Python Simulations of a Quantum Computer

JT Turner and Ben Alexander
Haverford College
(Dated: April 28, 2023)

1. INTRODUCTION

For this project, we chose to investigate quantum computing as this technology is at the forefront of research in many fields. Quantum computing is has applications spanning various areas of physics and computer science. We sought to apply our knowledge of quantum mechanics, and in particular matrix representations thereof to quantum computing problems. Our goal for this project was to simulate a quantum computer using python. We used this to simulation simulate various algorithms. Ultimately, our efforts were mainly focused on implementing Grover's algorithm due to its several potential uses.

2. CODE STRUCTURE

This code is built on the fundamental matrix mathematics used to represent quantum mechanics. While our final goal was to implement Grover's algorithm, we started by figuring out how to represent and measure states, then progressed to single particle gates. Once that was completed, we worked the multi-particle CNOT gate. This gave us a group of gates proven to be sufficient to construct any quantum circuit [3]. Finally, we built the circuits necessary to run Grover's algorithm. Since the focus was learning how to make the system, we did not rely on pre-built packages such as Qiskit. We also did not focus on efficiency. This lead to choices like choosing NumPy over a module with sparse matrix support. For these reasons, this code is particularly slow for systems with many particles. Additional particles will result in an exponential increase in the dimensions of the matrices used and hence severely harm run-time.

This section will cover how we coded each step leading up to implementing complex algorithms. Our program also includes some helper functions to make the code easier to read, although we don't always delve into the details of their construction here.

2.1. Representing States

To represent states in our code, we decided to use the computational basis. This mean that a state like $|01\rangle$ would represent particle 0 being in state $|0\rangle$ and particle 1 in state $|1\rangle$.

Note that for an m particle system, this meant there were $N = 2^m$ possible basis states. This relationship is one of the fundamental advantages of quantum computing. In our code, we represented this with a NumPy array vector of length N, with the ability to represent an arbitrary state in the computational basis.

We also created a function, `generate_basis_state()` to create the vector representing any basis state in the computational basis. We created initial states using this function to improve readability.

2.2. Measuring States

The function `measure_state()` selects an option for the state based on the weighted probability of each possible acceptable value. The probabilities are calculated by multiplying the state times its conjugate. A value is then selected with `np.random.choice()` based on the weighted probabilities. The value determined by `np.random.choice()` is then returned to the user. This was the final step in every quantum algorithm we implemented.

2.3. Single Particle Operators

Our implementation of basic operators relied on the ability of the NumPy module to handle matrices. We sought to create function that would apply various single particle gates to a given particle. In order to accomplish this, we created a function, `single_particle_op_mat()`, that would construct the $N \times N$ matrix needed to represent such an operation; recall N refers to the total number of basis states.

The matrix multiplication was accomplished through direct product with the identity (NumPy's `kron()` method). As an example of this process, consider attempting to represent applying the operator $\hat{U}_x$ in a four particle system. In order to apply it to the second particle in the system, our matrix representation would be

$$\hat{I} \otimes \hat{U}_x \otimes \hat{I} \otimes \hat{I}.$$

Here the position of the operator $\hat{U}_x$ dictates which particle it is being applied to, while the number of identity operators is determined by the size of the system.

Such an approach was used for the single particle operators $\hat{U}_x$, $\hat{U}_y$, $\hat{U}_z$, and $\hat{U}_H$. Although not all of these were used throughout the code, we thought it important to define the behavior of these fundamental operations – recall that they can in principle be used to construct any unitary operator [3].

2.4. Two Particle Operator: CNOT

In comparison to the single particle operator, the CNOT operator was far less straightforward to implement. Recall that the we might represent CNOT operating on a two particle system (control particle 0, target particle 1) with the matrix

$$\hat{U}_{CNOT} \rightarrow \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

If we are satisfied with maintaining these same target and control particles, we may use the same approach as for single particle operators to expand this to an m particle system; for example, a four particle system would would be represented by

$$\hat{U}_{CNOT} \otimes \hat{I} \otimes \hat{I}.$$

However, this approach has a limit in that it does not allow us to specify which particle is target and which is control. It is particularly difficult if the particles are not adjacent to each other in index (for example, particle 0 as control and particle 2 as target). We solve this by using a change of basis – that is, if we simply temporally represent our system in a way where the particle 0 is the control and particle 1 is the target, implementation of the CNOT operator is reduced to the above case.

The new basis was constructed by re-ordering the standard basis states. We observed that swapping associated bits in the binary labels for the basis states allowed us to determine which states needed to be swapped. After reordering the states, it was straightforward to construct change of basis matrices using linear algebra.

3. DEUTSCH JOZSA ALGORITHM

3.1. Background

The Deutsch Jozsa algorithm [1, 2] is a compelling demonstration of the efficiency of a quantum computer. This algorithm considers a function that takes in some even number of qubits and returns either zero or one. The function may be either even, meaning exactly half the outputs are 0 and half are 1, or constant, meaning all outputs are the same. The task of the algorithm is to classify the function.

In a quantum circuit diagram, this problem may be represented as shown in figure 1. Here y is an extra bit that encodes the result of f, while $x_1, ..., x_n$ are input bits.

While we will not delve into a complete derivation here, our classwork [6] is sufficient to show that one can obtain the system's behavior by applying a Hadamard gate to all x particles on both sides of the circuit. This begins

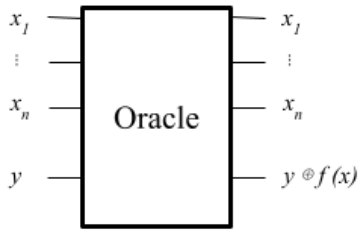

FIG. 1: Quantum circuit diagram for Deutsch Jozsa algorithm

the system in a uniform superposition of all computational basis states on the left. The superposition allows measurement of x_1 to determine the entire system's behavior.

3.2. Implementation

The Deutsch Jozsa Algorithm function takes in two parameters, the oracle and the number of particles to be input (from our class discussion these are $x_1, ..., x_n$). We include an extra particle in our calculation to fulfill the role of the y particle, although we don't care about how it measures. We then generate a uniform superposition of all possible input states using Hadamard gates. The state is put through the oracle and measured, with results returned to the user.

We've constructed three example oracles in the code, including both constant oracles and one potential even oracle that applies CNOT gates to the y particle as the target with half of the x particles as controls.

4. GROVER'S ALGORITHM

4.1. Background

Grover's algorithm [4] is a search method that reduces the number of searches necessary to find a given item in an unstructured list. Classically, given a list of n items,

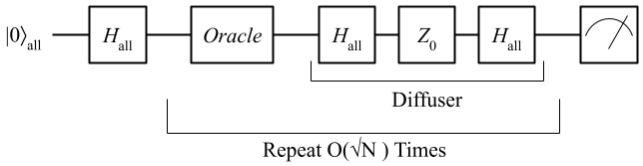

FIG. 2: The circuit representation of Grover's Algorithm, inspired by a figure in [5]

a specific item will be found in an unstructured search after an average of $\frac{n}{2}$ searches, an $O(n)$ algorithm. However, Grover's algorithm reduces this to $O(\sqrt{n})$ by taking

advantage of quantum superposition. A visual overview of the algorithm is shown in figure 2. We initialize the system in the state $|0\rangle$ and apply a Hadamard gate to each particle to create a uniform superposition of states. Then, the oracle [7] flips the phase of the desired result, identifying it from the other elements in the search. Finally, the diffuser amplifies the out-of-phase result compared to the other states.

This diffuser is constructed as shown in figure 2. Here H_{all} represents applying a Hadamard gate to all particles and $\hat{\underline{Z}}_0$ is the partial diffuser gate defined in [7], taking the matrix form

$$\hat{\underline{Z}}_0 \rightarrow \begin{bmatrix} 1 & & & \\ & -1 & & \\ & & -1 & \\ & & & \ddots \end{bmatrix}.$$

However, the diffuser only increases the relative amplitude of the desired output, but does not ensure the particle is in that state. Therefore, to achieve a high likelihood of success, the oracle and diffuser combination must be repeated $O(\sqrt{N})$ times [4]. These repetitions do not guarantee that the algorithm will find the correct state, but ensure a high probability of this occurring. If the answer is incorrect, Grover's algorithm may be repeated until successful.

4.2. Implementation

We designed a function to perform Grover's algorithm within our simulation. The function takes in an oracle, representing the problem to be solve and a number of particles to use. It then creates a uniform superposition of basis states. It does this by initializing the state $|0\rangle$ and applying an Hadamard gate to each particle.

To proceed with the algorithm, our code needed to represent the oracle. Recall that the oracle identifies the state we are looking for by flipping its sign. A possible oracle might have a matrix representation

$$\hat{\underline{U}}_{oracle} \rightarrow \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & -1 & \\ & & & \ddots \end{bmatrix}.$$

With the -1 representing that this oracle has an "answer" that is the third element. While it is possible to construct this oracle using the basic circuit components we defined in section 2.1, we did not believe it would be productive from a coding or run-time perspective. Instead, we implemented the oracle using this matrix representation.

We also implemented the $\hat{\underline{Z}}_0$ gate from the above section in a similar manner, using its matrix representation rather than constructing it from basic gates. Having the component Hadamard and $\hat{\underline{Z}}_0$ gates, we were thus able to code a function to perform the functionality of the diffuser.

In order to count the operations performed, we implemented a counter. We added a number equal to the number of particles in the system to the count each time we performed the combined oracle and diffuser operation. This was to model the fact that this combination of operators has a physical complexity that scales linearly with the number of particles. In measuring operations, we care most about the order of growth, any constant scaling of that number is de-emphasized.

4.3. Results

To test the effectiveness of the Grover's algorithm implementation, we compared it to the classical version of finding an element in a list. In particular, we wanted to see how the number of iterations required scaled with the number of particles present. This was accomplished by running both Grover's algorithm and a classical equivalent to find elements in lists of $N = 4, 8, 16, 32, 64$ and 128 elements. This corresponded to $m = 2, 3, ...7$ particles for Grover's algorithm. Target elements in the list were randomly generated for a total of 20 trials at each list size. Results are displayed in figures 3 and 4.

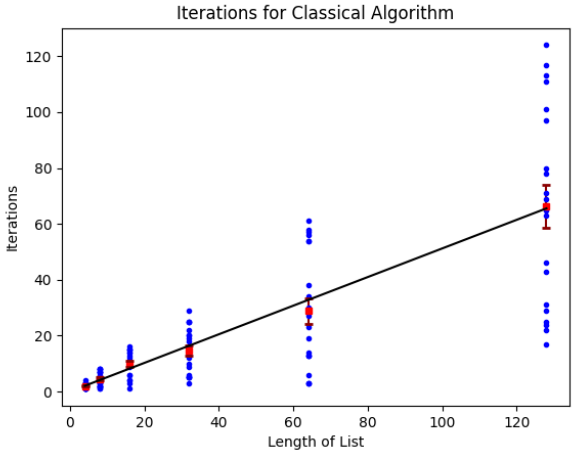

FIG. 3: The results of the classical implementation for $m = 2$ to $m = 7$. Blue is the number of iterations needed to find the item during each attempt. Red represents the average number of iterations for each value of m. In black, we see a linear fit to the average data.

Examining these figures, we can see significant differences between the results for the two algorithms. To begin with, there is much greater variation in the data for the classical algorithm. This is because the classical algorithm stops after finding the element in the list. In contrast, the efficiency of Grover's algorithm is completely independent of the position of the element in the

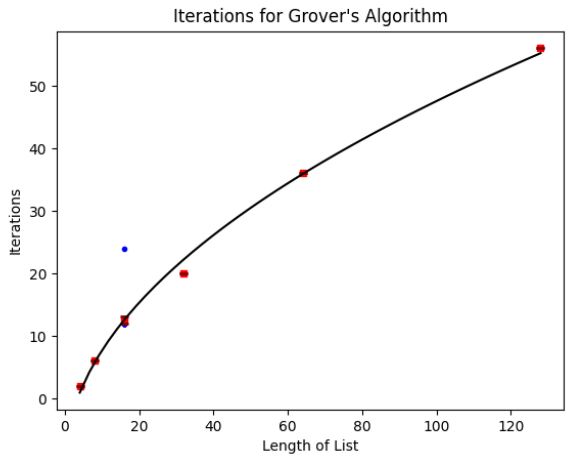

FIG. 4: The results of the Grover's algorithm implementation for $m = 2$ to $m = 7$. Blue is the number of iterations needed to find the item during each attempt. Red represents the average number of iterations for each value of m. In black, we see the average data fit to the function $a\sqrt{x} + b$.

list. Most of the time, in fact, it requires the same number of iterations for a list of a given size. The one data point not centered at its average in figure 4 is a result of Grover's algorithm failing on its first attempt.

As expected, Grover's algorithm scales more efficiently than the classical algorithm. Step count in the classical implementation scales linearly with step size. In contrast, the scaling is sublinear for Grover's algorithm. In figure 4, we have fit the data to a function of power $\frac{1}{2}$, as predicted by Grover [4].

The one downside to our implementation of Grover's algorithm is the algorithm took significantly more computational time than the classical method. However, this is a physical consequence of the fact that we we don't have a quantum computer.

Ultimately, these results are computational evidence that Grover's algorithm is capable of performing significantly better than classical alternatives, with sublinear scaling with the number of elements in a list.

Appendix A: Python Code

Our code files are available at https://github.com/JT-Turner-2/Quantum_Mech.

[1] Collins, D., Kim, K. W., & Holton, W. C. 1998, Phys. Rev. A, 58, R1633, doi: 10.1103/PhysRevA.58.R1633

[2] Deutsch, D., & Jozsa, R. 1992, Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences, 439, 553, doi: 10.1098/rspa.1992.0167

[3] DiVincenzo, D. P. 1995, Phys. Rev. A, 51, 1015, doi: 10.1103/PhysRevA.51.1015

[4] Grover, L. K. 1996, A fast quantum mechanical algorithm for database search. https://arxiv.org/abs/quant-ph/9605043

[5] Quantum, I. 2023, Grover's Algorithm. https://quantum-computing.ibm.com/composer/docs/iqx/guide/grovers-algorithm

[6] Smith, W. 2023, Lecture on Deutsch Jozsa Algorithm in Haverford Physics 302, Haverford College

[7] Wright, J. 2015, Lecture 4: Grover's Algorithm. https://www.cs.cmu.edu/~odonnell/quantum15/lecture04.pdf