ndarray): Array of ansatz parameters

ansatz (QuantumCircuit): Parameterized ansatz circuit

hamiltonian (SparsePauliOp): Operator representation of Hamiltonian

estimator (EstimatorV2): Estimator primitive instance

cost_history_dict: Dictionary for storing intermediate results

Returns:

float: Energy estimate

```
"""
pub = (ansatz, [hamiltonian], [params])
result = estimator.run(pubs=[pub]).result()
energy = result[0].data.evs[0]

cost_history_dict["iters"] += 1
cost_history_dict["prev_vector"] = params
cost_history_dict["cost_history"].append(energy)
print(f"Iters. done: {cost_history_dict['iters']} [Current cost: {energy}]")

return energy
```

1. **Input: The function starts by accepting four arguments:**
    - **`params`: This is a NumPy array named `params` containing the current values for the parameters of the quantum circuit.**
    - **`ansatz`: This is a `QuantumCircuit` object named `ansatz` representing the specific quantum circuit used in the VQE process.**
    - **`hamiltonian`: This is a `SparsePauliOp` object named `hamiltonian`. It represents the mathematical equation describing the system's energy landscape (the Hamiltonian).**
    - **`estimator`: This is an object named `estimator` of type `EstimatorV2`. This object is used to estimate the actual energy of the system on the quantum computer.**
2. **Preparing the Estimator: The function creates a variable named `pub`. This variable becomes a package of information for the estimator. It's a tuple containing three elements:**
    - **The `ansatz` circuit itself.**
    - **A list containing only the `hamiltonian` (wrapped in square brackets). This specifies what observable we want to measure (the Hamiltonian's expectation value).**
    - **A list containing only the current `params` (wrapped in square brackets). This likely tells the estimator which parameters of the circuit to use during the measurement.**
3. **Running the Estimator: The function calls the `estimator.run(pubs=[pub])` method. This essentially sends the `pub` package to the estimator. The estimator interacts with the quantum**

**hardware (or simulator) based on the information in `pub`. It performs a measurement to estimate the system's energy according to the provided circuit, Hamiltonian, and parameter values. The result of this estimation is stored in a variable named `result`.**

4. **Extracting the Energy Estimate: The function then extracts the estimated energy value from the `result` obtained from the estimator. It accesses `result[0].data.evs[0]`. Here's a breakdown of this access:**
    - **`result[0]`: This likely refers to the first element in the output from the estimator, which might contain multiple measurements.**
    - **`.data`: This accesses the actual data obtained from the measurement.**
    - **`.evs[0]`: This extracts the first eigenvalue from the data. In this case, this eigenvalue represents the estimated energy of the system based on the current circuit parameters.**
5. **Printing and Returning: There are commented-out lines that seem intended for storing information about the optimization history in a dictionary.**

**Finally, the function:**

- **Prints a message showing the current iteration number and the estimated energy value using the `params` and the extracted energy value.**
- **Returns the estimated energy, which becomes the cost value used by the VQE optimization algorithm to guide its search for the ground state.**

```
cost_history_dict = {
    "prev_vector": None,
    "iters": 0,
    "cost_history": [],
}
```

**Note that, in addition to the array of optimization parameters that must be the first argument, we use additional arguments to pass the terms needed in the cost function, such as the `cost_history_dict`**

**. This dictionary stores the current vector at each iteration, for example in case you need to restart the routine due to failure, and also returns the current iteration number and average time per iteration.**

**We can now use a classical optimizer of our choice to minimize the cost function. Here, we use the COBYLA routine from SciPy through the `minimize` function. Note that when running on real quantum hardware, the choice of optimizer is important, as not all optimizers handle noisy cost function landscapes equally well.**
**To begin the routine, specify a random initial set of parameters:**

```
x0 = 2 * np.pi * np.random.random(num_params)
```

**the code creates an array where each element is a random value between 0 and 2 * pi (6.28318) this is the initial set of parameters where the number of parameters is determined by the input into the method (num_params)**

```
with Session(backend=backend) as session:
    estimator = Estimator(session=session)
    estimator.options.default_shots = 10000

    res = minimize(
        cost_func,
        x0,
        args=(ansatz_isa, hamiltonian_isa, estimator),
        method="cobyla",
    )
```

1. **Session Management:**
    - `with Session(backend=backend) as session:` **creates a session for interacting with the specified quantum backend. This ensures proper resource management and allows for concurrent job execution.**
2. **Estimator Creation:**
    - `estimator = Estimator(session=session)` **creates an estimator object, which is used to estimate the expectation value of an observable (in this case, the Hamiltonian). The estimator is tied to the created session.**
    - `estimator.options.default_shots = 10000` **sets the default number of shots (measurements) to be used for each estimation.**
3. **Optimization:**

- **`minimize` is a function from the `scipy.optimize` module used for optimization.**
- **`cost_func`: This is a custom function defined elsewhere (likely as shown in previous explanations) that calculates the energy of the system given the current circuit parameters.**
- **`x0`: This is the initial guess for the circuit parameters (usually randomly generated).**
- **`args=(ansatz_isa, hamiltonian_isa, estimator)`: These are additional arguments passed to the `cost_func`. They include the transpiled ansatz, the transpiled Hamiltonian, and the estimator object.**
- **`method="cobyla"`: Specifies the optimization method to be used, in this case, the COBYLA algorithm.**

**Overall Purpose:**

**This code block establishes a session with a quantum backend, creates an estimator for calculating energy estimates, and sets up the optimization process using the `minimize` function. The goal is to find the optimal set of parameters for the `ansatz` circuit that minimizes the energy as defined by the `hamiltonian`.**

**Key Points:**

- **The `with Session()` block ensures proper resource management.**
- **The `Estimator` is used for calculating energy estimates.**
- **The `minimize` function is used to optimize the circuit parameters.**
- **The `cost_func` provides the objective function for the optimization.**

res

**`res` is the output of the optimization process, specifically from the `scipy.optimize.minimize` function. It's a standard Python dictionary-like object that contains information about the optimization results.**

message: Optimization terminated successfully.
 success: True
  status: 1
    fun: -0.634701203143904
      x: [ 2.581e+00  4.153e-01 ...  1.070e+00  3.123e+00]

nfev: 146
maxcv: 0.0

- **message: This indicates the overall status of the optimization process. In this case, it says "Optimization terminated successfully".**
- **success: A boolean flag indicating whether the optimization was successful. In this case, it's `True`.**
- **status: An integer representing the status of the optimization. Different values have different meanings, but in this case, `1` typically indicates successful termination.**
- **fun: The value of the objective function (in this case, the energy) at the optimal parameters found. It's negative, which is common in quantum chemistry calculations as lower energy states are more stable.**
- **x: The optimal parameters found for the ansatz circuit. This is the primary result of the optimization process and will be used to construct the final quantum circuit.**
- **nfev: The number of function evaluations (calls to the `cost_func`) performed during the optimization. This gives an indication of the computational effort required.**
- **maxcv: The maximum constraint violation. This is relevant when dealing with constrained optimization problems, but in this case, it's 0, indicating no constraints were violated.**

**In essence, this output tells you that the optimization was successful, it found a set of parameters (x) that minimize the energy (fun), and it took 146 attempts to find this solution.**

**If the procedure terminates correctly, then the `prev_vector` and `iters` values in our `cost_history_dict` dictionary should be equal to the solution vector and total number of function evaluations, respectively. This is easy to verify:**

```
all(cost_history_dict["prev_vector"] == res.x)
```

**`cost_history_dict["prev_vector"]`: This accesses the "prev_vector" key within the `cost_history_dict` dictionary. This likely stores the previous set of parameters used in the optimization process.**

**`res.x`: This accesses the optimal parameters found during the optimization process, as stored in the `res` object.**

**`all(...)`: This function checks if all elements in the two compared arrays are equal.**

cost_history_dict["iters"] == res.nfev

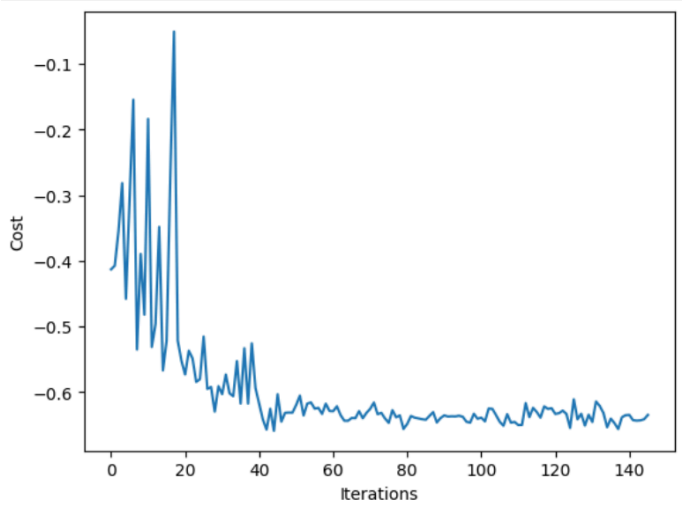
**`cost_history_dict["iters"]` represents the number of iterations the cost function has been evaluated, as tracked manually within your code.**

**`res.nfev` represents the number of function evaluations performed by the optimizer.**

**The fact that these two values are not equal indicates a discrepancy between the manual iteration tracking and the optimizer's internal counter.**

**We can also now view the progress toward convergence as monitored by the cost history at each iteration:**

```
fig, ax = plt.subplots()
ax.plot(range(cost_history_dict["iters"]), cost_history_dict["cost_history"])
ax.set_xlabel("Iterations")
ax.set_ylabel("Cost")
plt.draw()
```




**Overview:**

**The graph depicts the evolution of the cost function over successive iterations of an optimization process. The cost function, typically representing energy in quantum systems, is minimized to find optimal parameters for a quantum circuit.**

**Key Observations:**

- **Initial Fluctuations: The cost function exhibits significant fluctuations in the early iterations. This is often observed in optimization algorithms as they explore the solution space.**
- **Rapid Descent: After the initial chaotic phase, the cost function rapidly decreases, indicating efficient convergence towards a lower energy state.**
- **Oscillations and Stabilization: The curve shows oscillations around a certain value as the optimization process matures. This suggests the algorithm is refining its solution within a local minimum.**
- **Convergence: Eventually, the curve flattens out, implying that the optimization has converged to a stable solution. Further iterations lead to minimal changes in the cost function.**

**Implications:**

- **Algorithm Efficiency: The rapid initial descent suggests an efficient optimization algorithm capable of quickly exploring the solution space.**
- **Local Minima: The presence of oscillations indicates that the algorithm might have encountered local minima. These are points where the cost function decreases, but not to the global minimum.**
- **Convergence: The eventual flattening of the curve confirms that the optimization process has successfully converged to a solution.**