QOSF Task 4: A Variational Quantum Eigensolver from Scratch

Satwik Nandala

September 26, 2020

1 Introduction

The Variational Quantum Eigensolver (VQE) is an algorithm that can be used to find the upper bound on the lowest eigenvalue of a (usually) large matrix [1]. Hopefully, the upper bound will equal the lowest eigenvalue. For us, this matrix will be the Hamilontian for a quantum system, such as a molecule, and finding the lowest eigenvalue and eigenvector represents finding the ground state of the system. For this task, we will use the Cirq Python framework [2] for building, simulating, and running quantum circuits. The Hamiltonian we will be running the algorithm on is

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

2 Algorithm

Each subsection that follows corresponds to a block of code in the Jupyter notebook:

`VQE_QOSF_Task4_Satwik_Nandala.ipynb`

included in the Github repository. The comments in each code block will designate the corresponding subsection, for the reader to follow along.

2.1 Decomposing the Hamiltonian

The first step in the algorithm is to decompose the Hamiltonian into the sum of Pauli matrices. The task hinted that matrices that would be used are from the set {XX, YY, ZZ, II} with weights from the set {-1, -1/2, 0, 1/2, 1}. By inspection, the Hamiltonian can be decomposed like so:

$$H = \frac{ZZ + II - XX - YY}{2}$$

2.2 The Ansatz & Creating the VQE Circuit Layers

The next step is to choose an ansatz and define the relevant layers. The ansatz is basically a parameterizable subcircuit can prepare a range of different quantum states. At risk of highly oversimplifying this, a rule of thumb is that the larger the ansatz (one with more gates and paremeters), the larger chance that we will generate the state that results in the minimum eigenvalue of the Hamiltonian. The tradeoff here though is that with each added parameter, we are exponentially increasing the search space, as well as decreasing the overall circuit fidelity. The ansatz we will begin with is

$$(R_x(\theta) \otimes I) \otimes CX \otimes (H \otimes I)$$

where the angle of rotation for the R_x gate will be our single parameter.
Two more layers we need to define are the transformation layers from X and Y bases to Z basis. The functions

```
XX_correction
```

and

```
YY_correction
```

take care of this, so that even if we have Hamiltonian terms X or Y, we can correctly measure in the Z basis, which is the usual computational basis for real world quantum computers.

2.3 Creating and Simulating the Circuits

The goal now is to iterate through each of the four terms in the Hamiltionian decomposition from section 2.1 and do the following for each:

1. Create the ansatz + correction circuit
2. Calculate the expectation values for a range of θ values
3. Sum up all the expectation values
4. Find the minimum of the resulting curve

Cirq makes it very easy to do these things. Lines 22 - 32 in the corresponding Jupyter cell create the circuit using ckt.append() with the layers that were defined in section 2.2. Printing them outputs the following:

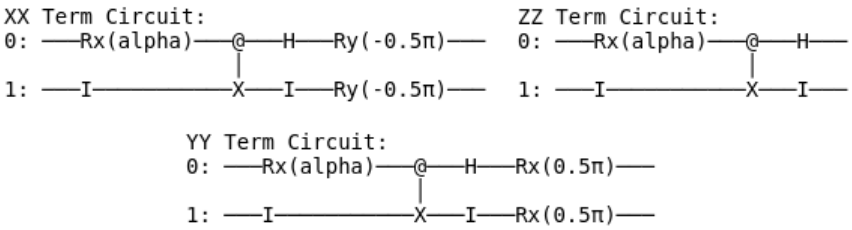

Figure 1: Circuits for each non-trivial term

We don't need to make a circuit for the II term, since we know the expectation for that operator is a constant for all values of theta (1).

For step 2, using Sympy, we can create a symbol α (line 12) and pass it to our circuit layers. We can also create a sweep over the symbol using cirq.Linspace (line 13) and execute the sweep using simulator.simulate_sweep() (line 34).

In lines 34-41, we get the results of our sweep and calculate expectation values using $\langle\Psi|H_n|\Psi\rangle$ where each H_n is a term from the decomposed Hamiltonian. We also scale our results accordingly and add them to a Python dictionary.

2.4 Plotting and Finding the Lower Bound

We can plot the expectations for each term in the Hamiltonian:

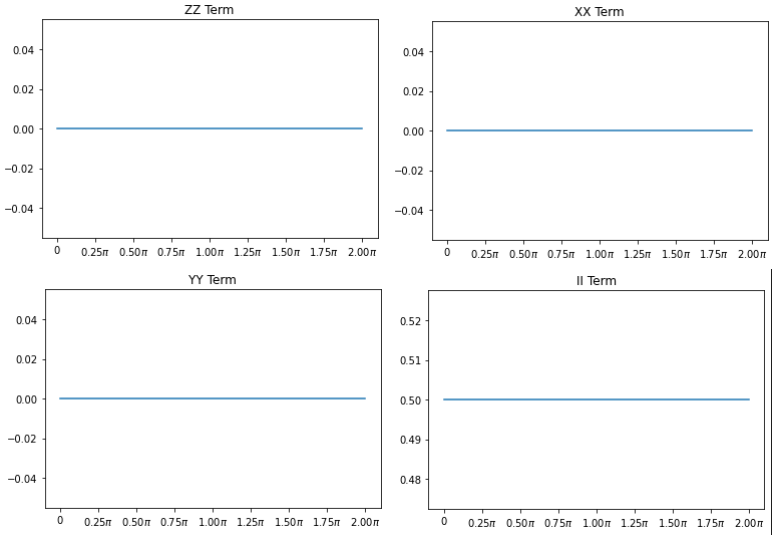

Figure 2: Plot of Expectation vs α for each Hamiltonian Term

We sum these and find the minimum, which is obvious from the plots.

```
The upper bound for the lowest eigenvalue for the Hamilotnian is: 0.5
The actual lowest eigenvalue for the given Hamilotnian is: -1.0
```

It seems that our lower bound isn't equal to the ground state! Furthermore, our ansatz didn't look like it helped us very much, since all the expectation values are zero! Let's think about why that might be.

2.5 Analysis

This was our ansatz:

$$(R_x(\theta) \otimes I) \otimes CX \otimes (H \otimes I)$$

Let us consider what states we are exploring with our ansatz. Note that the first qubit has an R_x and H gate that directly affect it. An R_x keeps the state on the XY plane, and a Hadamard is a 90° rotation about the Y axis, then a 180° rotation about the X axis. Between these two operations and a CX gate, much of the Bloch sphere is still out of reach for both qubits. It would be nice if we could express more states - an easy way to do that is to introduce some paramaterized R_y gates [3].

3 A New Ansatz

Let us add some R_y gates to our ansatz to make things more interesting and expand the range of the states we can visit. We'll use three parameters instead of just one now. We can try these new ansatzes:

$$(R_x(\alpha) \otimes I) \otimes CX \otimes (R_y(\beta) \otimes I)$$
$$(R_x(\alpha) \otimes R_y(\beta) \otimes CX \otimes (R_y(\gamma) \otimes I)$$
$$(R_y(\alpha) \otimes R_y(\beta) \otimes CX \otimes (R_y(\gamma) \otimes I)$$

Finding the expectations, summing them, and finding the minimum, we get very close to 0 for each ansatz. We only show one here, but the reader can verify the other two in the Jupyter notebook.

```
calc_expectations_all(qubits, ansatz_RXI_CX_RYI, angles)
print(np.amin(sum(expectations.values())))
```

```
0.03015366843805878
```

It's an improvement over the 1/2 we got previously, but still not equal to the true eigenvalue we're looking for of -1. Let's add a fourth parameter.

3.1 A 4th Parameter

Our new ansatz is

$$(R_x(\alpha) \otimes R_x(\beta) \otimes CX \otimes (R_y(\gamma) \otimes R_y(\delta)) \otimes CX$$

```
calc_expectations_all(qubits, ansatz_RXRX_CX_RYRY_CX, angles)
print(np.amin(sum(expectations.values())))
```

```
0.0004580881253091196
```

It seems we didn't do much better, even with a fourth parameter! This is harder than it seems.

4 Conclusion

The natural question from here is why we aren't getting a better result with another parameter. There are two main factors for this:

1. Choosing ansatz is not easy. We didn't use any R_z gates, and it's possible we just don't have enough parameters.
2. Each parameter sweep is only over 10 values from the interval 0 to 2π. The algorithm would take a very long time to run if we tried to sweep 100 values for each parameter (that would be $100^4 = 100{,}000{,}000$ values!)

The best improvement we can make for the future is to use an optimization algorithm like gradient descent instead a brute force search to see if we can get closer to the true minimum. This will be added in the future.

References

[1] Michał Stechły. Variational quantum eigensolver explained, 2018.

[2] The Cirq Contributors. *Cirq, a python framework for creating, editing, and invoking Noisy Intermediate Scale Quantum (NISQ) circuits.*

[3] Sukin Sim, Peter D. Johnson, and Alan Aspuru-Guzik. Expressibility and entangling capability of parameterized quantum circuits for hybrid quantum-classical algorithms, 2019.