7

VQE: Variational Quantum Eigensolver

From so simple a beginning endless forms most beautiful and most wonderful have been, and are being, evolved.

— Charles Darwin

In the previous chapters of this part of the book, we have studied how quantum algorithms can help us solve combinatorial optimization problems, but there are many other important types of optimization problems out there! This chapter will broaden the scope of our optimization methods to cover more general settings, including applications in fields such as chemistry and physics.

We will achieve this by studying the famous **Variational Quantum Eigensolver** (**VQE**) algorithm, which can be seen as a generalization of the Quantum Approximate Optimization Algorithm that we studied back in *Chapter 5*, *QAOA: Quantum Approximate Optimization Algorithm*. Actually, it would be more precise to say that we can see QAOA as a particular case of VQE; in fact, VQE was introduced earlier than QAOA in a now famous paper by Peruzzo et al. [52].

We shall begin by expanding our knowledge of Hamiltonians and by better understanding how to estimate their expectation values with quantum computers. That will allow us to define VQE in all its glory and to appreciate both the simplicity of its formulation and its wide applicability for finding the ground state of different types of Hamiltonians.

We will then show how to use VQE with both Qiskit and PennyLane using examples from the field of chemistry. We will also show how to study the influence of errors on the algorithm by running simulations of noisy quantum computers, and we will even discuss some techniques to mitigate the adverse effect of readout errors.

After reading this chapter, you will know both the theoretical foundations of VQE and how to use it in a wide variety of practical situations, on simulators and on actual quantum computers.

The topics that we will cover in this chapter are as follows:

- Hamiltonians, observables, and their expectation values
- Introducing the Variational Quantum Eigensolver
- Using VQE with Qiskit
- Using VQE with PennyLane

We have quite a lot to learn and, in fact, endless forms most beautiful to discover. So, let's not waste time and get started right away!

7.1 Hamiltonians, observables, and their expectation values

So far, we've found in Hamiltonians a way to encode combinatorial optimization problems. As you surely remember, in these optimization problems, we start with a function f that assigns real numbers to binary strings of a certain length n, and we seek to find a binary string x with minimum cost $f(x)$. In order to do that with quantum algorithms, we define a Hamiltonian H_f such that

$$\langle x| H_f |x\rangle = f(x)$$

holds for every binary string x of length n. Then, we can solve our original problem by finding a ground state of H_f (that is, a state $|\psi\rangle$ such that the expectation value $\langle\psi| H_f |\psi\rangle$ is minimum).

This was just a very quick summary of *Chapter 3*, *Working with Quadratic Unconstrained Binary Optimization Problems*. When you read that chapter, you may have noticed that the Hamiltonian associated to f has an additional, very remarkable property. We have mentioned this a couple of times already, but it is worth remembering that, for every computational basis state $|x\rangle$, it holds that

$$H_f |x\rangle = f(x) |x\rangle .$$

This means that each $|x\rangle$ is an eigenvector of H_f with associated eigenvalue $f(x)$ (if you do not remember what eigenvectors and eigenvalues are, check *Appendix B*, *Installing the Tools*, for all the relevant definitions and concepts). In fact, this is easy to see because we have always used Hamiltonians that are sums of tensor products of Z matrices, which are clearly diagonal. But tensor products of diagonal matrices are diagonal matrices themselves, and sums of diagonal matrices are still diagonal. Thus, since these Hamiltonians are diagonal, the computational basis states are their eigenvectors.

What is more, if we have a state $|\psi\rangle$, we can always write it as a linear combination of the computational basis states. In fact, it holds that

$$|\psi\rangle = \sum_x \alpha_x |x\rangle ,$$

where the sum is over all the computational basis states $|x\rangle$ and $\alpha_x = \langle x|\psi\rangle$. This is easy to check, because

$$\langle x|\psi\rangle = \langle x| \sum_y \alpha_y |y\rangle = \sum_y \alpha_y \langle x|y\rangle = \alpha_x .$$

The last identity follows from the fact that $\langle x|y\rangle$ is 1 if $x = y$ and 0 otherwise (remember that the computational basis is an orthonormal basis).

Then, the expectation value of H_f in the state $|\psi\rangle$ can be computed as

$$\langle\psi| H_f |\psi\rangle = \sum_y \alpha_y^* \langle y| H_f \sum_x \alpha_x |x\rangle = \sum_{x,y} \alpha_y^*\alpha_x \langle y| H_f |x\rangle = \sum_{x,y} \alpha_y^*\alpha_x f(x) \langle y|x\rangle$$
$$= \sum_x \alpha_x^*\alpha_x f(x) = \sum_x |\alpha_x|^2 f(x).$$

Moreover, we know that $|\alpha_x|^2 = |\langle x|\psi\rangle|^2$ is the probability of obtaining $|x\rangle$ when measuring $|\psi\rangle$ in the computational basis; in this way, the expectation value matches the statistical expected value of the measurement. As you surely remember, this is exactly the fact that we used back in *Chapter 5*, *QAOA: Quantum Approximate Optimization Algorithm*, to estimate the value of the cost function when running QAOA circuits in a quantum computer.

These properties may seem dependent on the particular form of the Hamiltonians that we have been using. But, in fact, they are very general results, and we will use them extensively in our study of the VQE algorithm. But before we get to that, we will need to introduce the general notion of "observable", which is precisely the topic of the next subsection.

7.1.1 Observables

Up until this point, we have only considered measurements in the computational basis. This has worked well enough for our purposes, but, in doing so, we've ignored some details about how measurements are truly understood and described in quantum mechanics. We are now going to fill that gap.

We encourage you to go slowly through this section. Take your time and maybe prepare yourself a good cup of your favourite hot beverage. The ideas presented here may seem a little bit strange at first, but you will soon realize that they fit nicely with what we have been doing so far.

In quantum mechanics, any physical magnitude that you can measure — also known as a **(physical) observable** — is represented by a Hermitian operator. In case you don't remember, these are linear operators A that are equal to their adjoints (their conjugate transposes), that is, they satisfy $A^{\dagger} = A$.

To learn more...

You may remember how in *Chapter 3*, *Working with Quadratic Unconstrained Binary Optimization Problems*, we worked extensively with Hamiltonians. These, in general, are Hermitian operators that are, indeed, associated with an observable magnitude. That magnitude is none other than the energy of the system!

The nice thing about Hermitian operators is that, for them, one can always find an orthonormal basis of eigenvectors with real eigenvalues (please, check *Appendix B*, *Basic Linear Algebra*, if you need to review these notions). This means that there exist real numbers λ_j, $j = 1, \ldots, l$, all of them different, and states $\left|\lambda_j^k\right\rangle$, where $j = 1, \ldots, l$ and $k = 1, \ldots, r_j$, such that the states $\{\left|\lambda_j^k\right\rangle\}_{j,k}$ form an orthonormal basis and

$$A\left|\lambda_j^k\right\rangle = \lambda_j\left|\lambda_j^k\right\rangle,$$

for every $j = 1, \ldots, l$ and for every $k = 1, \ldots, r_j$.

Here, we are considering the possibility of having several eigenvectors $\left|\lambda_j^k\right\rangle$ associated with the same eigenvalue λ_j, hence the use of the superindices $k = 1, \ldots, r_j$, where r_j is the number of eigenvectors associated with the λ_j^k eigenvalue. If all the eigenvalues are different (a quite common case), then we will have $r_j = 1$ for every j and we can simply drop the k superindices.

What is the connection of these Hermitian operators with physical measurements? Let's consider an observable represented by a Hermitian operator A, and also an orthonormal basis of eigenvectors $\{\left|\lambda_j^k\right\rangle\}_{j,k}$ such that $A\left|\lambda_j^k\right\rangle = \lambda_j\left|\lambda_j^k\right\rangle$. This representation must be chosen to take the following into account:

- The possible outcomes of the measurement of the observable must be represented by the different eigenvalues λ_j

- The probability that a state $|\psi\rangle$ will, upon measurement, yield λ_j must be $\sum_k \left|\left\langle \lambda_j^k \middle| \psi \right\rangle\right|^2$

All of this is axiomatic. It is a fact of life that any physical observable can be represented by a Hermitian operator in such a way that those are requirements are satisfied. Moreover, it is a postulate of quantum mechanics that if the measurement returns the result associated to an eigenvalue λ_j, the state of the system will then become the normalized projection of $|\psi\rangle$ onto the space of eigenvectors with eigenvalue λ_j. This means that if we measure a state in a superposition such as

$$\sum_{j,k} \alpha_j^k \left|\lambda_j^k\right\rangle$$

and we obtain λ_j as the result, then the new state will be

$$\frac{\sum_k \alpha_j^k \left|\lambda_j^k\right\rangle}{\sqrt{\sum_k \left|\alpha_j^k\right|^2}}.$$

This is what we call the *collapse* of the original state and it is exactly the same phenomenon that we considered when studied measurements in the computational basis back in *Chapter 1, Foundations of Quantum Computing*.

The word "observable" is often used for both physical observables and for any Hermitian operators that represent them. Thus, we may refer to a Hermitian operator itself as an observable. To avoid confusions, we will usually not omit the "physical" adjective when referring to physical observables.

As a simple example, whenever we measure in the computational basis, we are indeed measuring some physical observable, and this physical observable can, of course, be represented by a Hermitian operator. This is, in a certain sense, the simplest observable in quantum computing and it is natural that it arises as a particular case of this, more general theory of quantum measurements.

The coordinated matrix of this measurement operator with respect to the computational basis could be the diagonal matrix

$$\begin{pmatrix} 0 & & & \\ & 1 & & \\ & & \ddots & \\ & & & 2^n - 1 \end{pmatrix}.$$

Exercise 7.1

Prove that, indeed, the previous matrix is the coordinate matrix on the computational basis of a Hermitian operator that represents a measurement in the computational basis.

When we measure a single qubit in the computational basis, the coordinate matrix with respect to the computational basis of the associated Hermitian operator could well be either of

$$N = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}, \qquad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

Yes, that last matrix was the unmistakable Pauli Z matrix. Both of these operators represent the same observable; they only differ in the eigenvalues that they associate to the distinct possible outcomes. The first operator associates the eigenvalues 0 and 1 to the qubit's value being 0 and 1 respectively, while the second observable associates the eigenvalues 1 and -1 to these outcomes.

Important note

Measurements in quantum mechanics are represented by Hermitian operators, which we refer to as observables. One possible operator corresponding to measuring a qubit in the computational basis can be the Pauli Z matrix.

Now that we know what an observable is, we can study what its **expectation value** is and how it can be computed. The expectation value of any observable under a state $|\psi\rangle$ can be defined as

$$\langle A\rangle_\psi = \sum_{j,k} \left|\left\langle \lambda_j^k \middle| \psi \right\rangle\right|^2 \lambda_j,$$

which is a natural definition that agrees with the statistical expected value of the results obtained when we measure $|\psi\rangle$ according to A. As intuitive as this expression may be, we can further simplify it as follows:

$$\begin{aligned}\langle A\rangle_\psi &= \sum_{j,k} \left|\left\langle \lambda_j^k \middle| \psi \right\rangle\right|^2 \lambda_j = \sum_{j,k} \left\langle \psi \middle| \lambda_j^k \right\rangle \left\langle \lambda_j^k \middle| \psi \right\rangle \lambda_j = \sum_{j,k} \left\langle \lambda_j^k \middle| \psi \right\rangle \left\langle \psi \middle| \lambda_j^k \right\rangle \lambda_j \\ &= \sum_{j,k} \left\langle \lambda_j^k \middle| \psi \right\rangle \langle \psi | A \left| \lambda_j^k \right\rangle = \langle \psi | A \sum_{j,k} \left\langle \lambda_j^k \middle| \psi \right\rangle \left| \lambda_j^k \right\rangle = \langle \psi | A | \psi \rangle .\end{aligned}$$

Notice that we have used the fact that $A\left|\lambda_j^k\right\rangle = \lambda_j \left|\lambda_j^k\right\rangle$ and that $|\psi\rangle = \sum_{j,k} \left\langle \lambda_j^k \middle| \psi \right\rangle \left|\lambda_j^k\right\rangle$. This latter identity follows from the fact that $\{\left|\lambda_j^k\right\rangle\}_{j,k}$ is an orthonormal basis and, in fact, it can be proved in exactly the same way we did for the computational basis at the beginning of this section.

This expression for the expectation value agrees with our previous work in *Chapter 3, Working with Quadratic Unconstrained Binary Optimization Problems.*

Important note

The expectation value of any Hermitian operator (observable) A is given by

$$\langle A\rangle_\psi = \sum_{j,k} \left|\left\langle \lambda_j^k \middle| \psi \right\rangle\right|^2 \lambda_j = \langle \psi | A | \psi \rangle .$$

Notice that, from the very definition of the expectation value of an observable, we can easily derive the variational principle. This principle states, as you may recall from *Chapter 3, Working with Quadratic Unconstrained Binary Optimization Problems*, that the smallest expectation value of an observable A is always achieved at an eigenvector of that observable. To prove it, suppose that λ_0 is minimal among all the eigenvalues of A. Then, for any state ψ it holds that

$$\langle A \rangle_\psi = \sum_{j,k} \left| \left\langle \lambda_j^k \middle| \psi \right\rangle \right|^2 \lambda_j \geq \sum_{j,k} \left| \left\langle \lambda_j^k \middle| \psi \right\rangle \right|^2 \lambda_0 = \lambda_0,$$

where the last equality follows from the fact that $\sum_{j,k} \left| \left\langle \lambda_j^k \middle| \psi \right\rangle \right|^2 = 1$, since the sum of the probabilities of all the outcomes must add up to 1.

If we now take any eigenvector $\left|\lambda_0^k\right\rangle$ associated to λ_0, its expectation value will be

$$\left\langle \lambda_0^k \middle| A \middle| \lambda_0^k \right\rangle = \lambda_0 \left\langle \lambda_0^k \middle| \lambda_0^k \right\rangle = \lambda_0,$$

proving that the minimum expectation value is indeed achieved at an eigenvector of A. Obviously, if there were several orthogonal eigenvectors associated to λ_0, any normalized linear combination of them would also be a ground state of A.

In this subsection, we have studied the mathematical expression for the expectation of any observable. But we don't yet know how to estimate these expectation values with quantum computers. How could we do that? Just keep reading, because we will be exploring it in the next subsection.

7.1.2 Estimating the expectation values of observables

In the context of the VQE algorithm, we will need to estimate the expectation value of a general observable A. That is, we will no longer assume that A is diagonal, as we have done in all the previous chapters. For this reason, we will need to develop a new method for estimating the expectation value $\langle \psi | A | \psi \rangle$.

We know that, for a given state $|\psi\rangle$, the expectation value of A can be computed by

$$\sum_{j,k} \left| \left\langle \lambda_j^k \middle| \psi \right\rangle \right|^2 \lambda_j.$$

Thus, if we knew the eigenvalues λ_j and the eigenvectors $\{\left|\lambda_j^k\right\rangle\}_{j,k}$ of A, we could try to compute $\left|\left\langle\lambda_j^k\middle|\psi\right\rangle\right|^2$ and, hence, the expectation value of A. However, this is information that we usually don't know. In fact, the purpose of VQE is, precisely, finding certain eigenvalues and eigenvectors of a Hamiltonian! Moreover, the number of eigenvectors grows exponentially with the number of qubits of our system, so, even if we knew them, computing expectation values in this way might be very computationally expensive.

Thus, we need to take an indirect route. For this, we will use the fact that we can always express an observable A on n qubits as a linear combination of tensor products of Pauli matrices (see, for example, *Chapter 7* on the famous lecture notes by John Preskill [53]). Actually, A will be, in most cases, given to us in such a form, in the same way that the Hamiltonians of our combinatorial optimization problems were always expressed as sums of tensor products of Z matrices.

So, consider, for example, that we are given an observable

$$A = \frac{1}{2}Z \otimes I \otimes X - 3I \otimes Y \otimes Y + 2Z \otimes X \otimes Z.$$

Notice that, thanks to linearity,

$$\begin{aligned}
\langle\psi| A |\psi\rangle &= \langle\psi| \left(\frac{1}{2}Z \otimes I \otimes X - 3I \otimes Y \otimes Y + 2Z \otimes X \otimes Z\right) |\psi\rangle \\
&= \langle\psi| \left(\frac{1}{2}(Z \otimes I \otimes X)|\psi\rangle - 3(I \otimes Y \otimes Y)|\psi\rangle + 2(Z \otimes X \otimes Z)|\psi\rangle\right) \\
&= \frac{1}{2}\langle\psi|(Z \otimes I \otimes X)|\psi\rangle - 3\langle\psi|(I \otimes Y \otimes Y)|\psi\rangle + 2\langle\psi|(Z \otimes X \otimes Z)|\psi\rangle .
\end{aligned}$$

Then, in order to compute the expectation value of A, we can compute the expectation values of $Z \otimes I \otimes X$, $I \otimes Y \otimes Y$, and $Z \otimes X \otimes Z$ and combine their results. But wait a minute! Isn't that even more complicated? After all, we would need to compute three expectation values instead of just one, right?

The key observation here lies in the fact that, while we may not know the eigenvalues and eigenvectors of A in advance, we can very easily obtain those of $Z \otimes I \otimes X$ or any other tensor product of Pauli matrices. It is so easy, in fact, that you will now learn how to do it yourself in the following two exercises.

Exercise 7.2

Suppose that $\left|\lambda_j\right\rangle$ is an eigenvector of A_j with associated eigenvalue λ_j for $j = 1, \ldots, n$. Prove that $|\lambda_1\rangle \otimes \cdots \otimes |\lambda_n\rangle$ is an eigenvector of $A_1 \otimes \cdots \otimes A_n$ with associated eigenvalue $\lambda_1 \cdot \ldots \cdot \lambda_n$.

Exercise 7.3

Prove that:

1. The eigenvectors of Z are $|0\rangle$ (with associated eigenvalue 1) and $|1\rangle$ (with associated eigenvalue -1).
2. The eigenvectors of X are $|+\rangle$ (with associated eigenvalue 1) and $|-\rangle$ (with associated eigenvalue -1).
3. The eigenvectors of Y are $\left(1/\sqrt{2}\right)(|0\rangle + i\,|1\rangle)$ (with associated eigenvalue 1) and $\left(1/\sqrt{2}\right)(|0\rangle - i\,|1\rangle)$ (with associated eigenvalue -1).
4. Any non-null state is an eigenvector of I with associated eigenvalue 1.

Using the results in these exercises, we can readily deduce that $|0\rangle|+\rangle|0\rangle$, $|0\rangle|-\rangle|1\rangle$, $|1\rangle|+\rangle|1\rangle$, and $|1\rangle|-\rangle|0\rangle$ are eigenvectors of $Z \otimes X \otimes Z$ with eigenvalue 1 and that $|0\rangle|+\rangle|1\rangle$, $|0\rangle|-\rangle|0\rangle$, $|1\rangle|+\rangle|0\rangle$, and $|1\rangle|-\rangle|1\rangle$ are eigenvectors of $Z \otimes X \otimes Z$ with eigenvalue -1. All these states together form an orthonormal basis of eigenvectors of $Z \otimes X \otimes Z$, as you can easily check if you compute their inner products.

Exercise 7.4

Find orthonormal bases of eigenvectors for $Z \otimes I \otimes X$ and $I \otimes Y \otimes Y$. Compute their associated eigenvalues.

So, now we know how to obtain the eigenvalues and eigenvectors of any tensor product of Pauli matrices. How can we use this to estimate their expectation values? Remember that, given a Hermitian matrix A, we can compute $\langle\psi| A |\psi\rangle$ by

$$\sum_{j,k} \left|\left\langle \lambda_j^k \middle| \psi \right\rangle\right|^2 \lambda_j,$$

where the eigenvalues of A are λ_j and the associated eigenvectors are $\{\left|\lambda_j^k\right\rangle\}_{j,k}$. In our case, we only have two eigenvalues: 1 and -1. So, if we are able to estimate the values $\left|\left\langle \lambda_j^k \middle| \psi \right\rangle\right|^2$, we will have all the ingredients needed to "cook" our expectation values.

A priori, trying to get the values $\left|\left\langle \lambda_j^k \middle| \psi \right\rangle\right|^2$ out of a quantum computer can seem like a difficult task. For example, you may wonder whether it will be necessary to perform some weird fancy measurements on our quantum device in order to get these probabilities! Well, it turns out that we can easily estimate them on any quantum computer using ordinary measurements in the computational basis and a bunch of quantum gates. So, don't worry. If you've just bought yourself a flashy quantum computer, there's no need for a hardware upgrade just yet.

In any case, how can we actually estimate these $\left|\left\langle \lambda_j^k \middle| \psi \right\rangle\right|^2$ values with the tools that we have? Let's first work with an example.

Let's consider the observable $Z\otimes X\otimes Z$. We have previously in this section obtained its eigenvectors, so let's focus on one of them: $|0\rangle |+\rangle |0\rangle$. If we wanted to compute $|(\langle 0| \langle +| \langle 0|) |\psi\rangle|^2$, where $|\psi\rangle$ is a certain 3-qubit state, we could just notice that

$$|0\rangle |+\rangle |0\rangle = (I \otimes H \otimes I) |0\rangle |0\rangle |0\rangle$$

and hence

$$\begin{aligned}\langle 0| \langle +| \langle 0| = (|0\rangle |+\rangle |0\rangle)^\dagger = ((I \otimes H \otimes I) |0\rangle |0\rangle |0\rangle)^\dagger = \langle 0| \langle 0| \langle 0| (I \otimes H \otimes I)^\dagger \\ = \langle 0| \langle 0| \langle 0| (I \otimes H \otimes I),\end{aligned}$$

where we have used the fact that I and H are self-adjoint, and hence so is $I \otimes H \otimes I$. Keep in mind, however, that we will still write daggers throughout this example whenever we mean to consider the adjoint of $I \otimes H \otimes I$ — even if it still represents the same operator.

From this, it follows directly that

$$|(\langle 0| \langle +| \langle 0|) |\psi\rangle|^2 = \left|\langle 0| \langle 0| \langle 0| (I \otimes H \otimes I)^\dagger |\psi\rangle\right|^2.$$

But for any state $|\varphi\rangle$, we know that $|(\langle 0| \langle 0| \langle 0|) |\varphi\rangle|^2$ is the probability of obtaining $|0\rangle |0\rangle |0\rangle$ when measuring it in the computational basis. As a consequence, we can estimate the value $|(\langle 0| \langle +| \langle 0|) |\psi\rangle|^2$ by repeatedly preparing the state $(I \otimes H \otimes I) |\psi\rangle = (I \otimes H \otimes I)^\dagger |\psi\rangle$, measuring it in the computational basis, and then computing the relative frequency of $|0\rangle |0\rangle |0\rangle$.

And this is not the only eigenvector for which this works. It turns out that for each and every eigenvector $|\lambda_A\rangle$ of $Z \otimes X \otimes Z$, there is a unique state in the computational basis $|\lambda_C\rangle$ such that

$$|\lambda_A\rangle = (I \otimes H \otimes I) |\lambda_C\rangle.$$

Actually, the correspondence is bijective: for every state in the computational basis $|\lambda_C\rangle$, there is also a unique eigenvector $|\lambda_A\rangle$ of $Z \otimes X \otimes Z$ such that $|\lambda_C\rangle = (I \otimes H \otimes I)^\dagger |\lambda_A\rangle$, where we have used the fact that, for unitary operators, $U^\dagger = U^{-1}$. For example,

$$|1\rangle |-\rangle |1\rangle = (I \otimes H \otimes I) |1\rangle |1\rangle |1\rangle, \qquad |1\rangle |1\rangle |1\rangle = (I \otimes H \otimes I)^\dagger |1\rangle |-\rangle |1\rangle.$$

This is the reason why we call $I \otimes H \otimes I$ the **change of basis operator** between the computational basis and the basis of eigenvectors of $Z \otimes X \otimes Z$.

In this way, if we want to estimate the probabilities $\left|\left\langle \lambda_j^k \middle| \psi \right\rangle\right|^2$ when the states $\left|\lambda_j^k\right\rangle$ happen to be the eigenvectors of $Z \otimes X \otimes Z$, we just need to prepare $(I \otimes H \otimes I)^\dagger |\psi\rangle$ and measure it in the computational basis. Then, given any eigenvector $|\lambda_A\rangle$ of $Z \otimes X \otimes Z$, the probability $|\langle \lambda_A|\psi\rangle|^2$ can be estimated by the relative frequency of the measurement outcome associated to the eigenstate $|\lambda_C\rangle = (I \otimes H \otimes I)^\dagger |\lambda_A\rangle$ in the computational basis. That's because

$$\langle\lambda_C| \left((I \otimes H \otimes I)^\dagger |\psi\rangle\right) = \langle\lambda_A| \left((I \otimes H \otimes I)(I \otimes H \otimes I)^\dagger |\psi\rangle\right) = \langle\lambda_A|\psi\rangle ,$$

where we have used the fact that, for any operator L and any states $|\alpha\rangle$ and $|\beta\rangle$, if $|\beta\rangle = L|\alpha\rangle$, then $\langle\beta| = \langle\alpha| L^\dagger$, and $L^{\dagger\dagger} = L$.

As a final note, in this example, when we set out to compute the probabilities $|\langle\lambda_A|\psi\rangle|^2$, we don't have to run executions for each of the probabilities individually: we can compute them all simultaneously. All we have to do is measure $(I \otimes H \otimes I)^\dagger |\psi\rangle$ in the computational basis a bunch of times and then retrieve the relative frequency of every outcome. This works because $(I \otimes H \otimes I)^\dagger$ transforms all the eigenvectors of A into the states of the computational basis. Then, the probability $|\langle\lambda_A|\psi\rangle|^2$ will be the relative frequency of the outcome, in the computational basis, associated to $(I \otimes H \otimes I)^\dagger |\lambda_A\rangle$. Of course, the higher the number of preparations and measurements, the more accurate our estimates will be.

To learn more...

Notice the similarity of this kind of procedure with the standard measurement in the computational basis. When we measure $|\psi\rangle$ in the computational basis, we have probability $|\langle x|\psi\rangle|^2$ of obtaining the outcome associated to $|x\rangle$. If we were measuring an observable that had all the $\left|\lambda_j^k\right\rangle$ as eigenvectors with a distinct eigenvalue for each of them — this is an observable that's able to distinguish all the eigenvectors in the basis — we would have probability $\left|\left\langle\lambda_j^k\middle|\psi\right\rangle\right|^2$ of getting the outcome associated to $\left|\lambda_j^k\right\rangle$.

This is why we refer to the process of changing basis and, then, measuring in the computational basis, as performing a **measurement in the eigenvector basis** $\{\left|\lambda_j^k\right\rangle\}$ **of** A. It is exactly the same as if we had an observable that's able to measure and distinguish all the eigenvectors of A.

But wait, there's more! Our being able to change bases in this case is by no means a happy coincidence. It turns out that for every tensor product of Pauli matrices A, there is a simple change of basis matrix that defines a perfect correspondence between the states in the computational basis and the eigenvectors of A. Again, this can be readily verified, and we invite you to do it in the following two exercises.

Exercise 7.5

Since the computational basis is an eigenvector basis of Z, a change of basis operator of Z can be the identity I. Check that, in order to change from the computational basis to the basis of eigenvectors of the X, you can use the Hadamard matrix H, and that to change to the basis of eigenvectors of Y you can use SH.

Exercise 7.6

Prove that if U_1 and U_2 are the respective change of basis operators from the computational basis to the eigenvector basis of two observables A_1 and A_2, then $U_1 \otimes U_2$ is the change of basis operator from the computational basis to the eigenvector basis of $A_1 \otimes A_2$.

Putting everything together, we can easily deduce that, for instance, $I \otimes I \otimes H$ takes the eigenvectors of $Z \otimes I \otimes X$ to the computational basis and that $I \otimes (SH)^\dagger \otimes (SH)^\dagger$ takes the eigenvectors of $I \otimes Y \otimes Y$ to states in the computational basis as well.

Therefore, in order to estimate the expectation value $\langle\psi| (Z \otimes I \otimes X) |\psi\rangle$, we can use whatever circuit we need to prepare $|\psi\rangle$ followed by $(I \otimes I \otimes H)^\dagger = I \otimes I \otimes H$, then measure in the computational basis, and then get the probabilities as we have just discussed. In a similar way, to estimate $\langle\psi| (I \otimes Y \otimes Y) |\psi\rangle$, we will first prepare $|\psi\rangle$, then apply $I \otimes HS^\dagger \otimes HS^\dagger$ and, finally, measure in the computational basis.

Notice, by the way, how I and H are self-adjoint, so, when we took their adjoints, there was no observable (no pun intended) effect. That's not the case with SH, because $(SH)^\dagger = H^\dagger S^\dagger = HS^\dagger$.

To learn more…

For any Hermitian operator A, there is always a unitary transformation that takes any basis of eigenvectors of A to the computational basis, and vice versa. However, this transformation could very well be difficult to implement. In the case where A is a tensor product of Pauli matrices we have just proved that we can always obtain the transformation as the tensor product of very simple one-qubit operations.

Finally, after we have estimated the expectation value of every Pauli term in our observable (in our case, $Z \otimes I \otimes X$, $I \otimes Y \otimes Y$, and $Z \otimes X \otimes Z$) we can multiply them by the corresponding coefficients in the linear combination and add everything together to get the final result. And we are done!

Now you know how to estimate the expectation value of an observable by measuring in different bases. You can proudly say, as the famous internet meme goes, "All your base are belong to us." And, in fact, this was the last technical element that we needed in order to introduce VQE, something we will immediately do in the next section.

7.2 Introducing VQE

The goal of the **Variational Quantum Eigensolver** (**VQE**) is to find a ground state of a given Hamiltonian H_1. This Hamiltonian can describe, for instance, the energy of a certain physical or chemical process, and we will use some such examples in the following two sections, which will cover how to execute VQE with Qiskit and PennyLane. For the moment, however, we will keep everything abstract and focus on finding a state $|\psi\rangle$ such that $\langle\psi| H_1 |\psi\rangle$ is minimum. Note that in this section, we will be using H_1 to refer to the Hamiltonian so that it does not get confused with the Hadamard matrix that we will also be using in our computations.

To learn more…

VQE is by no means the only quantum algorithm that has been proposed to find the ground states of Hamiltonians. Some very promising options use a quantum subroutine known as **Quantum Phase Estimation** (**QPE**) (see, for instance, the excellent surveys by McArdle et al. [54] and by Cao et al. [55]). The main disadvantage of these approaches is that QPE uses the Quantum Fourier Transform that we studied in *Chapter 6*, *GAS: Grover Adaptative Search*, and, thus, requires quantum computers that are resilient to noise. An experimental demonstration of these limitations (and of the relative robustness of VQE) can be found, for instance, in the paper by O'Malley et al. [56]. For this reason, we will focus mainly on VQE and its applications, which seem to obtain better results with the NISQ computers that are available today.

The general structure of VQE is very similar to that of QAOA, which you surely remember from *Chapter 5*, *QAOA: Quantum Approximate Optimization Algorithm*: we prepare a parameterized quantum state, we measure it, we estimate its energy, and we change the parameters in order to minimize it; then, we repeat this process several times until some stopping criteria are met. The preparation and measurement of the state are done on the quantum computer, while the energy estimation and parameter minimization are handled by a classical computer.

The parametrized circuit, usually called **variational form** or **ansatz**, is usually chosen taking into account information from the problem domain. For instance, you could consider ansatzes that parametrize typical solutions to the kind of problem under study. We will show some examples of this in the last two sections of this same chapter. In any case, the ansatz is selected in advance and it is usually easy to implement on a quantum circuit.

Important note

In many applications, we can distinguish two parts in the creation of the parameterized state: the preparation of an initial state $\left|\psi_0\right\rangle$, that does not depend on any parameters, and then the variational form $V(\theta)$ itself, that obviously depends on θ. Thus, if we have $\left|\psi_0\right\rangle = U\left|0\right\rangle$, for some unitary transformation U implemented with some quantum gates, the ansatz gives us the state $V(\theta)U\left|0\right\rangle$. Notice, however, that we can always consider the whole operation $V(\theta)U$ as the ansatz and require the initial state to be $\left|0\right\rangle$. To simplify our notation, this is what we will usually do, although we will explicitly distinguish between initial state and ansatz in some practical examples that we will consider later in the chapter.

Algorithm 7.1 gives the pseudocode for VQE. Notice the similarities with Algorithm 5.1 from *Chapter 5*, *QAOA: Quantum Approximate Optimization Algorithm*.

Algorithm 7.1 (VQE).

```
Require: $H_1$ given as a linear combination of tensor products of Pauli matrices
  Choose a variational form (ansatz) $V(\theta)$
  Choose a starting set of values for $\theta$
  while the stopping criteria are not met do
      Prepare the state $\left|\psi(\theta)\right\rangle = V(\theta)\left|0\right\rangle$      ▷ This is done on the quantum computer!
      From the measurements of $\left|\psi(\theta)\right\rangle$ in different bases, estimate $\left\langle\psi(\theta)\right| H_1 \left|\psi(\theta)\right\rangle$
      Update $\theta$ according to the minimization algorithm
  Prepare the state $\left|\psi(\theta)\right\rangle = V(\theta)\left|0\right\rangle$      ▷ This is done on the quantum computer!
  From the measurements of $\left|\psi(\theta)\right\rangle$ in different bases, estimate $\left\langle\psi(\theta)\right| H_1 \left|\psi(\theta)\right\rangle$
```

Let's remark on a couple of things about this pseudocode. Notice that we require that H_1 be given as a linear combination of tensor products of Pauli matrices; this is so that we can use the techniques that we introduced in the previous section to estimate $\left\langle\psi\right| H_1 \left|\psi\right\rangle$. Of course, the more terms we have in the linear combination, the bigger the number of bases in which we may need to perform measurements. Nevertheless, in some cases, we may group several measurements together. For example, if we have terms such as $I \otimes X \otimes I \otimes X$, $I \otimes I \otimes X \otimes X$, and $I \otimes I \otimes X \otimes X$, we can use $I \otimes H \otimes H \otimes H$ as our change of basis matrix (be careful! This H is the Hadamard matrix, not the Hamiltonian!) because it works for the three terms at the same time — keep in mind that any orthonormal basis is an eigenvector basis for I, not just $\{|0\rangle, |1\rangle\}$. Obviously, another hyperparameter that will impact the execution time of VQE is the number of times that we measure $|\psi\rangle$ in each basis. The higher this number, the more precise the estimation, but also the higher the time needed to estimate $\langle\psi| H_1 |\psi\rangle$.

Notice also that the pseudocode of Algorithm 7.1 concludes by estimating $\langle\psi(\theta)| H_1 |\psi(\theta)\rangle$ for the last state $|\psi(\theta)\rangle$ found by the minimization algorithm. This is a quite common use case, for instance, if we want to determine the ground state energy for a particular system. However, you are not restricted to just that. At the end of the VQE execution, you also know the θ_0 parameters that were used to build the ground state, and you could use them to reconstruct $|\psi(\theta_0)\rangle = V(\theta_0)|0\rangle$. This state could then be used for other purposes, such as being sent into another quantum algorithm.

In fact, in the next subsection, we are going to explore one of such uses: the computation of additional **eigenstates** (another name for our old friends, the eigenvectors) of Hamiltonians. You should be excited!

7.2.1 Getting excited with VQE

As we have just explained, VQE is used to search for a ground state of a given Hamiltonian H. However, with a small modification, we can also use it to find **excited states**: eigenstates with higher energies. Let's explain how to achieve this.

Suppose that you have been given a Hamiltonian H and you have used VQE to find a ground state $|\psi_0\rangle = V(\theta_0)|0\rangle$ with energy λ_0. Then, we may consider the modified Hamiltonian

$$H' = H + C|\psi_0\rangle\langle\psi_0|,$$

where C is a positive real number.

Before we move on to detailing why H' can enable us to find excited states, let's explain what the term $|\psi_0\rangle\langle\psi_0|$ in that expression means. First of all, notice that this term represents a square matrix: it is the product of a column vector ($|\psi_0\rangle$) and a row vector ($\langle\psi_0|$) of the same length. Moreover, it is a Hermitian matrix, because

$$(\left|\psi_0\right\rangle\left\langle\psi_0\right|)^\dagger = \left\langle\psi_0\right|^\dagger \left|\psi_0\right\rangle^\dagger = \left|\psi_0\right\rangle\left\langle\psi_0\right|.$$

Then, H' is the sum of two Hermitian matrices and is, therefore, also Hermitian. And what is its expectation value? If we have a generic quantum state $\left|\psi\right\rangle$, then

$$\left\langle\psi\right| H' \left|\psi\right\rangle = \left\langle\psi\right| H \left|\psi\right\rangle + C\left\langle\psi|\psi_0\right\rangle\left\langle\psi_0|\psi\right\rangle = \left\langle\psi\right| H \left|\psi\right\rangle + C|\left\langle\psi_0|\psi\right\rangle|^2.$$

That is, the expectation value of H' in a state $\left|\psi\right\rangle$ is the expectation value of H plus a non-negative value that quantifies the overlap of $\left|\psi\right\rangle$ and $\left|\psi_0\right\rangle$. Hence we have two extreme cases for $C|\left\langle\psi_0|\psi\right\rangle|^2$. If $\left|\psi\right\rangle = \left|\psi_0\right\rangle$, this term will be C. If $\left|\psi\right\rangle$ and $\left|\psi_0\right\rangle$ are orthogonal, the term will be 0.

Thus, if we make C big enough, $\left|\psi_0\right\rangle$ will no longer be a ground state of H'. Let's prove this is in a more formal way. To this end, let $\lambda_0 \leq \lambda_1 \leq \ldots \leq \lambda_n$ be the eigenvalues of H associated to each eigenvector in an orthonormal eigenvector basis $\{\left|\lambda_j\right\rangle\}$ (since different eigenvectors may have the same eigenvalue, some of the energies may be repeated). As $\left|\psi_0\right\rangle$ is, by hypothesis, a ground state, we shall assume that $\left|\lambda_0\right\rangle = \left|\psi_0\right\rangle$. The states $\{\left|\lambda_j\right\rangle\}$ are also eigenvectors of H' because, on the one hand, if $j \neq 0$,

$$H'\left|\lambda_j\right\rangle = H\left|\lambda_j\right\rangle + C\left|\psi_0\right\rangle\left\langle\psi_0|\lambda_j\right\rangle = H\left|\lambda_j\right\rangle + C\left|\lambda_0\right\rangle\left\langle\lambda_0|\lambda_j\right\rangle = H\left|\lambda_j\right\rangle = \lambda_j\left|\lambda_j\right\rangle,$$

since $\left|\lambda_0\right\rangle$ and $\left|\lambda_j\right\rangle$ are orthogonal. On the other hand,

$$H'\left|\lambda_0\right\rangle = H\left|\lambda_0\right\rangle + C\left|\psi_0\right\rangle\left\langle\psi_0|\lambda_0\right\rangle = H\left|\lambda_0\right\rangle + C\left|\lambda_0\right\rangle\left\langle\lambda_0|\lambda_0\right\rangle = H\left|\lambda_0\right\rangle + C\left|\lambda_0\right\rangle = (\lambda_0 + C)\left|\lambda_0\right\rangle.$$

Thus, it follows that $\left\langle\lambda_j\right| H' \left|\lambda_j\right\rangle = \lambda_j$ when $j \neq 0$ and that $\left\langle\lambda_0\right| H' \left|\lambda_0\right\rangle = C + \lambda_0$. Hence, if $C > \lambda_1 - \lambda_0$, then $\left|\psi_0\right\rangle = \left|\lambda_0\right\rangle$ will no longer be a ground state of H', because the energy of $\left|\lambda_1\right\rangle$ will be lower than that of $\left|\psi_0\right\rangle$. Thanks to the variational principle, we know that the minimum energy is attained at an eigenvector of H', so $\left|\lambda_1\right\rangle$ must be a ground state of H'.

We can then use VQE to search for a ground state of H' and obtain the state $|\lambda_1\rangle$, as we intended.

To learn more...

Notice that it could be the case that $\lambda_1 = \lambda_0$. In that situation, $|\lambda_1\rangle$ would be another ground state of H. Otherwise, it will be the first excited state of H.

You may have also noticed that, even if the ground state is unique, the first excited eigenstate may not be so. This happens if and only if $|\lambda_2\rangle$ (and possibly other states in the basis) has the same energy as $|\lambda_1\rangle$, (that is, $\lambda_2 = \lambda_1$). In that case, any normalized linear combination of those eigenvectors will be a ground state of H'. Any of them will serve our purposes equally well.

Of course, once you obtain $|\lambda_1\rangle$, you can consider $H'' = H' + C' |\lambda_1\rangle \langle\lambda_1|$ and use VQE to search for $|\lambda_2\rangle$, and so on and so forth. Keep in mind that, in this process, we would have to pick the constants $C, C', \ldots$ properly — just to make sure that none of the eigenstates that we already know becomes a ground state again!

With this, our problem of finding eigenvectors of increasing energy is solved. Or is it?

There is just one little implementation detail that might be bothering you. In the previous section, we discussed how to estimate the expectation value of a Hamiltonian under the assumption that it was given as a sum of tensor products of Pauli matrices. However, the $|\psi_0\rangle \langle\psi_0|$ term is not of that form. In fact, we know $|\psi_0\rangle$ only as the result of applying VQE, so it is very likely that we will not know $|\psi_0\rangle$ explicitly; instead, we will have nothing more than some parameters θ_0 such that $V(\theta_0)|0\rangle = |\psi_0\rangle$. This, nonetheless, is enough to compute the expectation values that we need.

Let's step back a little bit and take a look at what we need to compute. At a given moment in the application of VQE, we have some parameters θ and we want to estimate the expectation value of $|\psi_0\rangle \langle\psi_0|$ with respect to $|\psi(\theta)\rangle = V(\theta)|0\rangle$. This quantity is

$$\langle\psi(\theta)|\psi_0\rangle \langle\psi_0|\psi(\theta)\rangle = |\langle\psi_0|\psi(\theta)\rangle|^2 = \left|\langle 0| V(\theta_0)^\dagger V(\theta) |0\rangle\right|^2.$$

But this is just the probability of obtaining $|0\rangle$ as the outcome of measuring $V(\theta_0)^\dagger V(\theta)|0\rangle$ in the computational basis! This is something that we can easily estimate because we can prepare $V(\theta_0)^\dagger V(\theta)|0\rangle$ by first applying our ansatz V, using θ as the parameters, to $|0\rangle$, and then applying the inverse of our ansatz, with parameters θ_0, to the resulting state. We will repeat this process several times, always measuring the resulting state $V(\theta_0)^\dagger V(\theta)|0\rangle$ in the computational basis and computing the relative frequency of the outcome $|0\rangle$. This is illustrated in *Figure 7.1*.

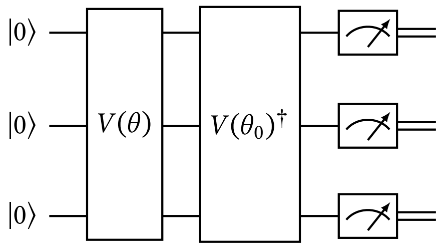

Figure 7.1: Circuit to compute $\left|\langle 0| V(\theta_0)^\dagger V(\theta)|0\rangle\right|^2$.

The only thing that may, at first, seem difficult with this method is to obtain the circuit for $V(\theta_0)^\dagger$. However, this is fairly easy. You just need to remember that every unitary gate is reversible. Thus, you can take the circuit for $V(\theta)$ and read the gates from right to left, reversing each one of them. As a simple example, if $\theta_0 = (a, b)$ and $V(\theta_0) = XR_Z(a)R_X(b)S$, then $V(\theta_0)^\dagger = S^\dagger R_X(-b)R_Z(-a)X$.

Do not forget about this technique for estimating $\left|\langle 0| V(\theta_0)^\dagger V(\theta)|0\rangle\right|^2$ because we will be using it again in *Chapter 9*, *Quantum Support Vector Machines*, in a completely different context.

This concludes our theoretical study of VQE. In the next section, we will learn how to use this algorithm with Qiskit.

7.3 Using VQE with Qiskit

In this section, we will show how we can use Qiskit to run VQE on both simulators and actual quantum hardware. To do that, we will use a problem taken from quantum chemistry: determining the energy of the H_2 or dihydrogen molecule. Our first subsection is devoted to defining this problem.

7.3.1 Defining a molecular problem in Qiskit

To illustrate how we can use VQE with Qiskit, we will consider a simple quantum chemistry problem. We will imagine that we have two atoms of hydrogen forming an H_2 molecule and that we want to compute its ground state and its energy. For that, we need to obtain the Hamiltonian of the system, which is a little bit different from the kind of Hamiltonian that we are used to. The Hamiltonians that we have considered so far are called **qubit Hamiltonians**, while the one that we need to describe the energy of the H_2 molecule is called a **fermionic Hamiltonian** — the name comes from the fact that it involves fermions, that is, particles such as electrons, protons, and neutrons.

We do not need to go into all the details of the computation of this type of Hamiltonian (if you are curious, you can refer to *Chapter 4* in the book by Sharkey and Chancé [57]), because all the necessary methods are provided in the Qiskit Nature package. What is more, no quantum computer is involved in the process; it is all computed and estimated classically.

To obtain the fermionic Hamiltonian for the dihydrogen molecule with Qiskit, we need to install the Qiskit Nature package as well as the pyscf library, which is used for the computational chemistry calculations (please, refer to *Appendix D*, *Installing the Tools*, for the installation procedure and note that we will be using version 0.4.5 of the package).

Then, we can execute the following instructions:

```
from qiskit_nature.drivers import Molecule
from qiskit_nature.drivers.second_quantization import \
    ElectronicStructureMoleculeDriver, ElectronicStructureDriverType
```

```
from qiskit_nature.problems.second_quantization import \
    ElectronicStructureProblem

mol = Molecule(geometry=[['H', [0., 0., -0.37]],
                        ['H', [0., 0., 0.37]]])
driver = ElectronicStructureMoleculeDriver(mol, basis='sto3g',
        driver_type=ElectronicStructureDriverType.PYSCF)
problem = ElectronicStructureProblem(driver)
secqop = problem.second_q_ops()
print(secqop[0])
```

Here, we are defining a molecule consisting of two hydrogen atoms located at coordinates $(0, 0, -0.37)$ and $(0, 0, 0.37)$ (measured in angstroms), which is close to an equilibrium state for this molecule. We are using the default parameters, such as, for instance, establishing that the molecule is not charged. Then, we define an electronic structure problem; that is, we ask the pyscf library, through the Qiskit interface, to compute the fermionic Hamiltonian that takes into account the different possible configurations for the electrons of the two hydrogen atoms. This is done with something called **second quantization** (hence the name `second_q_ops` for the method that we use).

When we run this piece of code, we obtain the following output:

```
Fermionic Operator
register length=4, number terms=36
  -1.2533097866459775 * ( +_0 -_0 )
+ -0.47506884877217725 * ( +_1 -_1 )
+ -1.2533097866459775 * ( +_2 -_2 )
+ -0.47506884877217725 * ( +_3 -_3 )
+ -0.3373779634072241 * ( +_0 +_0 -_0 -_0 )
+ -0.0 ...
```

This is a truncated view of the fermionic Hamiltonian, involving something called **creation** and **annihilation** operators that describe how electrons move from one orbital to another (more details can be found in *Chapter 4* of the book by Sharkey and Chancé [57]).

That is all very nice, but we can't yet use it on our shiny quantum computers. For that, we need to transform the fermionic Hamiltonian into a qubit Hamiltonian, involving Pauli gates. There are several ways to do this. One of the most popular ones is the **Jordan-Wigner** transformation (again, see the book by Sharkey and Chancé [57] for a thorough explanation), that we can use in Qiskit with the following instructions:

```
from qiskit_nature.converters.second_quantization import QubitConverter
from qiskit_nature.mappers.second_quantization import JordanWignerMapper

qconverter = QubitConverter(JordanWignerMapper())
qhamiltonian = qconverter.convert(secqop[0])
print("Qubit Hamiltonian")
print(qhamiltonian)
```

Upon running this code, we will obtain the following output:

```
Qubit Hamiltonian
-0.8121706072487122 * IIII
+ 0.17141282644776915 * IIIZ
- 0.22343153690813483 * IIZI
+ 0.17141282644776915 * IZII
- 0.22343153690813483 * ZIII
+ 0.12062523483390415 * IIZZ
+ 0.16868898170361205 * IZIZ
+ 0.04530261550379923 * YYYY
+ 0.04530261550379923 * XXYY
+ 0.04530261550379923 * YYXX
+ 0.04530261550379923 * XXXX
+ 0.16592785033770338 * ZIIZ
+ 0.16592785033770338 * IZZI
+ 0.1744128761226159 * ZIZI
+ 0.12062523483390415 * ZZII
```

Now we are back on known territory once again! This is, indeed, one of the Hamiltonians that we have come to know and love. In fact, this is a Hamiltonian on 4 qubits, involving tensor products of I, X, Y and Z gates, as the ones appearing in terms such as `0.17141282644776915 * IIIZ` or `0.04530261550379923 * XXYY`.

What is more important to us: this is the kind of Hamiltonian to which we can apply the VQE algorithm in order to obtain its ground state. And, without further ado, that is exactly what we will be doing in the next subsection.

7.3.2 Using VQE with Hamiltonians

Now that we have a qubit Hamiltonian describing our electronic problem, let's see how we can use VQE with Qiskit to find its ground state. Remember that, to use VQE, we first need to choose an ansatz. To start with, we will use something simple. We will select one of the variational forms provided by Qiskit: the `EfficientSU2` form. We can define it and draw its circuit for 4 qubits with the following instructions (remember that you need to install the pylatexenc library to draw with the `"mpl"` option; please, refer to *Appendix D, Installing the Tools*):

```python
from qiskit.circuit.library import EfficientSU2

ansatz = EfficientSU2(num_qubits=4, reps=1, entanglement="linear",
    insert_barriers = True)
ansatz.decompose().draw("mpl")
```

Here, we have specified that we are using the variational form on 4 qubits, that we only use one repetition (that is, a single layer of CNOT gates) and that the entanglement that we want to use is linear: this means that each qubit is entangled with a CNOT gate to the following one. After running this piece of code, we will obtain the image depicted in *Figure 7.2*. As you can see, we are using R_Y and R_Z gates, together with entangling gates (CNOT gates, in this case). In total, we have 16 different tunable parameters, represented by $\theta[0], \ldots, \theta[15]$ in the figure. We will discuss more variational forms, similar to this one, in *Chapters 9* and *10*. For now, it is enough to notice that this is a circuit that we can easily implement in current quantum hardware (because it only involves simple one and two-qubit gates), but that allows us to create somewhat complicated quantum states, with entanglement among all the qubits.

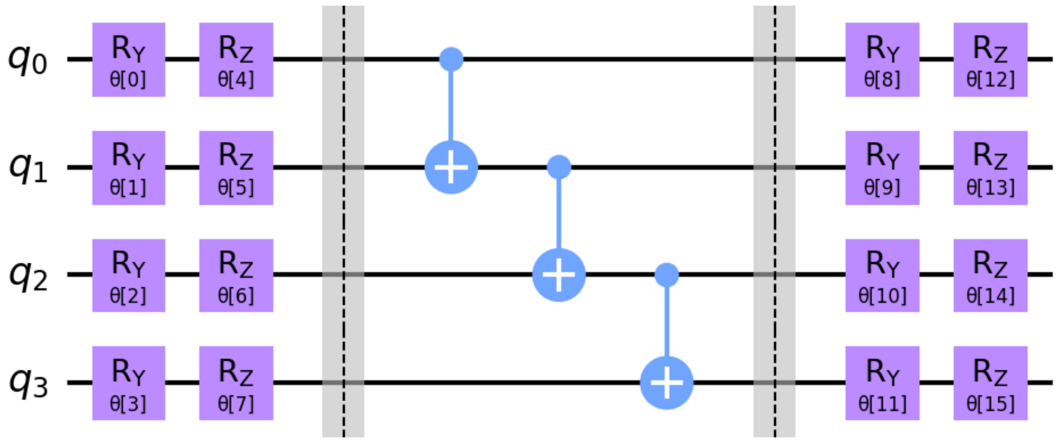

Figure 7.2: The EfficientSU2 variational form on 4 qubits.

Once we have selected our ansatz, we can define a VQE instance. In order to do that, we can use the following instructions:

```
from qiskit.algorithms import VQE
from qiskit import Aer
from qiskit.utils import QuantumInstance, algorithm_globals
import numpy as np
from qiskit.algorithms.optimizers import COBYLA

seed = 1234
np.random.seed(seed)
```

```
algorithm_globals.random_seed = seed

optimizer = COBYLA()

initial_point = np.random.random(ansatz.num_parameters)
quantum_instance = QuantumInstance(backend =
    Aer.get_backend('aer_simulator_statevector'))

vqe = VQE(ansatz=ansatz, optimizer=optimizer,
    initial_point=initial_point,
    quantum_instance=quantum_instance)
```

After the necessary imports, we set a seed for reproducibility. Then, we selected COBYLA as our classical optimizer; that is, the algorithm in charge of varying the parameters in order to find those that achieve the minimum energy. We also set some random initial values for our parameters and we declared a `QuantumInstance` that encapsulates a state vector simulator. Finally, we declared our VQE instance with the ansatz, optimizer, initial values, and quantum instance options.

Running the VQE is now very easy. We only need to execute the following instructions:

```
result = vqe.compute_minimum_eigenvalue(qhamiltonian)
print(result)
```

After a few seconds, we obtain the following output:

```
{   'aux_operator_eigenvalues': None,
    'cost_function_evals': 888,
    'eigenstate': array([ 1.55163279e-09+7.04522580e-10j,
        1.17994431e-06+6.29389934e-07j,
       -6.87287902e-05-1.19175176e-04j,  9.01607105e-09+1.75153048e-10j,
        3.17070261e-06-2.71251777e-05j, -9.23514532e-01-3.66685696e-01j,
       -6.50833666e-07-1.04178617e-06j, -6.40877389e-06-1.04499914e-05j,
       -1.33988128e-06+3.63309921e-07j,  1.08441415e-05+7.61755332e-08j,
        1.04578392e-01+4.15432635e-02j, -5.85921512e-06+4.47076415e-06j,
       -1.01179799e-09+1.85616927e-09j,  5.57085679e-05+5.29593190e-05j,
        1.47630244e-07+4.00357904e-08j,  1.51330159e-10+9.41869390e-10j]),
    'eigenvalue': (-1.8523881417094914+0j),
    'optimal_circuit': None,
    'optimal_parameters': {
        ParameterVectorElement(θ[7]): -0.10263498379273155,
        ParameterVectorElement(θ[6]): -0.13154223054201972,
        ParameterVectorElement(θ[8]): 3.1416468430294864,
        ParameterVectorElement(θ[13]): 0.6426987629297032,
        ParameterVectorElement(θ[9]): 2.4674114077579344e-05,
        ParameterVectorElement(θ[14]): -0.11387081297526412,
        ParameterVectorElement(θ[15]): 2.525254909939928,
        ParameterVectorElement(θ[12]): 1.8446272942674344,
        ParameterVectorElement(θ[11]): -0.0011789455587669483,
        ParameterVectorElement(θ[10]): 2.7179451047891577e-06,
        ParameterVectorElement(θ[3]): 3.1403232388683655,
        ParameterVectorElement(θ[1]): 9.061128731357842e-06,
        ParameterVectorElement(θ[2]): 3.141570826032646,
        ParameterVectorElement(θ[0]): -0.22553325325129397,
        ParameterVectorElement(θ[5]): 2.1513214842441912,
        ParameterVectorElement(θ[4]): 1.7045601611970793},
    'optimal_point': array([-2.25533253e-01,  9.06112873e-06,
        3.14157083e+00,  3.14032324e+00,
        1.70456016e+00,  2.15132148e+00, -1.31542231e-01, -1.02634984e-01,
        3.14164684e+00,  2.46741141e-05,  2.71794510e-06, -1.17894556e-03,
        1.84462729e+00,  6.42698763e-01, -1.13870813e-01,  2.52525491e+00]),
    'optimal_value': -1.8523881417094914,
    'optimizer_evals': None,
    'optimizer_result': None,
    'optimizer_time': 3.0011892318725586}
```

This may seem like a lot of information but, in fact, some data is presented several times in different ways and, all in all, the format is quite similar to what we are used to from our experience using QAOA in Qiskit back in *Chapter 5, QAOA: Quantum Approximate Optimization Algorithm*. As you can see, we have obtained the optimal values for the circuit parameters, the state that is generated with those parameters (the `'eigenstate'` field) and what we were looking for: the energy of that state, which happens to be about -1.8524 hartrees (the unit of energy commonly used in molecular orbital calculations). This means that…we have solved our problem! Or have we? How can we be sure that the value that we have obtained is correct?

In this case, the Hamiltonian that we are using is quite small (only 4 qubits), so we can check our result using a classical solver that finds the exact ground state. We will use `NumPyMinimumEigensolver`, just as we did with the combinatorial optimization problems that we considered back in *Chapter 5, QAOA: Quantum Approximate Optimization Algorithm*. For that, we can run the following piece of code:

```
from qiskit.algorithms.minimum_eigensolvers import \
    NumPyMinimumEigensolver
solver = NumPyMinimumEigensolver()
result = solver.compute_minimum_eigenvalue(qhamiltonian)
print(result)
```

The output of these instructions is the following:

```
{   'aux_operators_evaluated': None,
    'eigenstate': Statevector([-1.53666363e-17-4.93701060e-20j,
             -4.57234900e-16-4.65250782e-16j,
              1.25565337e-17-2.11612780e-17j,
              4.73690908e-16-1.33060132e-16j,
              1.52564317e-16-1.40021223e-16j,
             -6.67316913e-01-7.36221442e-01j,
             -1.62999711e-16-2.24584031e-16j,
             -8.42710421e-17+6.43081213e-17j,
             -7.98957973e-17-1.35250844e-17j,
              1.90408979e-16+3.25517112e-16j,
              7.55826341e-02+8.33870007e-02j,
             -3.56170534e-17+9.82948865e-17j,
             -4.51619835e-16+1.70721750e-16j,
              1.91645940e-17-1.45775129e-16j,
             -4.79331105e-17+5.57184037e-17j,
             -3.62080563e-17+4.86380668e-17j],
            dims=(2, 2, 2, 2)),
    'eigenvalue': -1.852388173569583}
```

This is certainly more concise than the VQE output, but the final energy is almost equal to the one we had obtained previously. Now we can really say it: we did it! We have successfully solved a molecular problem with VQE!

Of course, we can use VQE with any type of Hamiltonian, not just with the ones that come from quantum chemistry problems. We can even use it with Hamiltonians for combinatorial optimization problems, as we did back in *Chapter 5*, *QAOA: Quantum Approximate Optimization Algorithm*. With what we already know, this is easy…so easy that we entrust it to you as an exercise.

Exercise 7.7

Use Qiskit's VQE implementation to solve the Max-Cut problem for a graph of 5 vertices in which the connections are $(0,1), (1,2), (2,3), (3,4)$ and $(4,0)$.

Once we know how to find the ground state of a Hamiltonian with VQE, why not be a little more ambitious? In the next subsection, we will also be looking for excited states!

7.3.3 Finding excited states with Qiskit

Back in *Section 7.2.1*, we learned how to use VQE iteratively to find not only the ground state of a Hamiltonian, but also states of higher energy that we call excited states. The algorithm that we studied is sometimes called **Variational Quantum Deflation** (this was the name used by Higgot, Wang, and Brierley when they introduced it [58]) or **VQD**, and it is implemented by Qiskit in the `VQD` class.

Using VQD in Qiskit is almost the same as using VQE. The only difference is that we need to specify how many eigenstates we want to obtain (of course, if we only request 1 this will be *exactly* like applying VQE). For instance, if we want to obtain two eigenstates (the ground state and the first excited state) in our molecular problem, we can use the following instructions:

```
from qiskit.algorithms import VQD
vqd = VQD(ansatz=ansatz,
    optimizer=optimizer,
    initial_point=initial_point,
    quantum_instance=quantum_instance,
    k = 2)
result = vqd.compute_eigenvalues(qhamiltonian)
print(result)
```

The `k` parameter is the one that we use to specify the number of eigenstates. Upon running these instructions, we obtain the following output (we have omitted part of it for brevity):

```
{   'aux_operator_eigenvalues': None,
    'cost_function_evals': array([ 888, 1000]),
    'eigenstates': ListOp([VectorStateFn(Statevector(
            [ 1.55163279e-09+7.04522580e-10j,
              1.17994431e-06+6.29389934e-07j,
              ...
              1.51330159e-10+9.41869390e-10j],
            dims=(2, 2, 2, 2)), coeff=1.0,
            is_measurement=False),
            VectorStateFn(Statevector(
            [-5.01605162e-02+4.38928908e-02j,
             -7.31117975e-01-3.69461649e-02j,
             -6.34876999e-03-5.19845422e-03j,
             ...
             -4.10301081e-02+2.77415065e-02j],
            dims=(2, 2, 2, 2)), coeff=1.0,
            is_measurement=False)], coeff=1.0,
            abelian=False),
    'eigenvalues': array([-1.85238814-1.11e-16j, -1.19536442+0.00e+00j]),
    'optimal_circuit': None,
    'optimal_parameters': [
    {   ParameterVectorElement(θ[0]): -0.22553325325129397,
        ParameterVectorElement(θ[1]): 9.061128731357842e-06,
        ...
        ParameterVectorElement(θ[15]): 2.525254909939928},
    {   ParameterVectorElement(θ[0]): 0.012174657752649348,
        ParameterVectorElement(θ[1]): -0.056812096977499754,
        ...
        ParameterVectorElement(θ[15]): 1.522408417522795}],
    'optimal_point':
    array([[-2.25533253e-01,  9.06112873e-06,  3.14157083e+00,
         3.14032324e+00,  1.70456016e+00,  2.15132148e+00,
         ...
```

```
         2.52525491e+00],
       [ 1.21746578e-02, -5.68120970e-02,  1.31641034e+00,
         4.59223490e-01,  7.25749716e-01,  9.54546607e-02,
         ...
         1.52240842e+00]]),
    'optimal_value': array([-1.85238814, -1.1952203 ]),
    'optimizer_evals': None,
    'optimizer_result': None,
    'optimizer_time': array([ 2.32541203, 53.26829457])}
```

As you can see, this output is structured like that of the VQE execution. However, in this case, we get two entries in each field, one for each of the eigenstates that we requested.

So far, we have learned how to use both VQE and VQD with Hamiltonians that may have come from any source. However, the use case of finding ground states of molecular Hamiltonians is so important that Qiskit provides special methods for dealing with them in particular. We will learn how in the next subsection.

7.3.4 Using VQE with molecular problems

In addition to using VQE to find ground states of any given Hamiltonian, we can use it directly with molecular problems that we define with the help of the Qiskit Nature utilities. For instance, we can use a VQE instance to solve the electronic problem that we defined in the previous subsection. To do that, we only need to run the following instructions:

```python
from qiskit_nature.algorithms import GroundStateEigensolver

solver = GroundStateEigensolver(qconverter, vqe)
result = solver.solve(problem)
print(result)
```

As you can see, we have defined a `GroundStateEigensolver` object that we then use to solve our problem. This object, in turn, uses two objects that we had defined previously —

qconverter, which is used to transform the fermionic Hamiltonian into a qubit Hamiltonian, and the instance of VQE that we used two subsections ago. When we run these instructions, we get the following output:

```
=== GROUND STATE ENERGY ===

* Electronic ground state energy (Hartree): -1.852388141709
  - computed part:      -1.852388141709
~ Nuclear repulsion energy (Hartree): 0.715104339081
> Total ground state energy (Hartree): -1.137283802628

=== MEASURED OBSERVABLES ===

  0:  # Particles: 2.000 S: 0.000 S^2: 0.000 M: 0.000

=== DIPOLE MOMENTS ===

~ Nuclear dipole moment (a.u.): [0.0  0.0  0.0]

  0:
  * Electronic dipole moment (a.u.): [0.0  0.0  0.00001495]
    - computed part:      [0.0  0.0  0.00001495]
  > Dipole moment (a.u.): [0.0  0.0  -0.00001495]  Total: 0.00001495
                 (debye): [0.0  0.0  -0.000038]  Total: 0.000038
```

The information that we get in this case is at a higher level of abstraction than the one we obtained before. For instance, we get data about the number of particles, dipole moments, and so on (don't worry if you do not understand these concepts; they are meant to make sense to chemists and physicists that work with this kind of problem). However, the numerical result of the electronic ground state energy is the same that we obtained with our previous application of VQE. The difference is that now, we are providing the solver not only with the Hamiltonian, but with the whole problem, and it can use that information to reconstruct the meaning of the calculations in physical terms. For instance, we now get some bonus information such as the **total ground state energy**, which is the sum of the energy due to the electronic structure (the one that we had computed previously) and the energy due to nuclear repulsion.

This type of output is much more legible. That's why we will use this solver for the rest of this section.

As an additional example of how to use VQE to solve molecular problems, we are now going to consider a different, more elaborate ansatz. Earlier in this chapter, we mentioned how, when selecting the variational form and initial state to be used with VQE, it could prove useful to take into account information from the problem domain. This is the case of the **Unitary Coupled-Cluster Singles and Doubles** or **UCCSD** ansatz, which is widely used for molecular computations (see the survey by McArdle et al. [54] for more details).

In Qiskit, we can use the UCSSD ansatz with the following instructions:

```
from qiskit_nature.algorithms import VQEUCCFactory

vqeuccf = VQEUCCFactory(quantum_instance = quantum_instance)
```

The `VQEUCCFactory` class creates a whole VQE instance, with the UCSSD ansatz as the default variational form. Here, we are using the `quantum_instance` object that we had defined previously. We can visualize the circuit for the ansatz created by `VQEUCCFactory` with the following instruction:

```
vqeuccf.get_solver(problem, qconverter).ansatz.decompose().draw("mpl")
```

Notice that we are calling the `get_solver` method, to which we pass the `problem` object defined previously to provide the information about the Hamiltonian involved in the computation. Then, we access the ansatz circuit through the `ansatz` attribute and we proceed to draw it. Upon running this instruction, we obtain the circuit depicted in *Figure 7.3*. As you can see, the ansatz involves exponential functions of tensor products of Pauli matrices. There are also two X gates at the beginning of the circuit that set the initial state to which the variational form is later applied. In this case, the state is called the **Hartree-Fock** state, again a widely used option when solving molecular problems with quantum computers — and the default value with `VQEUCCFactory`.

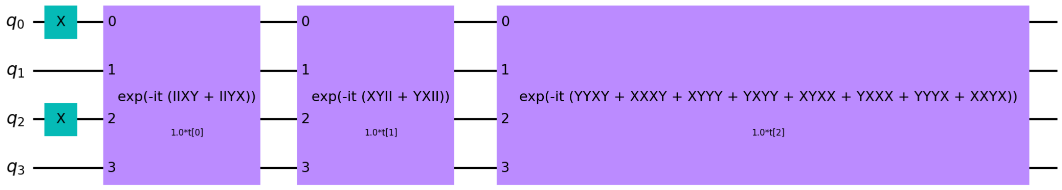

Figure 7.3: UCCSD ansatz for our problem

Now, we can easily use VQE to solve our problem with the selected ansatz by running the following piece of code:

```
solver = GroundStateEigensolver(qconverter, vqeuccf)
result = solver.solve(problem)
print(result)
```

This will give us the following output:

```
=== GROUND STATE ENERGY ===


* Electronic ground state energy (Hartree): -1.852388173513
  - computed part:      -1.852388173513
~ Nuclear repulsion energy (Hartree): 0.715104339081
> Total ground state energy (Hartree): -1.137283834432


=== MEASURED OBSERVABLES ===


  0:  # Particles: 2.000 S: 0.000 S^2: 0.000 M: 0.000


=== DIPOLE MOMENTS ===
~ Nuclear dipole moment (a.u.): [0.0  0.0  0.0]


  0:
  * Electronic dipole moment (a.u.): [0.0  0.0  -0.00000013]
    - computed part:      [0.0  0.0  -0.00000013]
  > Dipole moment (a.u.): [0.0  0.0  0.00000013]  Total: 0.00000013
                 (debye): [0.0  0.0  0.00000033]  Total: 0.00000033
```

This result is very similar to the one that we obtained with the `EfficientSU2` ansatz.

Exercise 7.8

Write code to use VQE with the UCCSD ansatz to compute the total ground state energy for two atoms of hydrogen that are at distances ranging from 0.2 to 2.0 angstroms, in steps of 0.01 angstroms. Plot the energy against the distance. This kind of plot is sometimes known as the **dissociation profile** of the molecule. *Hint:* when running VQE on a molecular problem, you can access the total ground state energy through the `total_energies` attribute of the result object.

Now that we know how to use VQE in different ways with simulators, we could try to run the algorithm on actual quantum computers. Nevertheless, before doing that, we will learn how to incorporate noise to our quantum simulator.

7.3.5 Simulations with noise

Going from a perfect, classical simulation of an algorithm to an execution on an actual quantum device can, sometimes, be too big a step. As we have mentioned in many occasions, current quantum computers suffer from the effect of different types of noise, including readout errors, imperfections in gate implementation, and decoherence, the loss of quantum properties of our states if the circuits are too deep.

For this reason, it is usually a good idea to perform a simulation of our algorithm under the effects of noise before going to the actual quantum device. In this way, we can study the performance of our algorithms in a controlled environment, and calibrate and adjust some of their parameters before running them on a quantum computer. For instance, if we observe that the results differ much from ideal simulation, we could decide to reduce the depth of our circuits by using a simpler ansatz.

There are several ways of conducting noisy simulations with Qiskit. Here, we will show how to use one that is both easy and very useful. We will create a simulator that mimics the behaviour of a real device, including the noise it is affected by. We can do this with the help of the `AerSimulator` class in the following way:

```
from qiskit.providers.aer import AerSimulator
from qiskit import IBMQ

provider = IBMQ.load_account()
backend = provider.get_backend('ibmq_manila')
quantum_instance = QuantumInstance(
    backend = AerSimulator.from_backend(backend),
    seed_simulator=seed, seed_transpiler = seed, shots = 1024)
```

Notice that we need to load an IBM account in order to access the calibration of a real device (`ibmq_manila`, in our example). This calibration is updated periodically to stay real to the state of the quantum computer and includes, among other things, information about readout errors, gate errors, and coherence times. Of course, this data will change from time to time, but we have decided to include seeds for our `QuantumInstance` object to make the result reproducible given the same calibration data. Notice that we are now specifying the number of shots, because we are not using state vector simulation anymore.

Now, we can run the VQE algorithm exactly as before:

```
vqe = VQE(
    ansatz=ansatz,
    optimizer=optimizer,
    initial_point=initial_point,
    quantum_instance=quantum_instance
)

solver = GroundStateEigensolver(qconverter, vqe)
result = solver.solve(problem)
print(result)
```

When we ran this code, we obtained the following output (your results may be different, depending on the device calibration):

```
=== GROUND STATE ENERGY ===

* Electronic ground state energy (Hartree): -1.763282965888
  - computed part:      -1.763282965888
~ Nuclear repulsion energy (Hartree): 0.715104339081
> Total ground state energy (Hartree): -1.048178626807

=== MEASURED OBSERVABLES ===

  0:  # Particles: 1.978 S: 0.080 S^2: 0.086 M: 0.001

=== DIPOLE MOMENTS ===

~ Nuclear dipole moment (a.u.): [0.0  0.0  0.0]

  0:
  * Electronic dipole moment (a.u.): [0.0  0.0  0.04634607]
    - computed part:      [0.0  0.0  0.04634607]
```

```
 > Dipole moment (a.u.): [0.0  0.0  -0.04634607]  Total: 0.04634607
                 (debye): [0.0  0.0  -0.11779994]  Total: 0.11779994
```

Observe how the effect of noise has affected the performance of the algorithm and degraded it, giving a result for the total ground state energy that is not that close to the correct one.

To learn more...

An alternative way of mimicking the behavior of real quantum computers is using objects of the `FakeProvider` class. The difference is that they use snapshots of past calibrations of the devices instead of the latest ones. You can find more details at `https://qiskit.org/documentation/apidoc/providers_fake_provider.html`.

Additionally, you can create custom noise models that include the different noise types implemented in the `qiskit_aer.noise` package. Check the documentation at `https://qiskit.org/documentation/apidoc/aer_noise.html` for further explanation.

A way to try of reducing the adverse effects of noise in our computations is using **readout error mitigation** methods. The idea behind the particular method that we are going to use is very simple. Imagine that we know that, when the state of our qubit is $|0\rangle$, there is a certain percentage of times it in which we obtain the incorrect value 1 when we measure it. Then, we can take into account this information to correct the measurement results that we have obtained.

In Qiskit, using readout error mitigation is very easy. We only need to create our Quantum Instance object in the following way:

```
from qiskit.utils.mitigation import CompleteMeasFitter

quantum_instance = QuantumInstance(
    backend = AerSimulator.from_backend(backend),
    measurement_error_mitigation_cls=CompleteMeasFitter,
    seed_simulator=seed, seed_transpiler = seed, shots = 1024)
```

Then, we can run VQE as usual, using this new `QuantumInstance` variable. In our case, that led to the following result (again, yours will likely differ because of the device calibration):

```
=== GROUND STATE ENERGY ===

* Electronic ground state energy (Hartree): -1.827326686753
  - computed part:      -1.827326686753
~ Nuclear repulsion energy (Hartree): 0.715104339081
> Total ground state energy (Hartree): -1.112222347671

=== MEASURED OBSERVABLES ===

  0:  # Particles: 1.991 S: -0.000 S^2: -0.000 M: -0.006

=== DIPOLE MOMENTS ===

~ Nuclear dipole moment (a.u.): [0.0  0.0  0.0]

  0:
  * Electronic dipole moment (a.u.): [0.0  0.0  -0.05906852]
    - computed part:      [0.0  0.0  -0.05906852]
  > Dipole moment (a.u.): [0.0  0.0  0.05906852]  Total: 0.05906852
                 (debye): [0.0  0.0  0.15013718]  Total: 0.15013718
```

As you can see, our current result — compared with our previous simulation with noise and no error mitigation — is now much closer to the real ground state energy (although you have surely noticed that there is still room for improvement).

In order to run this kind of error mitigation, we need to know the probability of measuring y when we actually have $|x\rangle$ for every pair of binary strings x and y. Of course, estimating these values is computationally very expensive, because the number of strings grows exponentially with the number of qubits. Alternatively, we could assume that the readout errors are local and estimate instead the probability of obtaining an incorrect result for each individual qubit only. In Qiskit, you choose to take this approach by replacing `CompleteMeasFitter` with `TensoredMeasFitter`. However, at the time of writing, not all backends support this possibility, so you'd better be careful if you decide to use it.

To learn more…

There is much more to say about trying to mitigate the effects of noise in quantum computations. Unfortunately, studying error mitigation further would make this chapter much, much longer (and it is already fairly long!). Should you be interested in this topic, we can recommend that you check the paper by Bravyi et al. [59] to learn more about measurement error mitigation and the papers by Temme et al. [60] and by Endo et al. [61] to learn more about how to mitigate errors in general, including the ones causes by imperfect gate implementation. You may also want to take a look at Mitiq, a very easy-to-use software package for error mitigation that is compatible with Qiskit and other quantum computing libraries [62].

The techniques that we have introduced to simulate noisy devices and to mitigate readout errors are not only applicable to the VQE algorithm. In fact, noisy simulation can be used when running any circuit, because we can just use a `backend` object created with the `AerSimulator.from_backend` function and a real quantum computer.

Moreover, readout error mitigation can be used with any algorithm that uses an object of the class `QuantumInstance` to run circuits. This includes QAOA and GAS, which we studied in *Chapters 5* and *6*, respectively, as well as the QSVMs, the QNNs and the QGANs that we will study in *Chapters 9*, *10*, *11*, and *12*.

But the possibilities don't end there. In fact, every `QuantumInstance` object provides an `execute` method that receives quantum circuits and executes them. So, you can create a `QuantumInstance` with a noisy backend and the `measurement_error_mitigation_cls` argument, and then invoke `execute` to obtain results with error mitigation.

Exercise 7.9

Create a noisy backend from a real quantum computer. Then, use it to run a simple two-qubit circuit consisting of a Hadamard gate on the first qubit, a CNOT gate with control on the first qubit and target in the second, and a final measurement of both qubits. Compare the results to those of ideal simulation. Then, create a `QuantumInstance` from your backend and using readout error mitigation. Run the circuit with it. Compare the results to what you obtained before.

Exercise 7.10

Run QAOA with a simple Hamiltonian on a noisy simulator with and without readout error mitigation. Compare the results.

Now that we know how to run simulations with noise, we are ready for the next big step: let's run VQE on actual quantum devices.

7.3.6 Running VQE on quantum computers

By now, you surely have guessed what we are going to say about running VQE on quantum devices. If you were thinking that we could just use a real backend when creating our `QuantumInstance` object, but that it would involve waiting multiple queues and that there must be a better way, you were completely spot on. In fact, we can use Runtime to send our VQE jobs to IBM's quantum computers, waiting only in one execution queue. The way in which we can use VQE with Runtime is very similar to what we showed in *Section 5.2.1* for QAOA. We can use the `VQEClient` as follows:

```
from qiskit_nature.runtime import VQEClient
backend = provider.get_backend('ibmq_manila')

vqe = VQEClient(
    ansatz=ansatz,
    provider=provider,
    backend=backend,
    shots=1024,
    initial_point = initial_point,
    measurement_error_mitigation=False
)

solver = GroundStateEigensolver(qconverter, vqe)
result = solver.solve(problem)
print(result)
```

This is completely analogous to how we run VQE on local simulators, but now we are sending the task to the real quantum device called `ibmq_manila`. Notice that we have specified the number of shots and that we have opted to use the default optimizer since we haven't provided a value for the optimizer argument. The default optimizer for this algorithm is SPSA.

The results that we obtained (after waiting some time in the queue) were the following:

```
=== GROUND STATE ENERGY ===

* Electronic ground state energy (Hartree): -1.745062049527
  - computed part:      -1.745062049527
~ Nuclear repulsion energy (Hartree): 0.715104339081
> Total ground state energy (Hartree): -1.029957710446

=== MEASURED OBSERVABLES ===

  0:  # Particles: 1.988 S: 0.131 S^2: 0.149 M: -0.005

=== DIPOLE MOMENTS ===
~ Nuclear dipole moment (a.u.): [0.0  0.0  0.0]


  0:
  * Electronic dipole moment (a.u.): [0.0  0.0  0.01726618]
    - computed part:      [0.0  0.0  0.01726618]
  > Dipole moment (a.u.): [0.0  0.0  -0.01726618]  Total: 0.01726618
                 (debye): [0.0  0.0  -0.04388625]  Total: 0.04388625
```

We can observe again the effect of noise in this execution. Of course, we can try to reduce it by setting `measurement_error_mitigation=True` and running the same code again. When we did that, we obtained the following output:

```
=== GROUND STATE ENERGY ===


* Electronic ground state energy (Hartree): -1.830922842008
  - computed part:      -1.830922842008
~ Nuclear repulsion energy (Hartree): 0.715104339081
> Total ground state energy (Hartree): -1.115818502927


=== MEASURED OBSERVABLES ===


  0:  # Particles: 2.020 S: 0.035 S^2: 0.036 M: 0.010


=== DIPOLE MOMENTS ===


~ Nuclear dipole moment (a.u.): [0.0  0.0  0.0]


  0:
  * Electronic dipole moment (a.u.): [0.0  0.0  -0.00999621]
    - computed part:      [0.0  0.0  -0.00999621]
  > Dipole moment (a.u.): [0.0  0.0  0.00999621]  Total: 0.00999621
                 (debye): [0.0  0.0  0.02540783]  Total: 0.02540783
```

That is a little bit better, right?

With this, we have covered everything we wanted to tell you about how to run VQE with Qiskit…or almost everything. In the next subsection, we will show you some new features that are being added to Qiskit and that can change the way in which algorithms such as VQE are used.

7.3.7 The shape of things to come: the future of Qiskit

Quantum computing software libraries are in constant evolution and Qiskit is no exception to this rule. Although everything that we have studied in this section is the main way of running VQE with the latest version of Qiskit (which is 0.39.2 at the time of writing this book), a new way of executing the algorithm is also being introduced and it will likely become the default one in the not-so-distant future.

This new way of doing things involves some modifications, the most important of which is replacing the use of `QuantumInstance` objects with `Estimator` variables. An `Estimator` is an object that is capable running a parametrized circuit to obtain a quantum state and then estimate (who would have guessed?) the expectation value of some observable on that state. Of course, this is exactly what we need in order to be able to run VQE, as you surely remember from *Section 7.2*.

Let's see an example of how this would work. The following code is a possible way of running VQE to solve the same molecular problem that we have been considering throughout this section with the new implementations that are being introduced in Qiskit:

```
from qiskit.algorithms.minimum_eigensolvers import VQE
from qiskit.primitives import Estimator

estimator= Estimator()
```

```
vqe = VQE(
    ansatz=ansatz,
    optimizer=optimizer,
    initial_point=initial_point,
    estimator=estimator
)

result = vqe.compute_minimum_eigenvalue(qhamiltonian)

print(result)
```

Notice that we are importing `VQE` from `qiskit.algorithms.minimum_eigensolvers`, not from `qiskit.algorithms` as before. Also notice how the `Estimator` object has replaced the `QuantumInstance` one that we used to use.

Running these instructions will give an output like the following one (shortened here for brevity):

```
{   'aux_operators_evaluated': None,
    'cost_function_evals': 1000,
    'eigenvalue': -1.8523881060316512,
    'optimal_circuit':
    <qiskit.circuit.library.n_local.efficient_su2.EfficientSU2
    object at 0x7f92367aac90>,
    'optimal_parameters': {
        ParameterVectorElement(θ[10]): -5.6469331359894016e-05,
        ParameterVectorElement(θ[7]): -0.07317113283182797,
        ...
        ParameterVectorElement(θ[15]): 2.5406547025358206},
    'optimal_point': array(
        [-2.25566150e-01, -3.48673819e-05, 3.14159358e+00,  3.13967948e+00,
        1.74766932e+00,  2.19381131e+00, -1.17362733e-01, -7.31711328e-02,
        3.14163959e+00, -5.24406909e-05, -5.64693314e-05, -1.86976530e-03,
        1.95315840e+00,  6.62795965e-01, -1.43666055e-01,  2.54065470e+00]),
    'optimal_value': -1.8523881060316512,
    'optimizer_evals': None,
    'optimizer_result':
    <qiskit.algorithms.optimizers.optimizer.OptimizerResult
    object at 0x7f9240af82d0>,
    'optimizer_time': 6.93215799331665}
```

This should sound familiar because it is the same kind of result that we obtained when using the current VQE implementation directly on a Hamiltonian.

As you can see, things are not going to change a lot in this new version. The main novelty is the use of `Estimator` objects. So, how do they work? Well, it depends. For instance, the one that we have imported from `qiskit.primitives` uses a state vector simulator to obtain a quantum state from a circuit. Then, it computes its expectation value by calling the `expectation_value` method, as we did back in *Section 3.2.2*. However, the `Estimator` class implemented in `qiskit_aer.primitives` uses the method that we explained in *Section 7.1.2* by appending additional gates to the parametrized circuit in order to perform measurements in different bases.

Unfortunately, at the time of writing this book, some of the features that we have covered in this section, such as noisy simulation and error mitigation, are still not completely supported by the new version of the algorithms. Moreover, some of the `Estimator` classes are not fully compatible with the new VQE implementation yet.

However, Qiskit changes rapidly, so maybe, in the future, you can fully reproduce our code with `Estimator` objects instead of `QuantumInstance` ones by the time you will be reading these lines. Time will tell!

Important note

The changes that we have described in this subsection are expected to also affect other algorithms implemented in Qiskit, such as VQD or QAOA. In the case of QAOA, instead of `Estimator` objects, you will need to use `Sampler` objects. As you can imagine, they will let you obtain samples from parametrized circuits, which can later be used by QAOA to estimate the value of the cost function.

And now, we promise, this is really all we wanted to tell you about running VQE with Qiskit. Our next stop is PennyLane.

7.4 Using VQE with PennyLane

In this section, we will illustrate how to run VQE with PennyLane. The problem that we will work with will be, again, finding the ground state of the dihydrogen molecule. This is a task we are already familiar with and, moreover, this will allow us to compare our results with those that we obtained with Qiskit in the previous section. So, without further ado, let's start by showing how to define the problem in PennyLane.

7.4.1 Defining a molecular problem in PennyLane

As with Qiskit, PennyLane provides methods to work with quantum chemistry problems. To study the H_2 molecule, we can use the following instructions:

```
import pennylane as qml
from pennylane import numpy as np

seed = 1234
np.random.seed(seed)

symbols = ["H", "H"]
coordinates = np.array([0.0, 0.0, -0.6991986158, 0.0, 0.0, 0.6991986158])
```

```
H, qubits = qml.qchem.molecular_hamiltonian(symbols, coordinates)
print("Qubit Hamiltonian: ")
print(H)
```

You may be thinking that there is something fishy here. When we defined this same molecule in Qiskit, we used coordinates `[0., 0., -0.37],[0., 0., 0.37]`, which seem different from the ones that we are using now. The explanation for this change is that, while Qiskit uses angstroms to measure atomic distances, PennyLane expects the values to be in atomic units. An angstrom is worth 1.8897259886 atomic units, hence the difference.

We can now obtain the qubit Hamiltonian that we need to use with VQE by running the following piece of code:

```
H, qubits = qml.qchem.molecular_hamiltonian(symbols, coordinates)
print("Qubit Hamiltonian: ")
print(H)
```

The output that we obtain is the following:

```
Qubit Hamiltonian:
  (-0.22343155727095726) [Z2]
+ (-0.22343155727095726) [Z3]
+ (-0.09706620778626623) [I0]
+ (0.17141283498167342) [Z1]
+ (0.1714128349816736) [Z0]
+ (0.12062523781179485) [Z0 Z2]
+ (0.12062523781179485) [Z1 Z3]
+ (0.16592785242008765) [Z0 Z3]
+ (0.16592785242008765) [Z1 Z2]
+ (0.16868898461469894) [Z0 Z1]
+ (0.17441287780052514) [Z2 Z3]
+ (-0.04530261460829278) [Y0 Y1 X2 X3]
+ (-0.04530261460829278) [X0 X1 Y2 Y3]
+ (0.04530261460829278) [Y0 X1 X2 Y3]
+ (0.04530261460829278) [X0 Y1 Y2 X3]
```

If you compare this Hamiltonian to the one that we obtained with Qiskit for the same problem you will notice that they are very different. But don't panic yet. While Qiskit gave us the Hamiltonian for the electronic structure of the molecule, PennyLane is accounting for the total energy, including nuclear repulsion. We will run the algorithm in a moment and, trust us, we will see how everything adds up.

7.4.2 Implementing and running VQE

Before using VQE, we need to decide what variational form we are going to use as the ansatz. To keep things simple, we will stick with the `EfficientSU2` that we used in the previous section.

This variational form is not included in PennyLane at the time of writing this book, but we can easily implement it with the following code:

```python
nqubits = 4
def EfficientSU2(theta):
    for i in range(nqubits):
        qml.RY(theta[i], wires = i)
        qml.RZ(theta[i+nqubits], wires = i)
    for i in range(nqubits-1):
        qml.CNOT(wires = [i, i + 1])
    for i in range(nqubits):
        qml.RY(theta[i+2*nqubits], wires = i)
        qml.RZ(theta[i+3*nqubits], wires = i)
```

Notice that we have fixed the number of repetitions to 1, which was the case that we were using with Qiskit in the previous section.

Now that we have our variational form, we can use it to implement the VQE algorithm in PennyLane. To do that, we will define the energy function, which we compile as a quantum node because it needs to be evaluated on a device capable of running quantum circuits. We can do that with the following instructions:

```
dev = qml.device("lightning.qubit", wires=qubits)
@qml.qnode(dev)
def energy(param):
    EfficientSU2(param)
    return qml.expval(H)
```

Notice how we have used the `EfficientSU2` ansatz followed by an evaluation of the expectation value of our Hamiltonian (by using the `qml.expval` function that we introduced in *Chapter 5*, *QAOA: Quantum Approximate Optimization Algorithm*). Now, to execute VQE, we only need to select some initial values for the ansatz parameters and use a minimizer to find their optimal values. We can achieve that with the following piece of code:

```
from scipy.optimize import minimize

theta = np.array(np.random.random(4*nqubits), requires_grad=True)
result = minimize(energy, x0=theta)

print("Optimal parameters", result.x)
print("Energy", result.fun)
```

We have imported the `minimize` function from the `scipy.optimize` package (scipy is a powerful and very popular Python library for scientific computing). We have chosen at random some initial values for the variational form parameters. We have used `requires_grad=True` to allow the minimizer to compute gradients in order to optimize the parameters (we will have much more to say about this in *Part 3* of the book). Then, we have minimized the energy function using the default parameters of the `minimize` method. Notice how the `x0` argument is used to specify the initial values.

The result we obtain upon running this code is the following:

```
Optimal parameters
[ 2.25573385e-01  3.14158133e+00  1.91103424e-07 -1.88149577e-06
 -2.71613763e-03 -7.94107899e-01  4.52510610e-01  6.17686238e-01
  3.14158772e+00  6.28319382e+00  3.14158403e+00  3.14160984e+00
  2.21495304e-01  5.01302639e-01  6.51839333e-01  7.36625551e-02]
Energy -1.137283835001276
```

This includes the optimal values found by the optimizer (the x field) as well as the minimum energy. As you can check, this fits nicely with the results that we have obtained with Qiskit for the total molecular energy.

Now that we know how to run VQE on a simulator with PennyLane, we will turn to the task of executing the algorithm on actual quantum computers.

7.4.3 Running VQE on real quantum devices

You may remember that, back in *Section 5.3*, we mentioned that there is a PennyLane Runtime client that can be used to run VQE programs. This is exactly what we need now, so it is the perfect moment to learn how to use it.

In fact, using this Runtime implementation is very easy, because it is quite similar to the one we used with Qiskit. First, we need be sure that we have `pennylane_qiskit` installed and our IBM Quantum account enabled (see *Appendix D*, *Installing the Tools*, for directions). Then, we can run the following instructions:

```
from pennylane_qiskit import upload_vqe_runner, vqe_runner

program_id = upload_vqe_runner(hub="ibm-q", group="open", project="main")

job = vqe_runner(
    program_id=program_id,
    backend="ibm_oslo",
    hamiltonian=H,
    ansatz=EfficientSU2,
    x0=np.array(np.random.random(4*nqubits)),
    shots=1024,
    optimizer="SPSA",
    kwargs={"hub": "ibm-q", "group": "open", "project": "main"}
)
print(job.result())
```

The code is pretty much self-explanatory: we are just selecting the options for our VQE execution, including the device to run the circuits which, in this case, happens to be `ibm_oslo`. After waiting for the job to finish running, we will obtain an output similar the following:

```
     fun: -1.0125211856761642
 message: 'Optimization terminated successfully.'
    nfev: 300
     nit: 100
 success: True
       x: array([-0.02558326,  0.50137847,  1.49781722,  2.83016638,
       1.50688742, -0.00830098,  1.56006908, -0.01401641, -0.08208851,
       2.71490414, 1.39380584,  1.30662208,  1.5691855 ,  1.34979806,
       1.50345895, 0.39946571])
```

You may be wondering if we can also use error mitigation to try to improve our results. The answer is yes, of course. In fact, it is straightforward to set up, because we only need to include the additional parameter `use_measurement_mitigation = True` when creating the `vqe_runner` object. Running with this option will give you a result similar to the following one, which is closer to the real optimal value:

```
     fun: -1.0835711819668128
 message: 'Optimization terminated successfully.'
```

```
    nfev: 300
     nit: 100
 success: True
       x: array([-0.06213913,  2.62825807,  2.85476345, -0.2260965,
       -0.07639407, -1.51018602,  1.73431192, -0.07301669, -0.16907148,
       2.60134032, 3.29831133, -0.2912491 ,  0.33893055,  1.90085806,
       1.7206114 , -1.49009082])
```

With this, we conclude our study of VQE and, in fact, we conclude the part of the book devoted to optimization problems. Starting with the next chapter, we will dive into the fascinating world of quantum machine learning. Hang tight and prepare for the ride!

Summary

In this chapter, we have studied Hamiltonians and observables in detail. In particular, we have learned how to derive mathematical expressions for their expectation values and how to estimate these quantities using quantum computers.

Then, we studied the VQE algorithm and how it can be used to find ground states of general Hamiltonians. We also described a modification of VQE called VQD that can also be used to compute excited states and not just states of minimum energy.

Then, we moved to practical matters and learned how to use Qiskit to run VQE and VQD. We illustrated this with a very interesting problem: that of finding the ground state of a simple molecule. We then introduced methods to simulate the behavior of quantum algorithms when there is noise and how to reduce the adverse effect of readout errors with some mitigation techniques. We also studied how to run VQE problems on actual quantum computers with IBM runtime.

After that, we also learned how to implement and run VQE on PennyLane, again solving a molecular structure problem. We even studied how to use Runtime from PennyLane to send VQE instances to real quantum computers.

After reading this chapter, you are now able to understand all the mathematical details behind the VQE algorithm. You also know how to run it on different types of problems with both Qiskit and PennyLane. Moreover, you now can run noisy simulations of all the algorithms that we have studied (and of any other quantum algorithm that you may learn in the future) as well as perform readout error mitigation on simulated and actual quantum devices.

In the next chapter, we will start studying the second big topic of the book: quantum machine learning. Prepare to learn how (quantum) machines learn!