Analyzing and Implementing the Variational Quantum Eigensolver (VQE) Algorithm with Cirq

Austin Hunt
Department of Computer Science
Vanderbilt University
Greenville, SC
austin.j.hunt@vanderbilt.edu

***Abstract*—This paper provides an analysis of the Variational Quantum Eigensolver (VQE) algorithm, a hybrid quantum-classical algorithm that leverages the variational method of quantum mechanics to efficiently calculate the ground state, or the lowest-energy state, of a quantum system, otherwise referred to as the minimum eigenvalue of that system's Hamiltonian; the ground state of a quantum system plays an important role in its simulation, and thus our ability to understand, predict, and exploit its behavior. The VQE algorithm belongs to a larger class of Variational Quantum Algorithms (VQAs) that are designed not just for calculating ground state energy levels, but for generally minimizing objective functions that can be expressed as quantum observables (such as the Hamiltonian operator). Motivation for the VQE algorithm is first discussed, followed by an overview of how the quantum algorithm works and finally a discussion of its implementation using Google's Cirq library for developing and running quantum circuits in Python.**

***Index Terms*—quantum computing, quantum algorithm, variational quantum eigensolver, variational method, optimization**

I. Motivation

There is an irony in the certain and consistent forward march of the quantum computing field given its inner probabilistic nature. While quantum computing does continue to rapidly advance — such as in the discovery of novel quantum states [1] and in the development of efficient electro-optic modulators for altering photon frequencies [2] — we are still living in the noisy intermediate-scale quantum (NISQ) era, a term coined by John Preskill in 2018 [3]. This means that the leading quantum processors, including Google's 53-qubit Sycamore processor that helped them achieve quantum supremacy[1] in 2019 [5] and even the recently announced 433-qubit IBM Osprey processor [6], are still not advanced enough to overcome the challenge of decoherence that arises through environmental noise to which these processors are very sensitive. This lingering, significant challenge of quantum decoherence (the undesired loss of quantum properties by quantum systems) means that we are still in search of true quantum advantage — that is, quantum processors remain too error-prone to demonstrate real-world advantages over classical approaches to solving real-world problems. Of course, this does have the silver lining of RSA not being completely cracked by Shor's algorithm yet, but as we know from the existence of the NIST Post-Quantum Cryptography Standardization project [7], "yet" is the key word and that silver line is more of a weakening thread. In essence, quantum hardware just needs to catch up with quantum theory.

We are seeing a number of creative approaches to address noise and decoherence in the industry, such as IBM's release of a beta update to the Qiskit Runtime offering users a new tunable "resilience level" option to intentionally trade speed for reduced error count [8] and a Japanese research team's use of light pulses driven by artificial intelligence to stabilize noisy quantum systems [9].

Moreover, the current limitations of quantum computers are also being addressed even at the level of algorithm design through the creation of hybrid quantum-classical algorithms that rely on both quantum and classical components to solve problems. With a hybrid algorithm, one reduces the demand on quantum hardware and consequently the error that would result from quantum decoherence. The Variational Quantum Eigensolver (VQE) algorithm discussed in Section II is one such hybrid algorithm. While VQE itself does not address the problem of quantum decoherence, the error-correction techniques it employs through its hybrid design reduce the impact of decoherence on the accuracy of calculations, and that is a significant goal in the advancement of quantum computation. VQE ultimately aids in the simulation of complex quantum systems and in the solving of large-scale linear algebra problems — particularly those that are computationally infeasible to solve for classical computers — and consequently has a broad range of quantum computing applications in fields like chemistry, materials science, and condensed matter physics; it is this ubiquity of VQE that motivated this research.

II. The Variational Quantum Eigensolver (VQE) Algorithm

Originally proposed by Peruzzo et al. in a 2014 Nature Communications paper [10], the VQE algorithm is a NISQ algorithm (an algorithm designed to run on NISQ era quantum computers) that leverages both quantum and classical compu-

[1]As discussed by Kevin Hartnett, a writer for Quanta Magazine, "quantum supremacy" refers not to actual supremacy of quantum computing over classical computing, but rather a demonstrated ability of a quantum computer to process a problem (even a contrived one that is not practically useful) faster than a classical computer [4]. This is different from quantum advantage, which focuses more on the ability of quantum computers to solve real-world problems faster than classical computers.

tation to solve the problem of finding a minimum eigenvalue of a Hermitian matrix.

Knowing that we can model the Hamiltonian[2] of a given quantum system using a Hermitian matrix, this means we can use VQE to find the ground state energy (lowest-energy state) of that system since that ground state is represented by the Hamiltonian's lowest possible eigenvalue.

A quantum system's ground state, or lowest-energy state, provides valuable information about its properties which can be utilized in simulating its behavior, and simulations allow us to better understand, predict, and exploit the behavior of physical systems. Simulating a system, quantum or otherwise, requires an understanding of how that system naturally evolves, and the lowest-energy state of a quantum system is the state to which the system tends toward naturally. As such, the ground state encodes important information about the system's lowest-energy configurations and internal interactions that offer significant value for simulation.

A. Variational Quantum Algorithms (VQAs)

The VQE algorithm belongs to a larger class of Variational Quantum Algorithms (VQAs) which, as described by Cerezo et al. [11], are designed to classically optimize parameterized quantum circuits (PQCs) — that is, quantum circuits that a Benedetti et al. publication [12] describes as trainable machine learning models. More specifically, a VQA takes a machine learning approach to quantum circuit optimization in that it defines a reusable, fixed-structure PQC with unchanging quantum gate types and iteratively runs that circuit many times with varying gate parameters, where changed parameters affect gate behavior and thus the measured outputs of the circuit. As an example, a PQC might have a parameter that controls the strength of a coupling between two qubits or a parameter that controls the angle of a rotation gate. By iteratively adjusting these parameters (e.g., with a grid search, which we also see in hyperparameter optimization of machine learning models [13]), one can identify the specific set of parameter values that produce the best solution to the problem in question.

In the case of VQE, where the goal is to find the ground state of a quantum system, we aim to find parameter values that minimize the expectation value[3] of the quantum system's Hamiltonian. Subsection II-B explains the principle behind this in more detail.

[2]The Hamiltonian operator H of a quantum system is an operator representing the total energy of that system. The Hamiltonian is represented as a matrix, and the eigenvalues of that matrix represent the possible energy levels which that system can take on.

[3]Expectation values (EVs) essentially help us mathematically describe measurements in quantum systems; the expectation value of an observable is the average of all possible measurement outcomes for that observable weighted by their respective probabilities. The expectation value of the Hamiltonian calculated with some wave function ψ is denoted $\langle\psi|H|\psi\rangle$ or simply $\langle H\rangle_\psi$ [14].

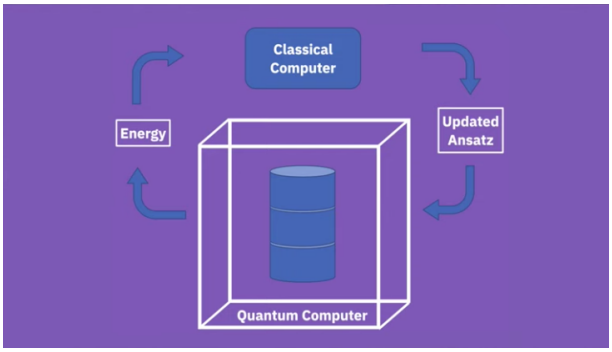

Fig. 1: A High-Level Look at the VQE Algorithm

B. The Variational Principle

The variational principle on which VQE is based states that for a given Hermitian matrix[4] H (e.g., the Hamiltonian of a quantum system), its expectation value calculated with a trial wave function ψ (denoted $\langle H\rangle_\psi$) must always be greater than or equal to its minimum eigenvalue λ_{min} where the minimum eigenvalue is equal to the ground state energy E_0 of the quantum system. This can be mathematically expressed as

$$\lambda_{\text{min}} \leq \langle H\rangle_\psi \implies E_0 \leq \langle H\rangle_\psi$$

We know that the Schrödinger equation describes how a given quantum system changes over time. With the time-independent version of that equation

$$H|\psi\rangle = E|\psi\rangle$$

the rate at which the system changes is constant and the solution to the equation consequently describes the allowed states of the system and their associated energy levels [15], which essentially equate to rules governing the system's behavior that can be used for simulating the system.

The idea with VQE is to start with an initial educated guess (called the "ansatz") for the trial wave function ψ, then use quantum computing to calculate the energy using that wave function, and then adjust that wave function using a classical optimizer, and repeat until the expectation value of the Hamiltonian H converges to a minimum. That minimum value, once found, must be the ground state energy E_0 of the quantum system since the variational principle guarantees that E_0 is the lower bound for the energy. A diagram of this flow is depicted in Figure 1. The key note here is that VQE uses quantum computing for calculating the energy but uses classical computation for optimizing variational parameters (the wave function).

[4]A Hermitian matrix is a complex square matrix that is equal to its own conjugate transpose, meaning the element in the i-th row and j-th column is equal to the complex conjugate of the element in the j-th row and i-th column, for all indices i and j.

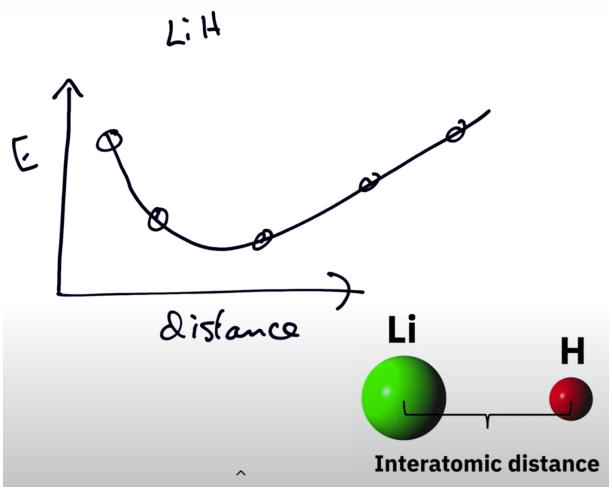

Fig. 2: Example Energy Profile of an LiH Molecule

C. How VQE Works

To understand how the VQE algorithm works, let us consider a specific, simple problem: finding the interatomic distance of a lithium hydride (LiH) molecule.

In this simple case, we can make the interatomic distance of LiH a parameter of our PQC, with the goal being to find the distance that results in the lowest energy measurement. Figure 2 depicts an approximation of the energy profile of the molecule. At each distance, VQE computes the lowest energy. After all of those energy computations, the distance that resulted in the lowest overall energy is the actual interatomic distance of the molecule.

The first thing we need in order to use VQE is an educated guess (an ansatz) for the trial wave function ψ of LiH. Note here that creating this initial ansatz requires specific domain knowledge of the problem; we need to encode our molecule into a quantum state (qubits) in order to use quantum computing to calculate its energy, which means we need to be very familiar with the molecule. This mapping of the molecule into the quantum wave function (i.e., into qubits) will take into account properties like its molecular geometry, its electronic orbitals, and its electron count.

Since we are interested in finding the molecule's interatomic distance (the unknown for this problem), we can parameterize the geometry property such that VQE will tell us which specific value of that property produces the lowest energy. First, though, we provide an educated guess for its value in our ansatz. Let our initial guess for the distance value be D_0.

Next, the quantum computer will execute a series of measurements using our initial ansatz with distance D_0 to calculate the energy of the LiH molecule.

After that, the energy measured is returned back to the classical optimizer which applies an update to the ansatz. Once the energy at this distance D_0 converges to a minimum value, VQE then moves to the next interatomic distance D_1 and repeats, as shown in Figure 1. The optimization approach can vary, but ultimately after the described iterations, the algorithm will identify a minimum energy level along with the parameters (the interatomic distance) that were used to produce that minimum. Those associated parameters reveal information about the state toward which that molecule (generally, the quantum system) naturally tends without external forces acting on it.

III. The Cirq Framework

The below section will provide information about the Cirq framework from Google and the motivation for using it and will subsequently provide a look at the implemented algorithm using that framework.

A. The Cirq Framework

Originally announced in July of 2018 at the International Workshop on Quantum Software and Quantum Machine Learning by Google's AI Quantum Team [16], Cirq is an open source framework for NISQ quantum computers that allows researchers to investigate whether these computers can solve real-world computational problems. One uses the Cirq framework via the "cirq" Python library [17] which offers useful abstractions for writing, optimizing, and running quantum circuits either on simulators or real NISQ computers.

Cirq was chosen as the VQE implementation tool for this project because of the extensive and clear documentation offered by Google Quantum AI on the use of their Python library for such an implementation [18], as well as the library's overall expressiveness and ease of use [17].

A supplemental implementation with Qiskit was also pursued in reference to their own documentation [19] but a number of issues were encountered due to rapid evolution of their "qiskit" SDK for Python (specifically with Qiskit Aqua and Qiskit Nature) [20] and its inconsistency with published resources and documentation. Nonetheless, apart from the implementation specifics, resources like IBM Quantum's YouTube video on the VQE algorithm provided useful theoretical insight on why the algorithm is useful and how to apply it to a specific problem like finding the interatomic distance of a molecule [21] as discussed in Subsection II-C.

IV. Implementation

The implementation of VQE discussed in this section focuses on the two-dimensional Ising model, a mathematical model of ferromagnetism[5] in statistical mechanics. Let us first explore the Ising model and the motivation for solving it with VQE, then we will look at the implementation using Python and the "cirq" library [17].

A. The Ising Model

Named after physicist Ernst Ising who solved the one-dimensional version in his 1924 thesis [23], the Ising model is simply a mathematical model of a system of particles which we can use to understand the behavior of materials at

[5]Ferromagnetism is an interesting physical phenomenon involving strong attraction between electrically uncharged materials. In the words of Barry Cipra [22], the term implies "spontaneous magnetization."

a microscopic level and, more importantly, how microscopic changes can lead to macroscopic results. More specifically, this model allows one to study how short-range, inter-particle interactions bring about and correlate with long-range system behavior [22].

In an Ising model composed of N particles, each particle's magnetic moment is allowed to point either "up" or "down", meaning each particle has two possible states [22].

In the two-dimensional version of the model specifically, on which this paper's VQE implementation focuses, and which was described in 1944 by Lars Onsager [24], the particles are arranged on a grid, and each particle interacts with its nearest neighbors (e.g., a particle $p_{i,j}$ in the i-th row and j-th column of the grid interacts with the particle $p_{i-1,j}$ above it, the particle $p_{i+1,j}$ below it, the particle $p_{i,j-1}$ to its left, and the particle $p_{i,j+1}$ to its right). The strength of that nearest neighbor interaction depends on the relative orientation of the particles' magnetic moments. The Hamiltonian of such a two-dimensional Ising model is as follows:

$$H = \sum_{<i,j} J_{i,j} Z_i Z_j + \sum_i h_i Z_i$$

where the left summation term captures those nearest-neighbor interactions for each particle in the grid and the second summation term captures interaction with an external magnetic field (outside of the model) perpendicular to the Z axis.

By mapping the two-dimensional Ising model to set of qubits and running VQE to minimize the expectation value of the described Hamiltonian H, we can find the ground state of the model, or the state toward which it naturally tends when not disturbed by external forces, and this ground state can tell us about a material's magnetic properties such that we can better simulate it.

B. Implementation with Python and "cirq"

Having explored the context of the two-dimensional Ising model and the motivation for finding its ground state with VQE in Subsection IV-A, we can now look at an implementation using Python and the "cirq" library [17].

1) Architecture: The implementation follows an object-orientated architecture, using a `VQEIsingSolver` class to encapsulate the logic of finding the Ising model ground state with VQE and a `Driver` class to drive the instantiation of `VQEIsingSolver` instances that solve[6] different instances of the Ising model, e.g., for varying grid sizes. Both of these classes inherit from a `Base` class for shared logging behavior.

2) VQEIsingSolver: The `VQEIsingSolver` class encapsulates the logic of the VQE algorithm specifically for solving an instance of the Ising model. The class itself as well as each method that it offers is heavily documented with explanations of how it relates to the logical flow of the VQE algorithm. The logic contained in the class is based on the Google Quantum AI documentation on VQE [18], but the implementation presented in this paper strips out the code from their documentation that is not needed, and reorganizes the important logic in a more expressive and object-oriented way such that the flow of the VQE algorithm is easier to understand.

In the final pages, I have included some of the key Python snippets from the implementation, which is available on GitHub [25].

Listing 1 depicts the head of the solver module and includes the constructor for the `VQEIsingSolver` class, which accepts a string to use for the name when logging, a boolean indicating whether to log verbosely, and an integer indicating the size of the grid to use for the Ising model.

Listing 2 includes the method used to build a random instance of the Ising model based on the grid size specified during instantiation. The problem instance is characterized by three attributes: the transverse field terms h, the interactions between vertically adjacent particles in the grid jr, and the interactions between horizontally adjacent particles in the grid jc. The instance is returned as a dictionary containing those three values, each of which are two-dimensional arrays, or lists of lists.

Listings 3 and 4 show the method used for preparing the paramaterized ansatz (the PQC). This method accepts three arguments, each of which are of type `sympy.Symbol`. Using `sympy.Symbol` objects as the parameters allows varying float values to be specified at runtime using a `sweep` combined with a `cirq.ParamResolver`, where the float values are rotation arguments passed to quantum gates. In essence, this means we are able to easily tune the values of these parameters to find the combination, via a grid search, that minimizes the energy of the system. You will notice in Listing 3 that creating the ansatz is broken into four steps. Each of the four steps has its own method that yields part of a sub-circuit but those implementations are not provided in this paper to avoid over-granularity. Listing 4 shows how those four step methods are combined. A problem instance is first created, the instance variables h, jr, and jc are set using that instance dictionary, a new circuit is created, and then the sub-circuits for ansatz preparation are appended to that circuit before it is returned to the caller.

Next, Listing 5 depicts the method used for calculating the expectation value of the system Hamiltonian given a `cirq.Result` object, which contains the measurements from running a simulation of the circuit. The expectation value here, as explained in Subsection II-A, is the average of all possible measurement outcomes for the Hamiltonian weighted by their respective probabilities. So, this method first calls the `.histogram(...)` method of the `cirq.Result` object and passes in a special-purpose function `self.energy_func(...)`[7] as the `fold_func` argument that allows us to pull all of the possible energy measurement outcomes as keys with their respective probabilities

[6] "Solving" an Ising model here means finding its ground state energy with VQE.

[7] The implementation of this function is not included in this paper.

as values. The method then uses that histogram dictionary to calculate and return the expectation value for the Hamiltonian.

Listing 6 contains the highest level method of the `VQEIsingSolver` class, which is the `simulate` method. This method drives the simulation and the flow of the VQE algorithm to find the ground state of the Ising model instance. It first creates a `cirq.Simulator` object to run the simulation. Then, it creates a square of `GridQubits`, which is a set of qubits onto which we can easily map our Ising model. Next, it initializes the three `sympy.Symbol` objects (called "alpha", "beta", and "gamma") to represent our three parameters whose values we want to optimize. It then prepares the initial ansatz circuit using the method from Listings 3 and 4. Once it has that ansatz circuit, the method also appends a measurement gate to the end of that circuit such that the qubits can all be measured at the end. It goes on to create a "sweep" which ultimately functions as a grid search, allowing the fixed-structure circuit we have defined to be executed over an equally spaced grid of many different parameter values. It very easily pulls the results from the grid search using `simulator.run_sweep`, passing in the circuit, the sweep parameters (all of the combinations of "alpha", "beta", and "gamma"), and the number of times to repeat each simulation which is pulled from a configuration file. After this point, it simply iterates over those results, calculates the expectation value of the Hamiltonian for each, and keeps track of the minimum value as well as the corresponding parameters "alpha", "beta", and "gamma" that produced it. It returns back to the caller a dictionary containing the minimum expectation value as well as its corresponding parameters.

Lastly, Listing 7 contains the fairly simple code for the `Driver` class. This class is housed in the main module which is the primary interface for running the implementation. The `Driver` class drives the instantiation of `VQEIsingSolver` instances and the running of simulations against Ising models of varying grid sizes. In the simple example provided, the driver iterates over the grid sizes of three and four, creating a `VQEIsingSolver` instance for each one, running the simulation, obtaining the results, and logging them into a file. A sample log for the `Driver` is provided in Listing 8. For example, we see that with a grid size of four (i.e. 16 particles), the minimum (ground state) energy level of the Ising model is -2.86, which corresponds to "alpha", "beta", and "gamma" respectively equaling 0.778, 0.222, and 0.667.

V. Summary

In summary, the Variational Quantum Eigensolver (VQE) algorithm is a ubiquitous, hybrid quantum-classical algorithm based on the variational principle that belongs to the class of Variational Quantum Algorithms (VQAs) and it has a broad range of use cases with significant implications for reducing the overall impact of quantum decoherence on computation accuracy. Through its hybrid design and its use of PQCs that basically function as trainable machine learning models, it allows one to identify ground states, or lowest-energy states, of quantum systems through classical optimizations of quantum computation results. By improving our understanding of ground states of quantum systems through VQE, we are able to learn how such systems tend to naturally evolve, which is a crucial part of being able to simulate them such that their behavior can be better understood and predicted. Where classical computers would struggle to solve optimization problems involving a large number of particles, and NISQ era quantum processors alone are too environmentally sensitive to be reliable, combining these two forms of computation into the hybrid VQE algorithm provides researchers with a unique advantage in pursuit of understanding the physical world.

References

[1] N. Tang, Y. Gritsenko, K. Kimura, S. Bhattacharjee, A. Sakai, M. Fu, H. Takeda, H. Man, K. Sugawara, Y. Matsumoto, and et al., "Spin–orbital liquid state and liquid–gas metamagnetic transition on a pyrochlore lattice," *Nature Physics*, 2022, accessed: 2022-12-03.

[2] D. Zhu, C. Chen, M. Yu, L. Shao, Y. Hu, C. Xin, M. Yeh, S. Ghosh, L. He, C. Reimer *et al.*, "Spectral control of nonclassical light pulses using an integrated thin-film lithium niobate modulator," *Light: Science & Applications*, vol. 11, no. 1, pp. 1–9, 2022.

[3] J. Preskill, "Quantum computing in the nisq era and beyond," *Quantum*, vol. 2, p. 79, 2018.

[4] K. Hartnett, "Quantum supremacy is coming: Here's what you should know," https://www.quantamagazine.org/quantum-supremacy-is-coming-heres-what-you-should-know-20190718/, 2019, accessed: 2022-12-03.

[5] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. Brandao, D. A. Buell *et al.*, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, 2019.

[6] H. Collins and C. Nay, "Ibm unveils 400 qubit-plus quantum processor and next-generation ibm quantum system two," https://tinyurl.com/ibm-osprey-2022-11-09, 2022, accessed: 2022-12-01.

[7] K. Kimball, "Announcing request for nominations for public-key post-quantum cryptographic algorithms," https://csrc.nist.gov/news/2016/public-key-post-quantum-cryptographic-algorithms, 2016, accessed: 2022-12-02.

[8] B. Johnson, T. Mittal, and J. Garcia, "Introducing new qiskit runtime capabilities — and how our clients are integrating them into their use," https://research.ibm.com/blog/qiskit-runtime-capabilities-integration, 2022, accessed: 2022-12-01.

[9] B. Sarma, S. Borah, A. Kani, and J. Twamley, "Accelerated motional cooling with deep reinforcement learning," *Physical Review Research*, vol. 4, no. 4, p. L042038, 2022, accessed: 2022-12-02.

[10] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, and J. L. O'brien, "A variational eigenvalue solver on a photonic quantum processor," *Nature communications*, vol. 5, no. 1, pp. 1–7, 2014, accessed: 2022-12-03.

[11] M. Cerezo, A. Arrasmith, R. Babbush, S. C. Benjamin, S. Endo, K. Fujii, J. R. McClean, K. Mitarai, X. Yuan, L. Cincio *et al.*, "Variational quantum algorithms," *Nature Reviews Physics*, vol. 3, no. 9, pp. 625–644, 2021, accessed: 2022-12-03.

[12] M. Benedetti, E. Lloyd, S. Sack, and M. Fiorentini, "Parameterized quantum circuits as machine learning models," *Quantum Science and Technology*, vol. 4, no. 4, p. 043001, 2019, accessed: 2022-12-03.

[13] P. Liashchynskyi and P. Liashchynskyi, "Grid search, random search, genetic algorithm: a big comparison for nas," *arXiv preprint arXiv:1912.06059*, 2019, accessed: 2022-12-03.

[14] P. M. Physics, "Expectation values in quantum mechanics," https://www.youtube.com/watch?v=NT1JX85MGA0, 2020, accessed: 2022-12-03.

[15] E. Schrodinger, "An undulatory theory of the mechanics of atoms and molecules," 1926, accessed: 2022-12-03.

[16] A. Ho and D. Bacon, "Announcing cirq: An open source framework for nisq algorithms," https://ai.googleblog.com/2018/07/announcing-cirq-open-source-framework.html, 2018, accessed: 2022-12-01.

[17] C. Developers, "Cirq," Apr. 2022, See full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors, Accessed: 2022-12-01. [Online]. Available: https://doi.org/10.5281/zenodo.6599601
[18] G. Q. A. Team, "Quantum variational algorithm," https://quantumai.google/cirq/experiments/variational_algorithm, 2022, accessed: 2022-12-01.
[19] Q. D. Team, "Vqe," https://qiskit.org/documentation/stubs/qiskit.algorithms.VQE.html#qiskit.algorithms.VQE, 2022, accessed: 2022-12-01.
[20] M. S. A. et al, "Qiskit: An open-source framework for quantum computing," 2021, See full list of authors on Github: https://github.com/Qiskit/qiskit, Accessed: 2022-12-02. [Online]. Available: https://doi.org/10.5281/zenodo.2573505
[21] Q. D. Team, "The variational quantum eigensolver — programming on quantum computers — coding with qiskit s2e4," https://www.youtube.com/watch?v=Z-A6G0WVI9w, 2020, accessed: 2022-12-02.
[22] B. A. Cipra, "An introduction to the ising model," *The American Mathematical Monthly*, vol. 94, no. 10, pp. 937–959, 1987.
[23] E. Ising, "Contribution to the theory of ferromagnetism," *Z. Phys*, vol. 31, no. 1, pp. 253–258, 1925, accessed: 2022-12-03.
[24] L. Onsager, "Crystal statistics. i. a two-dimensional model with an order-disorder transition," *Physical Review*, vol. 65, no. 3-4, p. 117, 1944, accessed: 2022-12-03.
[25] A. Hunt, "Analyzing and Implementing the Variational Quantum Eigensolver (VQE) Algorithm with Cirq," 12 2022.

```python
"""
solver.py

This module demonstrates use of the Variational Quantum Eigensolver (VQE)
algorithm to find the ground state of a 2D +/- Ising model with a transverse field. It uses Google's cirq library to create and simulate quantum circuit execution to handle the quantum portion of the VQE algorithm.
It is based on the Google Quantum AI documentation found here:
https://quantumai.google/cirq/experiments/variational_algorithm
But this module rearchitects the code they provide in a more expressive, object-oriented form, primarily to assist with understanding the logical flow of the VQE algorithm.

The Driver in main.py instantiates one of these solvers, passing in a grid size to use for the Ising model. Then the driver executes the solver.simulate() method to run the simulation and identify the minimum energy level (ground state energy) of a given instance of the Ising model. The simulate method of the solver returns a dictionary containing that minimum energy as well as the parameters that resulted in that minimum energy.
"""

import cirq
import random
import numpy as np
from sympy import Symbol
from base import Base
from conf import NUM_SIMULATION_REPETITIONS

class VQEIsingSolver(Base):
    """ VQEIsingSolver is a class that encapsulates logic for using VQE
    algorithm to solve an instance of the 2D +/- Ising problem. It inherits
    from Base for logging purposes. """

    def __init__(self,
        name: str = 'VQEIsingSolver', # name for logging purposes
        verbose: bool = False, # whether to use verbose logging
        ising_grid_size: int = 3): # size of the square model grid
        super().__init__(name, verbose)
        # Init the ising model grid size
        self.ising_grid_size = ising_grid_size

        self.h = None # 2D array (list of lists) representing transverse field terms
        self.jr = None # 2D array (list of lists) representing inter-row links
        self.jc = None # 2D array (list of lists) representing inter-column links

```

Listing 1: VQEIsingSolver Constructor

```
    def build_random_problem_instance(self):
        """
        We have a lot of freedom in how we define our variational ansatz.
        In this case, we will use a variation of a Quantum Approximate
        Optimization Algorithm (QAOA) technique to define an ansatz that is
        related to our specific Ising problem.

        We first need to identify how the problem instances will be represented
        (i.e. what are the parameters that we need to encode and optimize to find the
        ground state?). With our 2D +/- Ising model, these are the values J and h in the
        Hamiltonian definition:
        H = [ SUM_<i,j> { J_{i,j}*Z_i*Z_j } ] + [ SUM_i { h_i * Z_i } ]

        We represent both J and h as 2D arrays (lists of lists). So we
        use a function build_random_2d_array that generates a
        random 2D array of dimension rows x cols.

        Args:

        Returns:
        {
            'transverse_field_terms': list of lists representing transverse field terms
            'links_in_row': list of lists representing inter-row links in Ising model
            'links_in_col': list of lists representing inter-column links in Ising model
        }
        """

        def build_random_2d_array(rows, cols):
            """ Build a random 2D array (list of lists) comprising """
            return [[random.choice([+1, -1]) for _ in range(cols)] for _ in range(rows)]

        # build h, the transverse field terms
        h_transverse_field_terms = build_random_2d_array(
            rows=self.ising_grid_size, cols=self.ising_grid_size)
        # build j_r links within a row (there are rows - 1 inter-row links)
        jr_links_in_row = build_random_2d_array(
            rows=self.ising_grid_size- 1, cols=self.ising_grid_size)
        # links within a column ((there are col - 1 inter-col links))
        jc_links_in_col = build_random_2d_array(rows=self.ising_grid_size,
        ↪  cols=self.ising_grid_size - 1)
        return { 'transverse_field_terms': h_transverse_field_terms, 'links_in_row':
        ↪  jr_links_in_row, 'links_in_col': jc_links_in_col }


```

Listing 2: Method to build a random Ising model problem instance

```python
def initialize_ansatz(self, x_half_turns: Symbol = None,
    h_half_turns: Symbol = None, j_half_turns: Symbol = None):
    """ First step of VQE is creating a parameterized (via sympy) "ansatz"  that
    ↪  essentially encodes the problem in question into qubits. This ansatz is an
    ↪  educated guess about the parameters for our Parameterized Quantum Circuit (PQC).
    Args:
    x_half_turns - sympy.Symbol - parameterized (non-static) number of half turns to apply
↪   with X rotation gate in step 4. Value range driven by sympy.
    h_half_turns - sympy.Symbol - parameterized (non-static) number of half turns to apply
↪   with Z rotation gate in step 2. Value range driven by sympy.
    j_half_turns - sympy.Symbol - parameterized (non-static) number of rotations about
↪   |11> conditioned on jr, jc to apply in step 3. Value range driven by sympy.

    Ansatz  will consist of two sub-circuits.
    Sub-circuit 1 is is step one.
    Sub-circuit 2 is steps 2-4.

    Step 1. Apply an initial mixing step that puts all qubits into the
    |+> = 1/sqrt(2) (|0>+|1>) state. (i.e., a superposition achieved with Hadamard gate)
    Step 2. Apply a cirq.ZPowGate for the same parameter for all qubits where
    the transverse field term h is +1
    Step 3. Apply a cirq.CZPowGate for the same parameter between all qubits where the
    coupling field term J is +1. If the field is -1, apply cirq.CZPowGate conjugated by X
    gates on all qubits.
    Step 4. Apply an cirq.XPowGate for the same parameter for all qubits.

    Returns:
    cirq.Circuit - the ansatz / educated initial guess with initial parameter values
    """

    def step_one():
        ...
    def step_two():
        ...
    def step_three():
        ...
    def step_four():
        ...

```

Listing 3: Method to prepare the parameterized ansatz circuit

```
    ...
    def step_one_wrapper():
        """ This function wraps the step one sub-circuit such that we can append the
        ↪  sub-circuit to a cirq.Circuit.
        Yields:
        sub-circuit handling step one of initializing ansatz.
        """
        yield step_one()

    def step_two_three_four_wrapper(h: list = [], jr: list = [], jc: list = [],
        x_half_turns: float = 0.1, h_half_turns: float = 0.1, j_half_turns: float = 0.1):
        """ This function wraps steps 2, 3, and 4 such that the result can be appended as
        a single sub-circuit to the parent cirq.Circuit.

        Args:
        h: list - 2D array (list of lists) representing transverse field terms in Ising
↪   model
        jr: list - 2D array (list of lists) representing inter-row links in Ising model
        jc: list - 2D array (list of lists) representing inter-column links in Ising model

        h_half_turns - float - number of half turns to apply with Z rotation gate in step
↪   2
        j_half_turns - float - number of rotations about |11> conditioned on jr, jc to
↪   apply in step 3
        x_half_turns - float - number of half turns to apply with X rotation gate in step
↪   4

        Yields:
        sub-circuit containing steps 2, 3, and 4 for initializing ansatz.
        """
        yield step_two(h=h, num_half_turns=h_half_turns)
        yield step_three(jr=jr, jc=jc, num_half_turns=j_half_turns)
        yield step_four(num_half_turns=x_half_turns)

    # First build a random problem instance
    problem_instance = self.build_random_problem_instance()
    self.h = problem_instance['transverse_field_terms']
    self.jr = problem_instance['links_in_row']
    self.jc = problem_instance['links_in_col']

    # Initialize a circuit
    ansatz_circuit = cirq.Circuit()

    # Append the step one sub-circuit
    ansatz_circuit.append(step_one_wrapper())
    # Append the steps 2,3,4 sub-circuit
    ansatz_circuit.append(step_two_three_four_wrapper(h=self.h, jr=self.jr, jc=self.jc,
    ↪  x_half_turns=x_half_turns, h_half_turns=h_half_turns, j_half_turns=j_half_turns))
    # Here we see that we have chosen particular parameter values 0.1, 0.2, 0.3
    self.info(ansatz_circuit)
    return ansatz_circuit

```

Listing 4: Method to prepare the parameterized ansatz circuit (continued)

```python
    def calculate_expectation_value(self, result: cirq.Result = None):
        """ Calculate the expectation value of the Hamiltonian from the results of
        the measurements, where the expectation value is the average of all
        possible measurement outcomes for an observable (the Hamiltonian) weighted
        by their respective probabilities.

        We first obtain the histogram of the measured energies, where the resulting
        Counter uses the energy as the key and the probability as the value.

        We then take SUM_i [energy_i * energyprobability_i] and divide that
        sum by the total number of simulation repetitions to get the average, i.e.,
        the expectation value.

        Args:

        result: cirq.Result - result containing all the measurements for a given
        simulation run.

        Returns:
        expectation value - float - expectation value of the Hamiltonian with
        these measurements
        """
        energies_histogram = result.histogram(
            key='x',
            # self.energy_func is another custom method not included in this paper that
            ↪  allows us to pull measured energies directly from this histogram result as
            ↪  keys, which we can then use to calculate the expectation value of H
            fold_func=self.energy_func(
                h=self.h,
                jr=self.jr,
                jc=self.jc
            )
        )
        return np.sum([k * v for k,v in energies_histogram.items()]) / result.repetitions

```

Listing 5: Method to calculate the expectation value of Hamiltonian given measurement results

```
    def simulate(self):
        """
        Run simulation to find minimum objective value of Hamiltonian (ground state energy
    ↪   of the Ising model). This can be done by parameterizing the ansatz circuit.
        Many of the values in the gate sets can, instead of being specified by a static
    ↪   float, be specified by a symply.Symbol which an be substituted for a value specified
    ↪   at execution time
        In essence, paramaterizing our values for our Parameterized Quantum Circuit (PQC)
    ↪   can be handled by passing sympy.Symbol objects rather than passing static float
    ↪   values.
        """
        # Create a simulator with cirq
        simulator = cirq.Simulator()

        # Create a square grid / lattice of qubits (into which we can encode
        # our ising problem in order to use VQE)
        qubits_grid = self.create_square_qubit_grid()

        # These three variables represent gate rotation params which are the parameters we
        ↪  are tuning for the optimization.
        alpha, beta, gamma = (Symbol(_) for _ in ['alpha', 'beta', 'gamma'])
        # Parameterize the circuit gates with sympy Symbols as the half turns
        # The parameter vals are specified at runtime using a cirq.ParamResolver,
        # which is just a dictionary from Symbol keys to runtime values.
        ansatz_circuit = self.initialize_ansatz(x_half_turns=alpha, h_half_turns=beta,
        ↪  j_half_turns=gamma)
        ansatz_circuit.append(cirq.measure(*qubits_grid, key='x'))

        # Use a Cirq sweep, which is a collection of parameter resolvers to evaluate our
        ↪  circuit over an equally spaced grid of many parameter values (Grid Search used
        ↪  in ML) We can easily create this using cirq.LinSpace.

        # Finding the minimum via Grid Search optimization
        sweep_size = 10
        sweep = (cirq.Linspace(key='alpha', start=0.0, stop=1.0, length=sweep_size)
                * cirq.Linspace(key='beta', start=0.0, stop=1.0, length=sweep_size)
                * cirq.Linspace(key='gamma', start=0.0, stop=1.0, length=sweep_size))
        results = simulator.run_sweep(ansatz_circuit, params=sweep,
        ↪  repetitions=NUM_SIMULATION_REPETITIONS)
        min, min_params = None, None
        for result in results:
            value = self.calculate_expectation_value(result)
            if min is None or value < min:
                min = value
                min_params = result.params
        self.info(f'Minimum objective value is {min}. Params to produce minimum are:
        ↪  {min_params}')
        return { 'min_energy': min, 'min_params': min_params }

```

Listing 6: Method to run the high-level VQE flow to find the Ising model ground state

```python
"""
main.py

This file contains a Driver that is used to actually drive the
instantiation of VQEIsingSolvers and the execution of their .simulate() methods.
"""
from base import Base
from vqe_ising_solver.solver import VQEIsingSolver

class Driver(Base):
    def __init__(self, name: str = 'Driver', verbose: bool = False):
        super().__init__(name, verbose)

    def run_ising_solver(self, ising_grid_size: int = 3):
        """ Run an Ising solver that uses VQE to solve Ising problem
        (i.e. finds ground state energy of Ising model that is
        ising_grid_size x ising_grid_size) in dimension """
        self.info(
            f'Creating VQE Ising Solver for Ising model with '
            f'square grid size {ising_grid_size}x{ising_grid_size}')
        solver =
        ↪  VQEIsingSolver(name=f'VQEIsingSolver-{ising_grid_size}x{ising_grid_size}')
        result = solver.simulate()
        self.info(
            f'With a grid size of {ising_grid_size}, the minimum '
            f'energy level is {result["min_energy"]} corresponding '
            f'to the parameters {result["min_params"]}')
        return result

if __name__ == "__main__":
    driver = Driver()
    for grid_size in range(3,5):
        result = driver.run_ising_solver(ising_grid_size=grid_size)

```

Listing 7: Driver class that drives instantiation of VQEIsingSolver objects and simulation execution

```
[Driver - main.py:16 - run_ising_solver() ] Creating VQE Ising Solver for Ising model with
↪  square grid size 3x3
[Driver - main.py:21 - run_ising_solver() ] With a grid size of 3, the minimum energy
↪  level is -2.74 corresponding to the parameters cirq.ParamResolver({'alpha':
↪  0.6666666666666666, 'beta': 0.0, 'gamma': 0.3333333333333333})
[Driver - main.py:16 - run_ising_solver() ] Creating VQE Ising Solver for Ising model with
↪  square grid size 4x4
[Driver - main.py:21 - run_ising_solver() ] With a grid size of 4, the minimum energy
↪  level is -2.86 corresponding to the parameters cirq.ParamResolver({'alpha':
↪  0.7777777777777778, 'beta': 0.2222222222222222, 'gamma': 0.6666666666666666})

```

Listing 8: Sample Log Output from the Driver