Simulation of a Circuit-Model Quantum Computer in Python

Isaac López Agudo, Will Brown,
Andrew Ceniccola, Zihe Fang,
Bradley Harvey, Hattie Lord,
Ewen O'Donnell

Quantum Computing Project
University of Edinburgh
10th April 2026

Abstract

This report explores two quantum algorithms: Grover's search algorithm and Shor's prime factorisation algorithm. We discuss the motivation, application, and implementation of these algorithms through a circuit-model simulation of a quantum computer in Python, using suitable examples to illustrate how these algorithms work.

Contents

Chapter 1

Introduction

Quantum computation is based upon the theory of quantum mechanics, which concerns the actions and interactions of subatomic particles. At a microscopic scale, these particles can not be measured with certainty. Well-known quantities such as position have little meaning when dealing with such systems.For example, Young's double slit experiment showed the bizarre nature of fundamental particles, where the mere act of taking an observation is enough to disturb the system and drastically change the output. Another example is entanglement where (oversimplifying) a measurement on one such particle will collapse the state of the other without needing to observe the second particle. Due to this fundamental difference in behaviour, quantum systems can be exploited to perform classical computations at much higher speeds. The difficultly, currently, is the hardware required to build such a quantum computer. However, the action of a quantum computer is much simpler to explore [1, 2].

The theory of quantum computation is, to some extent, divorced from the physical theory of quantum mechanics. One can understand, implement, and contribute to the field of quantum computation with very little knowledge of physics at all. The only requirement is a basic understanding of finite vector spaces over the complex numbers. The simplification to a collection of two-state systems—the qubit—greatly simplifies the mathematics even further. The simulations used within this report are based on this, implementing quantum states and gates as familiar vectors and matrices acting on each other.

In this report, we discuss and simulate two of the more popular quantum computation algorithms. The first, Grover's algorithm, is the quantum mechanical analogue of a classical search algorithm, which has the potential to find an element of an array significantly quicker than a classical computer. The second, Shor's algorithm, tackles the 'factoring problem': decomposing arbitrarily large pseudo-prime numbers into a product of two primes. This is a difficult problem by classical means; this fact alone is the reason why prime factorisation forms the basis of information encryption. In theory, Shor's algorithm presents a faster way to break these types of encryption when used on a quantum system. However, in reality, no-one has been able to do this effectively with more than a few quantum bits [3].

Chapter 2

Simulating a Quantum Computer in Python

2.1 Qubits

2.1.1 Bit and Qubit

The quantum computer use qubits to store and represent data. These are the quantum counterpart of the traditional 'bit' used in standard digital computers. To understand what a qubit is, one must first begin with the idea of bits.

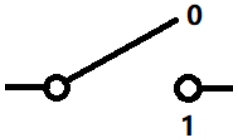

Figure 2.1: A switch can represent 0 and 1 state.

In digital computers, a bit is a electronic switch which can represent 0 and 1 by its open and closed states (Figure 2.1). Using the characters 0 and 1 to represent any number, for example:

$$\begin{aligned} 11010010 &= (1 \times 2^7) + (1 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) \\ &\quad + (0 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (0 \times 2^0) \\ &= 210 \end{aligned} \tag{2.1}$$

The strings of bits, known as binary strings, can be used to represent far more than just numbers by defining special meanings to different bits. By using the first bit (left most) to represent the sign of a number, can then represent negative numbers, by slicing the binary string into different sections, represent a floating-point number (decimal number), and by following some character encoding standard like American Standard Code for Information Interchange (ASCII) and Unicode, represent text. In a computer, all things such as images, songs, and movies are binary strings.

Similar to digital computers, a quantum computer stores data using qubits. The difference is, a classical bit always has a well-defined state, either 0 or 1, because a switch cannot be both open and closed at the same time. However, a qubit can represent both 0 and 1 simultaneously and only needs to be well-defined when making a measurement to extract a result. By representing a qubit mathematically, using vectors. For qubits that give a certain output of a 0 or a 1, can represent them by two orthogonal 2×1 vectors:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{2.2}$$

These combinations can be used to represent vectors that look like

$$|q_0\rangle = \begin{pmatrix} a_0 \\ a_1 \end{pmatrix}. \tag{2.3}$$

This is called the *state vector*, which is a linear combination of the vector $|0\rangle$ and $|1\rangle$, coined 'superposition' in quantum mechanics:

$$|q_0\rangle = a_0 |0\rangle + a_1 |1\rangle. \tag{2.4}$$

2.1.2 Rules of Measurement

A quantum computer stores data in qubits, and we can get results by making measurements. In quantum computation, the word measurement means to find the probability of measuring a defined state $|x\rangle$ in the superposition of all quantum states $|\psi\rangle$.

Definition 1 (Probability of Measurement)**.** The probability of measuring state $|x\rangle$ in the superposition $|\psi\rangle$ is given by

$$p(|x\rangle) = |\langle x|\psi\rangle|^2 \tag{2.5}$$

where $\langle a| = |a\rangle^\dagger = (a_0^* \; a_1^* \ldots a_{n-1}^*)$ and

$$\langle a|b\rangle = \begin{cases} 1 & a = b \\ 0 & a \neq b \end{cases}. \tag{2.6}$$

For a single qubit:

$$
\begin{aligned}
|q\rangle_0 &= \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle \\
\langle 0|p_0\rangle &= \frac{1}{\sqrt{2}}\langle 0|0\rangle + \frac{1}{\sqrt{2}}\langle 0|1\rangle \\
&= \frac{1}{\sqrt{2}}\cdot 1 + \frac{1}{\sqrt{2}}\cdot 0 \\
&= \frac{1}{\sqrt{2}} \\
|\langle x|\psi\rangle|^2 &= \frac{1}{2}.
\end{aligned}
\tag{2.7}
$$

Because the measurement is extracting the probability of all possible quantum states, the result must be normalized, which means summing to 1.

$$
\langle\psi|\psi\rangle = |a_0|^2 + |a_1|^2 = 1. \tag{2.8}
$$

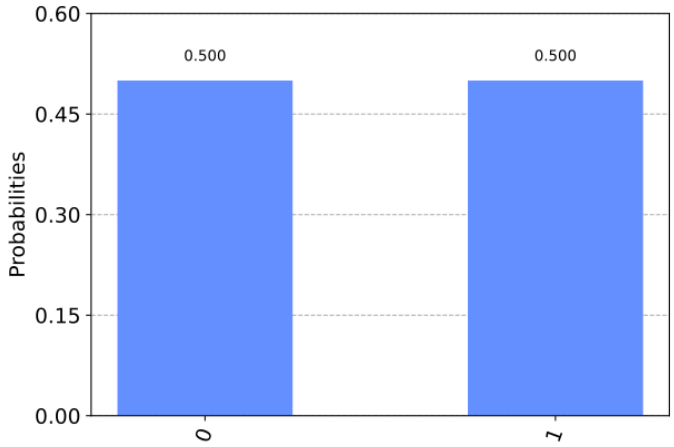

Figure 2.2: The measurement of a single qubit.
The total probability sums to 1.

2.1.3 The Observer Effect in a Real Quantum Computer

As we know that a qubit must be either 0 or 1 after measurement, if we measure a qubit and find it in the state $|1\rangle$, then we have a 100% chance of finding it in the state $|1\rangle$ when we measure again. This means measuring a qubit changes its state.

$$
|q\rangle = a_0|0\rangle + a_1|1\rangle \xrightarrow{measure} |q\rangle = |1\rangle. \tag{2.9}
$$

This is known as collapsing the state of the qubit. So, if we keep track of our qubits at each step of a computation, they would always be in well-defined states, either $|0\rangle$ or $|1\rangle$, and would be no different from classical bits in digital

computer. In quantum computation, to take advantage of quantum superposition, measurements are only used when we need to extract a result.

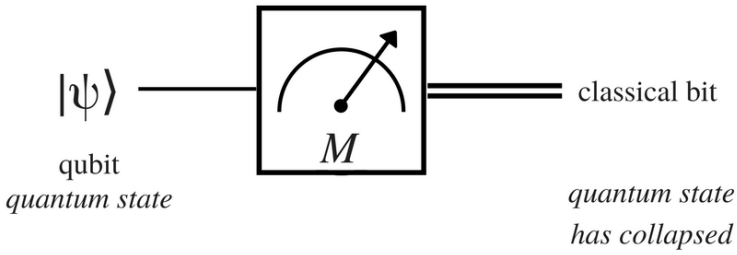

Figure 2.3: The measurement cause a qubit collapse to a classical bit [4].

2.2 Quantum Entanglement

2.2.1 Representing an n-qubit system

Consider an n-qubit system, which has $N = 2^n$ basis states. These basis states are denoted by $|m\rangle_n$ which is equivalent to the state $|M\rangle$ where M is the n-digit binary representation of the integer m. The state $|m\rangle_n = |M\rangle$ refers to the mth basis state of the n-qubit system, which is a tensor product of the single-qubit basis states $|0\rangle, |1\rangle, \ldots, |N-1\rangle$.

Example 2. For a 3-qubit system ($n = 3$),

$$|6\rangle_3 \equiv |110\rangle \equiv |1\rangle \otimes |1\rangle \otimes |0\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 0 \\ 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{2.10}$$

Note here that $N = 2^3 = 8$, $m = 6 \in \{0, \ldots 8\}$, and $M = 8$ (base 2) $= 110$.

This report uses a mixture of Dirac (bra-ket) notation and matrix/vector notation. The connection between the two is the tensor product.

Definition 3. The tensor product acts on 2×1 qubits in the following way. Consider a 2-qubit system ($n = 2$), $a, b, c, d \in \mathbb{C}$.

$$\begin{pmatrix} a \\ b \end{pmatrix} \otimes \begin{pmatrix} c \\ d \end{pmatrix} = \begin{pmatrix} a \cdot \begin{pmatrix} c \\ d \end{pmatrix} \\ b \cdot \begin{pmatrix} c \\ d \end{pmatrix} \end{pmatrix} = \begin{pmatrix} ac \\ ad \\ bc \\ bd \end{pmatrix}. \tag{2.11}$$

The same method applies to larger arrays. The components of a superposition of basis states in this vector notation represent the coefficients of each basis state of the superposition in Dirac notation. For example, for a 2-qubit system, the state $|\psi\rangle = 2|0\rangle - |1\rangle + 3|3\rangle$ is written in vector notation as

$$|\psi\rangle = \begin{pmatrix} 2 \\ -1 \\ 0 \\ 3 \end{pmatrix}. \tag{2.12}$$

Here, the columns are labelled by the basis states sequentially.

2.2.2 Measuring an n-qubit system

Similarly to measuring a single qubit, measuring an n-qubit system means extracting the probability of finding each basis state. For example, in the Figure 2.4, the red block was found at block 2, so the result of the measurement is $|2\rangle$.

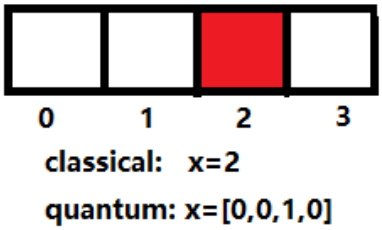

Figure 2.4: The red block at position 2 has a state vector $|2\rangle$.

Because we are still measuring the probability, the rule of normalization holds. For an n-qubit system, the probability still has to sum to unity.

$$\langle\psi|\psi\rangle = \sum_{i=0}^{N-1} |a_i|^2 = 1. \tag{2.13}$$

2.3 Quantum Gates

A classical computer uses a combination of logic gates to perform complex calculations. A logic gate operates on classical bits. For example, the 'NOT gate' flips a bit (0 to 1 and 1 to 0) and the 'AND gate' only gives an output of 1 when both input bits are 1. Similarly, a quantum computer works with quantum gates, and a quantum gate operates on qubits to change their state. We know that a qubit and n-qubit systems can be represented by state vectors, so a quantum gate will operate on a vector and give a new vector. It is not hard to guess that they can be represented as matrices.

2.3.1 Single-Qubit Gates

A simple example of single-qubit gate is the $\hat{X}$-gate: it switches the amplitudes of $|0\rangle$ and $|1\rangle$ for a qubit.

$$\hat{X} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{2.14}$$

$$\hat{X}\,|0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle \tag{2.15}$$

Another important quantum gate is called the Hadamard gate ($\hat{H}_d$-gate), which allows us to create a superposition formed of well-defined states ($|0\rangle$ and $|1\rangle$).

$$\hat{H}_d = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{2.16}$$

$$\hat{H}_d\,|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \tag{2.17}$$

2.3.2 Single-Qubit Gates on Multi-Qubit System

The single-qubit gates are represented by 2×2 matrices so they can acts on $2d$ vectors. The simultaneous operation is represented by the Kronecker product (tensor product) of quantum gates. The action of three Hadamard gates on a 3-qubit system is shown diagrammatically in Figure 2.5 and mathematically in (2.18).

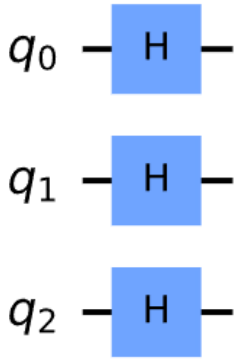

Figure 2.5: $\hat{H}_d$-gates in 3-qubit system [5].

$$\hat{H}_d\,|0\rangle \otimes \hat{H}_d\,|0\rangle \otimes \hat{H}_d\,|0\rangle = (\hat{H}_d \otimes \hat{H}_d \otimes \hat{H}_d)\,|000\rangle = \frac{1}{\sqrt{2^3}} \tag{2.18}$$

2.4 About our Python Project

As we know, the mechanism of quantum computers is quantum gates acting on qubits, which can be represented mathematically by matrices acting on state vectors. Therefore, simulating a quantum computer is not too difficult. A quantum register class was created to initialize and store a state vector. The class also holds a list of matrices that fits the number of qubits used to perform the simulation.

When performing a simulation, the program will iterate through the list of matrices and combine them into a final matrix. Considering the size of the state vector will grow exponentially with the number of qubits, the matrix calculation is lazy, which means the program will do all of the calculations only when performing a measurement. This can be helpful when debugging the program.

The measuring method in the quantum register class is used to extract the probability of the state vector. It activates the matrix calculation and returns a histogram representing the probability of each quantum state. The class also holds methods that can generate matrices using different quantum algorithms fitting the number of qubits.

The matrix calculation was managed by another python script, which was designed to automatically solve the mathematics acting on different kinds of matrices (dense, sparse, and lazy). Numpy library was used to hold the matrix data and perform linear algebra calculations. In quantum computing, most matrices are found to be sparse (a lot of zeros) so a special algorithm for sparse matrices can help to increase the performance of the program.

2.4.1 Sparse Matrices

Sparse Matrices are implemented using a Compressed Sparse Rows (CSR) format. This is a popular format along with Compressed Sparse Columns (CSC) because almost every algebraic operation applied on those data structures runs faster than other implementations. Nevertheless, there exist some drawbacks with this implementation such as modifying the entries of the matrix, reshaping the matrix, or some other I/O operations related. In those cases, we should consider other implementations such as Dictionary of Keys (DOK) when creating or modifying the matrix, and then switch to CSR or CSC to operate with them [6, 7].

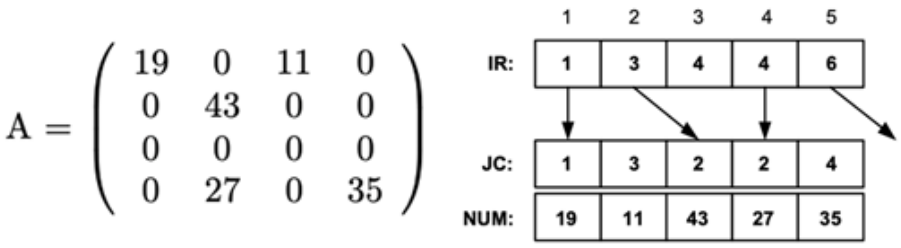

Figure 2.6: CSR representation of matrix A [8].

This data structure consists of three arrays, or linked lists, see Figure 2.6. The IR array stands for the compressed rows index i.e. if we want to access the elements on the second row, we take the pointer stored on position two of the array and the next one, those will be the start and end pointers of the row. Now we can access the other two arrays with the previous pointers on the JC and NUM (JC [start: end] and NUM [start: end] if we use Python notation, note that this returns elements from start to end-1) arrays which store the column and element in position $A[2][jc]$. By using this representation, we can skip multiple rows and therefore calculations when the start pointer is the same as the end pointer.

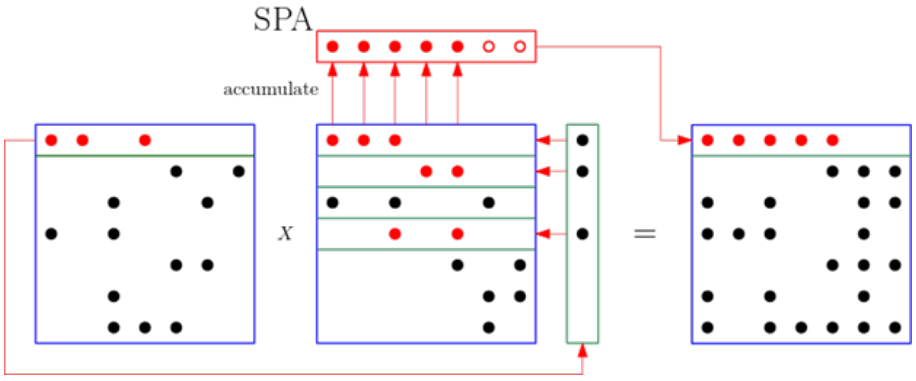

Figure 2.7: Row-wise multiplication using Sparse Accumulator (SPA).

The algorithm to initialise the data structure from a dense matrix—or from another types of formats such as DOK—should consist of an iteration though all the elements or entries of the matrix or the dictionary by rows and store the non-zero elements with its column index by just pushing the results at the JC and NUM vectors. While doing this, we must count how many elements we added to those arrays for this row and when the row is done, push to IR the cumulative sum with the elements we counted.

We can clearly tell the memory space that we are saving now. The JC and NUM vectors are size nnz (non-zero elements) and IR is size $m+1$ (m being the number of rows of the dense matrix representation).

Now that we have introduced our implementation for sparse matrices, we can talk about the algorithms designed for optimised algebraic operations. This section will consist of the explanation of operations with vectors, operations with dense matrices, and operations with other sparse matrices. Will also talk about some auxiliar and the initialisation algorithms.

Figure 2.8: Kronecker product of matrices [9].

The algorithm for multiplication of sparse matrices and vectors consists of the traditional one, traversing the matrix by rows or columns, depending on the order of multiplication and output a vector. In case of needing to traverse the sparse matrix by columns, the algorithm transposes the matrix, obtaining a CSC. For multiplication of sparse matrix and dense matrix we use the same concept as before, we can see the multiplication process as a multiple vector multiplication. Nevertheless, for multiplications between two sparse matrices, we implemented a row-wise multiplication algorithm. This implementation suits perfectly because

we only need to iterate through the rows of the matrices so there is no need to transpose any of them. The algorithm uses an auxiliar data structure called SPA. In our case, coding with Python, this data structure can be as simple as a Python list [8].

We can observe on the Figure 2.7 how we use the SPA to accumulate the partial results of the multiplications and then store that SPA as a new row of the result. We can exploit the CSR format to access only the i-th rows on the second matrix, i being the columns of the first matrix row that are occupied by an element, in other words the column index of non-zero elements.

	Time Complexity	**Space Complexity**
mulSPMV	$\mathcal{O}(nnz) + \mathcal{O}(\text{transSPM})^*$	$\mathcal{O}(n)$
mulSPMDM	$\mathcal{O}(nnz \cdot n)$	$\mathcal{O}(nnz) + \mathcal{O}(mA)$
mulSPMSPM	$\mathcal{O}(nnzA \cdot nnzB) + \mathcal{O}(n)$	$\mathcal{O}(mA \cdot mB)$
tenSPMM	$\mathcal{O}(m\text{DM} \cdot nnz)$	$\mathcal{O}(mA \cdot mB)$
tenSPMSPM	$\mathcal{O}(nnzA \cdot nnzB)$	$\mathcal{O}(nnz)$
$DM \rightarrow DOK$	$\mathcal{O}(m \cdot n)$	$\mathcal{O}(nnz)$
$CSR \rightarrow DOK$	$\mathcal{O}(nnz)$	$\mathcal{O}(nnz)$
$DOK \rightarrow CSR$	$\mathcal{O}(nnz) + \mathcal{O}(\text{sort dictionary})$	$\mathcal{O}(m) + \mathcal{O}(nnz)$
$CSR \rightarrow DM$	$\mathcal{O}(nnz)$	$\mathcal{O}(m \cdot n)$
transSPM	$\mathcal{O}(nnz) + \mathcal{O}(n)$	$\mathcal{O}(m) + \mathcal{O}(nnz)$
addSPMSPM	$\mathcal{O}(nnzA) + \mathcal{O}(nnzB)$	$\mathcal{O}(m) + \mathcal{O}(nnz)$

Table 2.1: Time and memory complexity for the algorithms mentioned. SPM: sparse matrix, DM: dense matrix, V: vector, mul: multiplication, ten: tensor product, trans: transpose, add: addition, and $\rightarrow$: format changing.
* Only using 'transSPM' depending if we are multiplying the vector on the left or on the right.

We used this idea and the basic algorithm of the Kronecker product with matrices to implement the tensor product. The idea for the algorithm is basically to traverse again the matrices by rows so at the end we construct the final matrix by rows, not by chunks which would be computing the 'same color cells' in Figure 2.8. That is because in doing so, we would have to access multiple times to the same first matrix row. We exploit the CSR format by efficient row accessing and we can easily compute the skipped zero values, therefore the new index for the new value by just a simple offset of the first matrix column and the second one. For this algorithm we can see the huge amount of space we can save, therefore execution time for following operations, by using sparse matrices.

Finally, the addition and the algorithms for changing the format of the sparse matrices are trivial. For the addition we access again in parallel to the rows and use an auxiliary dictionary to hash all the elements to their respective position using a key of (row,col) and convert the DOK format generated to a CSR one.

Now that we have discussed all the algorithms implemented and how they work, we present in Table 2.1 the estimated time and memory complexities for the algorithms. Notice that some algorithms are subject to other internal python implementations that we could not find its time and memory complexity [10].

Chapter 3

Grover's algorithm

3.1 Motivation

Grover's algorithm is a search algorithm, analogous to a classical computer searching for a particular element in a list. For a list consisting of N elements, the classical method entails acting on each element in turn. For the worst case scenario in which the element you are looking for is the last one you check, would take N operations. Using quantum computation and Grover's algorithm, this can be reduced to $\mathcal{O}\sqrt{N}$ operations.

Definition 4. Grover's algorithm on an n-qubit system act as the function f acting on the integers $s, x \in (0, \dots N)$ $(N = 2^n)$, where f is given by

$$f(x) = \begin{cases} 1 & x = s \\ 0 & \text{otherwise} \end{cases}. \tag{3.1}$$

This $\mathcal{O}\sqrt{N}$ comes from the Principle of Superposition that governs quantum mechanics. In principle, Grover's algorithm is checking each of the 2^n elements of a database, for an n-qubit system, to see if it matches a certain query, $f(x)$. In a classical computer, one must check each element individually. However, in the quantum domain, if you can evaluate f on either element x or y individually, then you can also evaluate f on the superposition $\frac{x+y}{\sqrt{2}}$ to yield the result $\frac{f(x)+f(y)}{\sqrt{2}}$. Hence, the number of operations required reduces to $\mathcal{O}\sqrt{N}$. Further, if there are M entries of the database that satisfy the query, this reduces to $\mathcal{O}\sqrt{\frac{N}{M}}$ [11].

Grover's algorithm can be seen as answering any of the following problems. It can be used to:

(i) Find a state within a quantum register;

(ii) Determine which state an unknown Oracle is singling out;

(iii) Or, it can do both at the same time.

Since the measurement of quantum states is probabilistic, Grover's algorithm achieves this by amplifying the probability of the desired state, say state $|s\rangle$,

through repeated application of the Oracle and the Grover Operator. Therefore, in the final state, after $\mathcal{O}\left(\sqrt{N}\right)$ iterations, the probability of measuring the state $|s\rangle$ is dominant. Therefore, yielding the state $|s\rangle$ upon measurement is effectively guaranteed.

3.2 Application

Grover's algorithm consists of four main steps:

1. Initialisation
2. Application of the Oracle
3. Application of the Grover Operator
4. Iteration

Step 1: Initialisation

The initial state is always prepared in the same way, using a tensor product of N Hadamard Operators on the basis state $|0\rangle_n$ to create an even superposition of states. By construction, this state has equal probability for each of the $N = 2^n$ basis states. Call this state $|\psi\rangle$.

$$|\psi\rangle \equiv \hat{H}_d^{\otimes_n}|0\rangle_n = \frac{1}{\sqrt{2^n}} \sum_{i=0}^{N-1} |i\rangle. \tag{3.2}$$

In vector notation, $|\psi\rangle$ can be represented as the $N \times 1$ vector

$$|\psi\rangle \equiv \frac{1}{\sqrt{2^n}} \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}. \tag{3.3}$$

3.2.1 Step 2: Application of the Oracle

The Oracle acts on the initial state $|\psi\rangle$ such that it is left unchanged except for the coefficient of the state $|s\rangle$ which is made negative. In Dirac notation, the Oracle is given as

$$\hat{O} = \hat{\mathbb{I}} - 2|s\rangle\langle s|, \tag{3.4}$$

where $\hat{\mathbb{I}}$ is the $N \times N$ identity matrix. In matrix notation, the Oracle is given by a matrix that is essentially the identity matrix except for the ss-element (sth row, sth column) which is equal to -1. For example, for a 3-qubit system with $s = 2$, the Oracle is given by

$$\hat{O}|_{n=3} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}. \tag{3.5}$$

In vector notation, the action of the Oracle on the initial state $|\psi\rangle$ results in the $N \times 1$ vector given by

$$\hat{O}|\psi\rangle = \frac{1}{\sqrt{2^n}} \begin{pmatrix} 1 \\ \vdots \\ -1 \\ \vdots \\ 1 \end{pmatrix}, \tag{3.6}$$

where the sth row contains the entry -1.

Presented here is the case in which the state $|s\rangle$ is a pure basis state of the n-qubit system. However, in general, this will work for any general superposition of these basis states. In this case, however, the Oracle would not be so simple to present in matrix notation. Nevertheless, the algorithm would proceed in the same way.

3.2.2 Step 3: Application of the Grover Operator

The Grover Operator acts on the state $\hat{O}|\psi\rangle$ such that the amplitude of the $|s\rangle$ state is increased while the rest remain equal, thus amplifying the state that we are wishing to single out. In Dirac notation, the Grover Operator is given by

$$\hat{G} = 2|\psi\rangle\langle\psi| - \hat{\mathbb{I}}, \tag{3.7}$$

where $|\psi\rangle$ is our initial state given by equation 3.2. In matrix notation, the Grover Operator is therefore given by

$$\hat{G} = \begin{pmatrix} 2^{1-n}-1 & \cdots & 2^{1-n} \\ 2^{1-n} & 2^{1-n}-1 & 2^{1-n} \\ \vdots & & \ddots & 2^{1-n} \\ 2^{1-n} & 2^{1-n} & & 2^{1-n}-1 \end{pmatrix}. \tag{3.8}$$

For example, for a 3-qubit system ($n = 3$),

$$\hat{G}|_{n=3} = \begin{pmatrix} -0.75 & 0.25 & 0.25 \\ 0.25 & -0.75 & 0.25 \\ 0.25 & 0.25 & -0.75 \end{pmatrix}. \tag{3.9}$$

In vector notation, the action of Grover's Operator on the state $\hat{O}|\psi\rangle$ results in the $N \times 1$ vector given by

$$\hat{G}\hat{O}|\psi\rangle = \frac{1}{\sqrt{2^n}} \begin{pmatrix} 2^{1-n} - 1 \\ \vdots \\ 2^{2-n} - 2^{1-n} + 1 \\ \vdots \\ 2^{1-n} - 1 \end{pmatrix} \tag{3.10}$$

where the sth element is amplified $\forall n$ since

$$2^{2-n} - 2^{1-n} + 1 > 2^{1-n} - 1 \Rightarrow 2^{1-n} > 2^{1-n} - 1, \text{ true } \forall n \in \mathbb{N}. \tag{3.11}$$

For example, for a 3-qubit state ($n = 3$),

$$\hat{G}\hat{O}|\psi\rangle = \begin{pmatrix} -0.2652 \\ \vdots \\ 0.4419 \\ \vdots \\ -0.2652 \end{pmatrix}. \tag{3.12}$$

Following this first iteration, the system is in the state given by

$$\hat{G}\hat{O}|\psi\rangle = \frac{1}{\sqrt{2^n}} \left[(2^{2-n} - 2^{1-n} + 1)|s\rangle + \sum_{i \neq s}^{N} (2^{1-n} - 1)|i\rangle \right]. \tag{3.13}$$

The probability of measuring a given state $|i\rangle$, $i \in \{0, \ldots, N-1\}$, is given by the inner product $\langle i|\psi\rangle = c_i^2$, where c_i^2 is the coefficient of the state $|i\rangle$ in the superposition $|\psi\rangle$. Hence, the probability of measuring the state $|s\rangle$ is given by $(2^{2-n} - 2^{1-n} + 1)^2$, as compared to the probability of measuring any other state $|i\rangle$, $i \in \{0, \ldots, N-1\}$, which is given by $(2^{1-n} - 1)^2$.

Proof. Claim: if the following statement holds $(2^{2-n} - 2^{1-n} + 1)^2 > (2^{1-n} - 1)^2$, then state $|s\rangle$ is amplified.

Let $n \in \mathbf{N}$. So,

$$\begin{aligned} (2^{2-n} - 2^{1-n} + 1)^2 &> (2^{1-n} - 1)^2 \\ 2^{4-2n} - 2 \cdot 2^{3-2n} + 2 \cdot 2^{2-n} - 2 \cdot 2^{1-n} + 2^{2-2n} + 1 &> 2^{2-2n} - 2 \cdot 2^{1-n} + 1 \\ 2^{4-2n} - 2 \cdot 2^{3-2n} + 2 \cdot 2^{2-n} &> 0 \\ 2^3 \cdot 2^{-n} &> 0 \\ 2^{-n} &> 0, \text{ true } \forall n \in \mathbf{N}. \end{aligned}$$

Thus, as the statement holds, the state must be amplified. □

3.2.3 Step 4: Iteration

The final step in Grover's algorithm is to repeat steps 2 and 3 i.e. the application of the Oracle and the Grover Operator $\mathcal{O}(\sqrt{N})$ times. Doing further repetitions

beyond this will result in the dominance of the probability of state $|s\rangle$ diminishing again.

3.3 Geometric Interpretation

Both the oracle and the Grover operator act as rotations of the states $|\psi\rangle$ and $\hat{O}|\psi\rangle$ in the plane spanned by $|s\rangle$ (the intended state) and $|s^T\rangle$ (the state perpendicular to $|s\rangle$). The action of repeating these rotations is to gradually move our initial state $|\psi\rangle$ closer to the intended state $|s\rangle$, which is demonstrated in Figure 3.1.

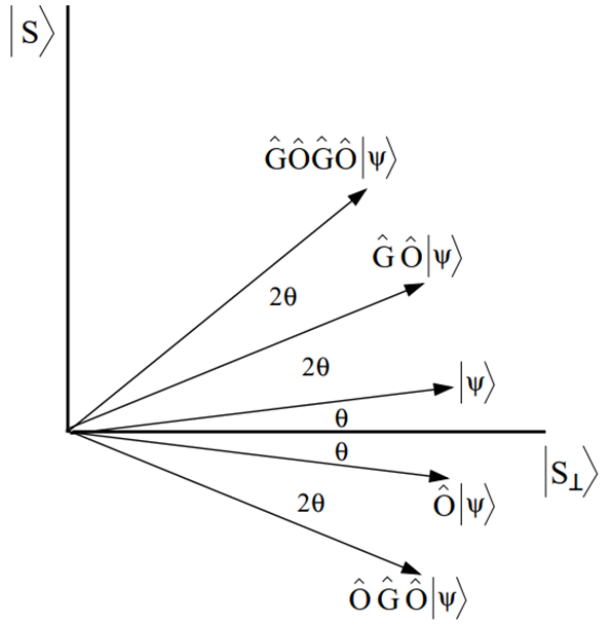

Figure 3.1: The geometric interpretation of the oracle, $\hat{O}$ and the Grover operator $\hat{G}$ acting on a state $|\psi\rangle$ with $|s\rangle$ as the state amplified by the oracle. θ is defined here as the angle between the initial state $|\psi\rangle$ and the target state $|s\rangle$ [12].

To see this algebraically, note that any arbitrary state $|u\rangle$ can be decomposed into the basis containing states $\{|s\rangle, |s^T\rangle\}$, such that $|u\rangle = a|s\rangle + b|s^T\rangle$. The action of the oracle on the state $|u\rangle$ is then,

$$\hat{O}|u\rangle = a|s^T\rangle - b|s\rangle. \tag{3.14}$$

As such, the action of $\hat{O}$ on an arbitrary state is to take the component of it along the s direction and reflect it about the remaining component of $|u\rangle$, orthogonal to $|s\rangle$.

Now, given that an arbitrary vector $|v\rangle$ can be decomposed into the basis containing states $\{|\psi\rangle, |\psi^T\rangle\}$, such that $|v\rangle = c|\psi\rangle + d|\psi^T\rangle$, then we can decompose $\hat{O}|u\rangle$ in this fashion, such that $\hat{O}|u\rangle = c|\psi\rangle + d|\psi^T\rangle$. Then, the action of the Grover operator on state $\hat{O}|u\rangle$ is given by,

$$\hat{G}\hat{O}\,|u\rangle = c\,|\psi\rangle - b\,|\psi^T\rangle\,. \tag{3.15}$$

Therefore, the action of $\hat{G}$ on an arbitrary state is to take the component of it that is orthogonal to $|\psi\rangle$ and reflect it about the $|\psi\rangle$ vector.

Together, this results in the original vector $|\psi\rangle$ being reflected about the $|s^T\rangle$ vector (under $\hat{O}$) and then reflected about the original vector $|\psi\rangle$ (under $\hat{G}$), moving it closer to the intended state $|s\rangle$.

3.4 Limitations

In applying Grover's algorithm to real-world problems, one begins to see its limitations. Many databases of information are coined explicit, meaning that the entries correspond to physical objects (such as people or Web pages). These databases can change and grow at a rapid pace, and so any algorithm searching through them needs to be scalable with the size of the database. However, since Grover's algorithm relies on a superposition of all elements in the database, it is not scalable in this way.

As a result, Grover's algorithm is currently restricted to working with implicit databases, such as in applications of cryptography. However, Grover's algorithm is in competition with classical parallel processing methods, which are already highly successful and efficient. Hence, it is not currently clear whether Grover's algorithm gives a sufficient advantage over classical methods [11].

Further, there is the issue of the Oracle. Grover's algorithm requires an Oracle to determine what the algorithm is searching for, although this often needs to be explicitly constructed for each search problem. Due to the nature of quantum computers, this needs to be implemented in quantum hardware which is not as simple as classical hardware, which are simply combinations of classical logic gates. Hence, the increased speed of Grover's algorithm can be diminished due to the need to both physically construct the Oracle and for the computer to run this gate [11].

3.5 Simulation and Results

This section presents the results of our simulation of Grover's algorithm in Python. After implementing the Grover and Oracle operators, we can now simulate how the algorithm would work on a classical computer.

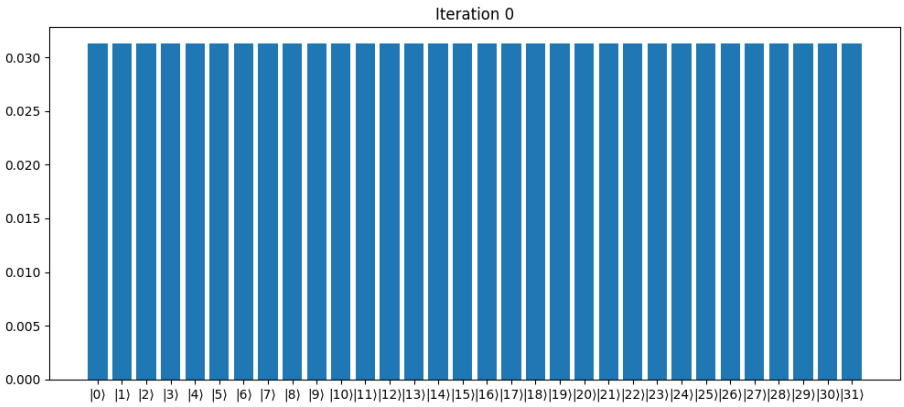

Figure 3.2: Probabilities of observing any of the states before applying any operator.

For this simulation we will use 5 qubits, which gives 2^5 basis states, and will take $|s\rangle = |2\rangle$ as the target state. So, as discussed in Section 3.2, the algorithm should amplify the probability of measuring state $|2\rangle$, the analogue to searching for the number 2 in a list of integers $\{0, 1, \ldots, 2^5 - 1\}$. The first step of the simulation is to create a superposition of all possible states with equal probability. Figure 3.2 shows the probabilities of measuring each state, all of which are identical.

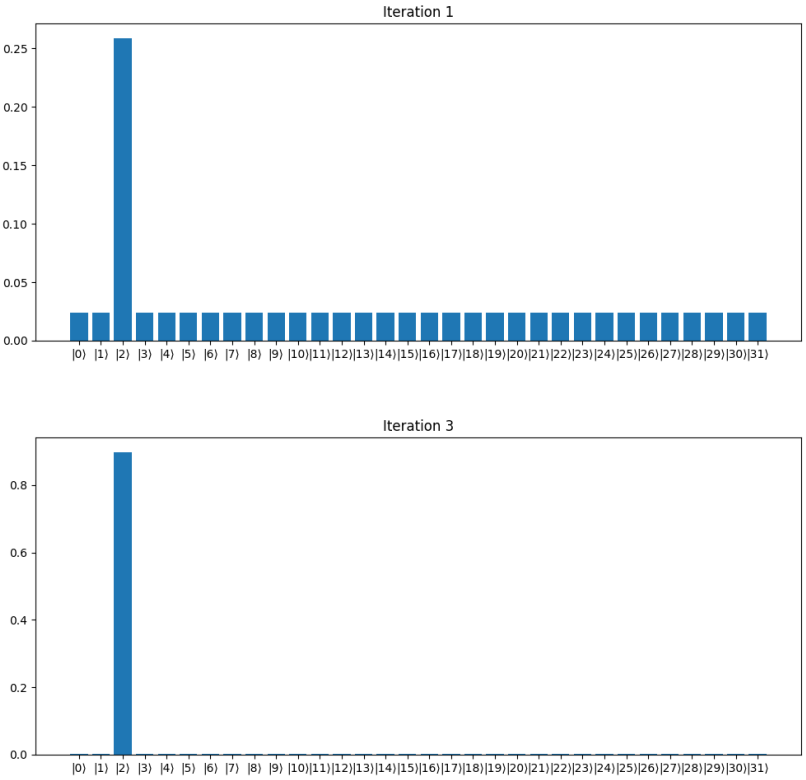

Figure 3.3: Probabilities of observing any of the states, iterations 1 and 3 of the simulation.

For the next step of the simulation, we will apply the Oracle and Grover operators 9 times, making it a 10-iteration process. We will measure the probabilities of observing any of the states at each iteration.

The simulation gives some interesting results; we can see how the probability of our target state increases through the first 3 iterations and reaches the highest value at iteration 4, where we can say that we will observe our target state with probability almost 1 when measuring our system, see Figure 3.3. We can see how fast the probability of observing the target state increases. Even at iteration 3 we will have a high probability of succeeding at observing the target state over all other states if we take a measurement.

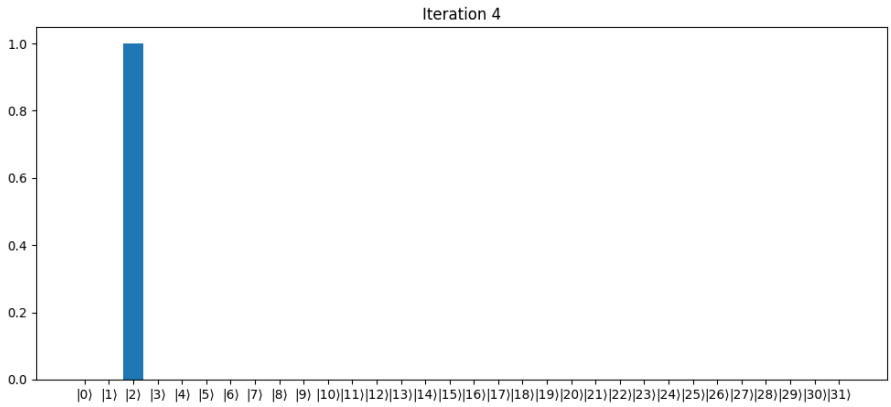

Figure 3.4: Probabilities of observing any of the states, iteration 4 of the simulation, optimal measuring time.

After the 'optimal iteration,' Figure 3.4, the simulation shows how the probability of our target state starts decreasing, which implies that the probability of the other states increases. On the following plots we can observe how iteration 5 is like iteration 3. We can see how that gets worse on the following iterations until iteration 7 where we end up with comparable results obtained in the first iteration of the simulation, see Figure 3.5.

From these results, we see that our system presents cyclic behaviour after a certain number of iterations. This is not unexpected when considering that Grover's algorithm, geometrically, is simply performing rotations on the state $|\psi\rangle$, as discussed in Section 3.3. On the last two iterations of the simulation, observe how the probability of the target state gets worse up to the point of getting a lower probability for the target state than the rest of them on iteration 8. On the last iteration observe how the probability of the target state bounces back to the top but not enough; it is just slightly better than the initial state.

* * *

After simulating Grover's algorithm, the expected result is achieved, maximizing the probability of observing the target state in few operations. The optimal number of operators to be applied to the initial state is reached in iteration 4. Considering the theoretical magnitude was of $\mathcal{O}(\sqrt{N})$, this is a decent result. The simulation results also demonstrate the geometric interpretation of Grover's algorithm due to the rotations leading to a cyclic nature, as previously mentioned. Further iterations would achieve equivalent results repeatedly, so the optimal solution is always the first optimal one, because it is the one obtained with fewer operators applied to the initial state.

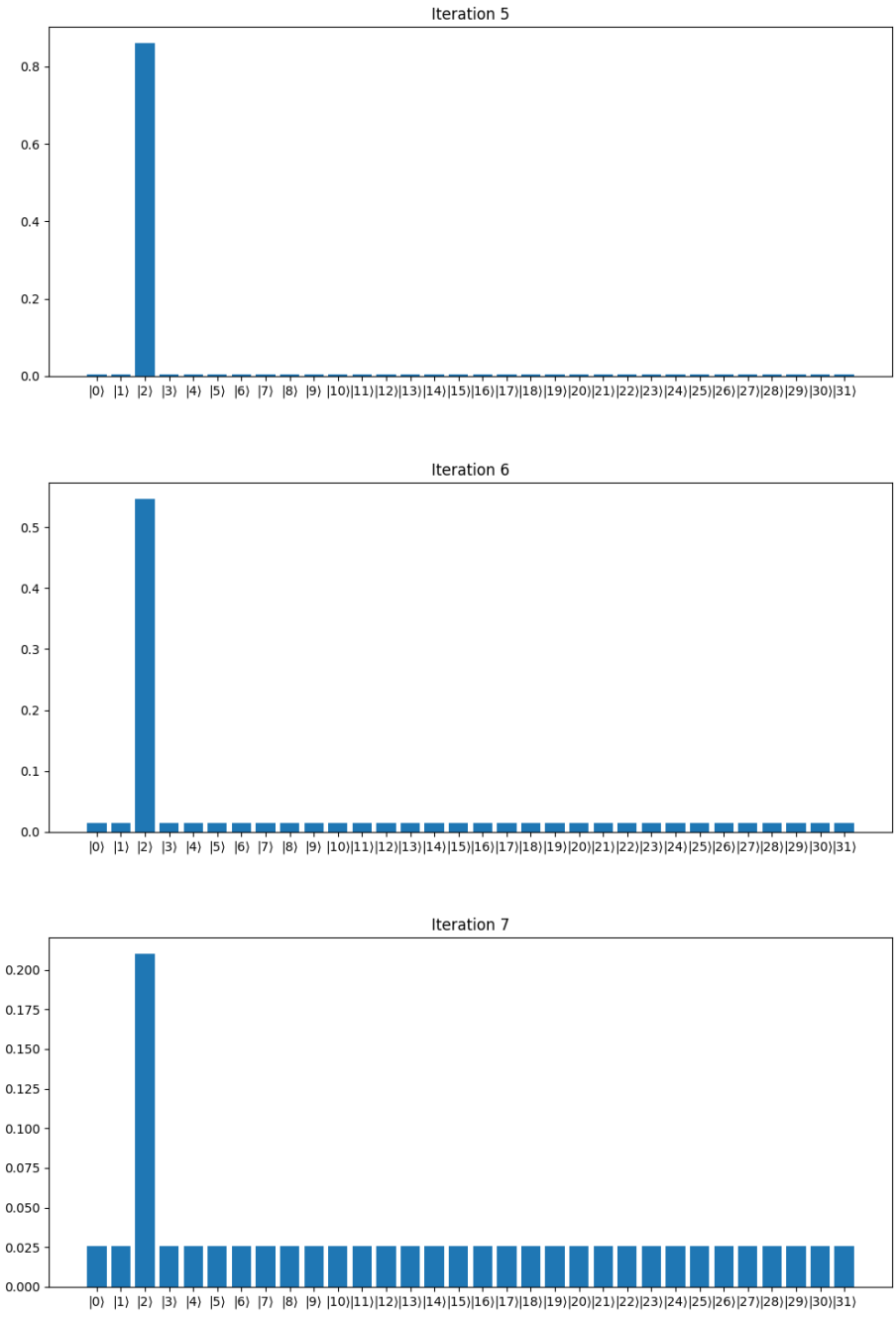

Figure 3.5: Probabilities of observing any of the states, iterations $5-7$ of the simulation.

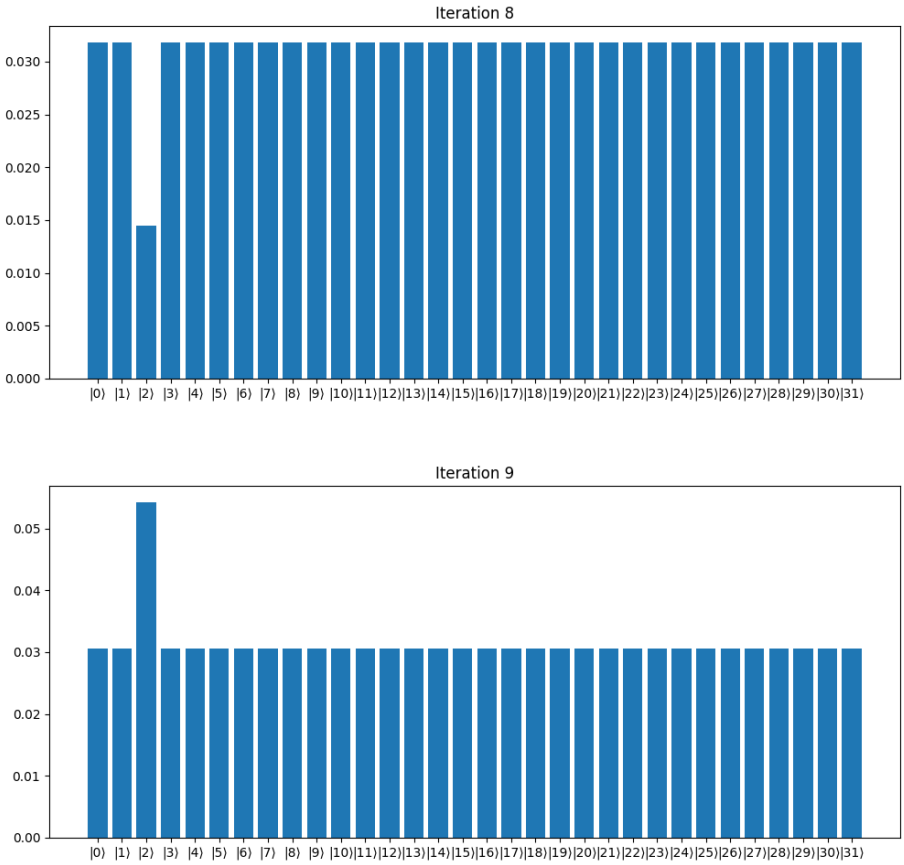

Figure 3.6: Probabilities of observing any of the states, iterations 8 and 9 of the simulation.

Chapter 4

Shor's algorithm

4.1 Motivation

Figure 4.1: A slow classical brute force method[13]

The main form of safeguarding information in the modern era is to use encryption to allow data to be sent or kept securely, passwords are one such example. From figure 4.1, observe the time taken for a classical computer to cracking encryption in a brute force manner. The main assumption is that it is nearly impossible to simply guess factors of large pseudo-prime numbers, but with help from Shor's algorithm that assumption breaks.

One form of this encryption is called Rivest–Shamir–Adleman (RSA) encryption which uses a large pseudo-prime number that can only be decrypted by knowing the factors. On classical computers, this process takes an astronomically long time, on the order of super polynomial. However, on a quantum computer, this is significantly cut to an order of polynomial degree to factor the product of two primes. Shor's is a basic algorithm and one major issue is that the number of required qubits is high compared to more optimised algorithms—real world quantum computers are only able to factor small numbers due to current technology limitations. To put it into perspective International Business Machines Corporation (IBM) in 2001 factored the number fifteen with seven qubits.

4.2 Method

The method begins by guessing a solution. One is very unlikely to guess a factor, but this can be checked using the Euclidean algorithm. Thus, we assume that our guess and the number to be factorised are coprime. To illustrate this, let us look at an example. Suppose the number to factorise is $N = 156287$ and guessing a factor of $g = 127$. By the Euclidean algorithm, it is simple to see that 123 is not a factor, but what if one could turn that guess g into a better guess.

Theorem 5. *For any two pairs of numbers g, N that are coprime $\exists\ p, m$ such that $g^p = mN + 1$ which implies two better guesses at factors of N: $g^{\frac{p}{2}} \pm 1$.*

Proof. [14] To begin this proof we start with the Carmichael function and manipulate it in such a way to achieve two possible solutions to the RHS.

$$\begin{aligned} g^p &= mN + 1, \\ g^p - 1 &= mN, \quad \text{(difference of two squares)} \\ \therefore\ (g^{\frac{p}{2}} + 1)(g^{\frac{p}{2}} - 1) &= mN. \end{aligned} \tag{4.1}$$

Thus, there clearly exists two better guesses at factors of N, $g^{\frac{p}{2}} \pm 1$, and thus concludes the proof. □

4.2.1 Problems With Shor's algorithm

There are three main problems with using $g^{\frac{p}{2}} \pm 1$ as a better guess. Firstly, the algorithm is dealing with integers only, so if p is an odd number $g^{\frac{p}{2}}$ is probably not an integer. Thus, a condition is placed on p, such that it must be even. Secondly one of these new guesses might be a multiple of N, thus the other a factor of m, and so neither of these guesses are helpful to us. Thirdly the act of finding p on a classical computer takes as long, if not longer, than to just brute force the factors of N. This final problem is what a quantum computer is used solve.

4.2.2 Finding p

By using the results of modular arithmetic, one can now implement a quantum procedure. First, getting the initial state in the form of (4.1). Then, applying a quantum Fourier transform to bring the state into Fourier space to then measure a frequency associated with the state. This gives us the value p and allows one to break the encryption. Comparatively, a classical computer would have no feasible way to do these computations in reasonable time, and thus would have to guess every possible value. On a quantum computer, we can do all of these calculations simultaneously as a superposition: this is the real advantage of using such a system.

The first of these manipulations is to take our initial state and raise it to every power up to $N - 1$ i.e. $|0\rangle + |1\rangle + \cdots \to |0, g^0\rangle + |1, g^1\rangle + \ldots$ or more generally $\sum_{n=0}^{N-1} |n\rangle \to \sum_{n=0}^{N-1} |n, g^n\rangle$ Our next step is to use modular arithmetic to find the remainders of our new state. For a given n, we have the equation

$$g^n \equiv r \pmod{N} \tag{4.2}$$

with different values of r for different values of n. However, due to the periodic nature of modular arithmetic, there will exist a period in which g^{n+p} gives the same remainder as (4.2), giving us

$$g^{n+p} \equiv r \pmod{N}. \tag{4.3}$$

Applying this to the state yields multiple groups with a spacing of $n + p$

$$|n_1, r_1\rangle + |n_1 + p, r_1\rangle + |n_2, r_2\rangle + |n_2 + p, r_2\rangle + \dots \tag{4.4}$$

Measuring the full state at this point will collapse the super position into a state where only a single value of r is chosen. A special property of quantum computation is that taking a measurement of a super-position that could have come from multiple elements in that state then the result will itself be a superposition of just those elements. As this builds off modular arithmetic, the result is known to all be some multiple of p apart from one another, regardless of the value of r that was measured. Thus, $|n_i, r_i\rangle + |n_i + p, r_i\rangle + \dots$.

Next, the quantum Fourier transformation takes the state into frequency space to extract the periodicity of the modified state. We start by defining the quantum Fourier transformation.

Definition 6 (The Quantum Fourier Transformation)**.** [15]

A version of the Fourier transformation that applies to super-positions is called the quantum Fourier transformation. It is defined acting on a generic state $|x\rangle$.

$$\hat{QFT}\,|x\rangle = \frac{1}{N}\sum_{n=0}^{N-1}\sum_{y=0}^{N-1} e^{\frac{2\pi i n y}{N}}\,|y\rangle \tag{4.5}$$

Applying it to our modified state $\sum_n \hat{f}\,|n\rangle$, where $\hat{f}$ is the collected manipulations up to this point, we get

$$\frac{1}{N}\sum_{n=0}^{N-1}\sum_{y=0}^{N-1} e^{\frac{2\pi i n y}{N}}\,\hat{f}\,|y\rangle\,. \tag{4.6}$$

To implement this in Python, an $N \times N$ matrix will have to be defined to take the state vectors to frequency space.

$$\hat{QFT} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega^{1\cdot 1} & \omega^{1\cdot 2} & \dots & \omega^{1\cdot(N-1)} \\ 1 & \omega^{2\cdot 1} & \omega^{2\cdot 2} & \dots & \omega^{2\cdot(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{(N-1)\cdot 1} & \omega^{(N-1)\cdot 2} & \dots & \omega^{(N-1)\cdot(N-1)} \end{pmatrix}, \omega = \exp\left(\frac{2\pi i}{N}\right). \tag{4.7}$$

Applying the quantum Fourier transformation yields us the sate $\hat{QFT}\hat{f}\,|y\rangle$, to which multiple measurements are taken. These measurements are multiples of the frequency. By taking many measurment, it is clear there is a common factor between them, and that is the value p.

Once the value p has been found, the next step is to check that it is an even integer, and then to calculate the better guesses as defined in (4.1). Lastly, we use the Euclidean algorithm to check that these improved guesses are indeed factors of N. If they are not, the algorithm starts over with a new value of g.

4.3 Simulation in Python

The following is a simulation of Shor's algorithm using Python with the following parameters:

- The number to decompose into a product of primes $N = 221$
- The initial guess $g = 12$

Step 1: Check that N and g are coprime

Using the Euclidean algorithm, we can show that the Greatest Common Divisor (GCD) of $N = 221$ and $g = 12$ is 1, and hence N and g are coprime. Therefore, we have failed to factorise 221 with our initial guess, and so continue with Shor's algorithm.

Step 2: Initialisation

Next, we create an initial state that is an even superposition of all 221 basis states using a modified Hadamard tensor product, the original version of which was used to initialise for Grover's algorithm. Here, we simply multiply to remove the pre-factor.

$$|x\rangle \equiv \sqrt{221} \cdot \hat{H}_d^{\otimes 221} |0\rangle_{221} = \sum_{n=0}^{220} |n\rangle \,. \tag{4.8}$$

In vector notation, $|x\rangle$ can be represented by the 221×1 vector

$$|x\rangle \equiv \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} . \tag{4.9}$$

Step 3: Find the remainders

Next, we apply the modular exponentiation gate $\hat{f}$ to our state $|x\rangle$, where $\hat{f}$ is defined by its action on the basis states:

$$\hat{f}\,|x\rangle = |g^n \pmod N\rangle = |n, r\rangle\,, n \in \{0, \ldots, N-1\}. \tag{4.10}$$

Hence, applying $\hat{f}$ to $|x\rangle$ yields the state

$$\hat{f}\,|x\rangle = \sum_{n=0}^{2} 20\,|12^n \pmod{221}\rangle\,. \tag{4.11}$$

In vector notation, this yields a superposition of the basis states where the coefficient is now the remainder. This results in the 221×1 vector, with the first 16 entries being

$$[12, 144, 181, 183, 207, 53, 194, 118, 90, 196, 142, 157, 116, 66, 129, 1, \dots], \quad (4.12)$$

which are then repeated periodically.

Step 4: Measure a remainder

Next, we measure one remainder by collapsing the superposition into a smaller superposition due to a property of quantum computation. Say it measures $r = 142$, the state that is left over is a superposition of all of the basis states that gave a remainder of 142 with the coefficient 142. In this case, this is a 221×1 vector with all zeroes except for the 10th, 26th, etc. elements that contain 142. In real computation, these non-zero terms are unknown since you can only know them by measuring them and hence collapsing your system into an eigenstate, losing the information. The only thing really known is that they are all separated by a period p.

Step 5: Quantum Fourier Transform

To extract the period of the new state, we apply the quantum Fourier transform as described in (4.7). This results in a superposition of basis states with coefficients that are periodic across the states. Hence, the probability of measuring each state gives a periodic function with respect to the Fourier state and which we can plot. Figure 4.2 shows this for our example parameters.

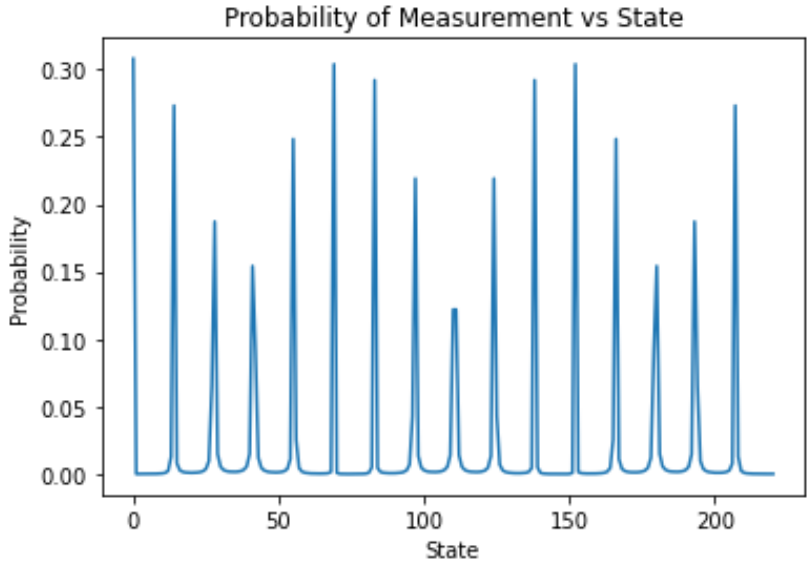

Figure 4.2: The probability of measuring each basis state after the application of the Quantum Fourier Transform.

Step 6: Find p

The fact that this function is periodic implies that the states that yielded that same remainder also follow a periodic pattern, as one would expect. In order to make the better guess, the period p needs to be extracted. However, this is not exactly equal to the period of our probability function. Instead, we make repeated measurements of the basis states to yield eigenvalues that share a common factor of $\frac{1}{p}$. In this case, $p = 16$.

Step 7: Make another guess and factorise N

Finally, with the value of p, we make a better guess. In this case, yielding a value of p that is usable, however we know that this is not always the case. Hence, for some numbers N and guesses g, this is the point at which one would have to try again, starting from Step 1 with a different guess g.

However, since our value of p is sufficient, the improved guesses $g_1 = g^{\frac{p}{2}} + 1$ and $g_2 = g^{\frac{p}{2}} - 1$ are:

$$\begin{aligned} g_1 &= g^{\frac{16}{2}} + 1 = 429981697, \\ g_2 &= g^{\frac{16}{2}} - 1 = 429981695. \end{aligned} \tag{4.13}$$

Then, finding the GCD of N and g_i, $i \in \{1, 2\}$, we have

$$\begin{aligned} \gcd(N, g_1) &= \gcd(221, 429981697) = 17, \\ \gcd(N, g_2) &= \gcd(221, 429981695) = 13. \end{aligned} \tag{4.14}$$

Thus,$\gcd(N, g_1) \cdot \gcd(N, g_2) = 17 \cdot 13 = 221 = N$. Hence, have factorised $N = 221$ into a product of prime factors, meaning that we have broken the encryption.

Chapter 5

Conclusion

In conclusion it is clear from the discussions and simulations that both Grover's algorithm and Shor's algorithm provide essential abilities to a quantum computer.

The ability to search through a list underlies most of modern data handling; the ability to do this in less time than the current, classical case is invaluable to businesses. Grover's algorithm has that potential. However, this comes with some caveats. There are reductions in this speed due to the hardware required to implement the algorithm, and Grover's algorithm is restricted to working with implicit databases.

In a similar sense to Grover's, the ability of Shor's algorithm to factorise numbers into products of primes is a key step in cryptography and breaking current modern encryption. New methods of quantum encryption are being developed due to the flaws in the assumptions they currently employ.

Many predict that quantum computers will take the place of classical computers entirely within the next couple of decades. However, it is more likely that they will work alongside classical computers as technology has reached a theoretical limit with the size of the transistor and avoiding quantum tunneling. The ability to provide security for users through encryption is essential to every form of modern living from banking to streaming to how businesses protect user data. However, Shor's algorithm has many limitations, and is not more efficient as solving the prime factorisation problem than more recent quantum algorithms. Despite this, we have seen the potential of prime factorisation algorithms in our own simulations, and concluding that further development of algorithms could be one of the major steps in progressing from a classical to a quantum world.

Acronyms

ASCII American Standard Code for Information Interchange

CSC Compressed Sparse Columns

CSR Compressed Sparse Rows

DOK Dictionary of Keys

GCD Greatest Common Divisor

IBM International Business Machines Corporation

RSA Rivest–Shamir–Adleman

SPA Sparse Accumulator

Bibliography

[1] N. D. Mermin. *Quantum Computer Science: An Introduction*. Cambridge University Press, 2007. DOI: 10.1017/CBO9780511813870.

[2] P. K. Lala. *Quantum Computing: A Beginner's Introduction*. 1st ed. McGraw-Hill Education, 2019. URL: https://www.accessengineeringlibrary.com/content/book/9781260123111 (visited on 21/03/2022).

[3] J. Chu. *The beginning of the end for encryption schemes?* 3rd Mar. 2016. URL: https://news.mit.edu/2016/quantum-computer-end-encryption-schemes-0303 (visited on 21/03/2022).

[4] N. Mishra. *Understanding basics of measurements in Quantum Computation*. Medium, Mar. 2021. URL: https://towardsdatascience.com/understanding-basics-of-measurements-in-quantum-computation-4c885879eba0 (visited on 22/03/2022).

[5] *Multiple Qubits and Entangled States*. community.qiskit.org. URL: https://qiskit.org/textbook/ch-gates/multiple-qubits-entangled-states.html (visited on 22/03/2022).

[6] *Sparse matrix*. Wikipedia, Mar. 2020. URL: https://en.wikipedia.org/wiki/Sparse_matrix (visited on 21/03/2022).

[7] J. Brownlee. *A Gentle Introduction to Sparse Matrices for Machine Learning*. Machine Learning Mastery, Mar. 2018. URL: https://machinelearningmastery.com/sparse-matrices-for-machine-learning/#:~:text=A%20sparse%20matrix%20is%20a (visited on 21/03/2022).

[8] A. Buluc, J. Gilbert and V. B. Shah. "Implementing Sparse Matrices for Graph Algorithms". In: *Graph Algorithms in the Language of Linear Algebra*. SIAM, 11th Sept. 2013. Chap. 13, pp. 287–313. ISBN: 978-0-89871-990-1. DOI: 10.1137/1.9780898719918.ch13.

[9] U. Thakker et al. "Compressing Language Models using Doped Kronecker Products". In: (Jan. 2020).

[10] *Sparse matrices (scipy.sparse) — SciPy v1.8.0 Manual*. docs.scipy.org. URL: https://docs.scipy.org/doc/scipy/reference/sparse.html (visited on 21/03/2022).

[11] G. Viamontes, I. Markov and J. Hayes. "Is Quantum Search Practical?" In: *Computing in Science & Engineering* 7 (June 2005), pp. 62–70. DOI: 10.1109/MCSE.2005.53.

[12] A. Berera. *Quantum Physics/Principles of Quantum Mechanics S2*. 2021.

[13] *Are Your Passwords in the Green?* Hive Systems. URL: https://www.hivesystems.io/blog/are-your-passwords-in-the-green (visited on 22/03/2022).

[14] *Carmichael function*. Wikipedia, 2019. URL: https://en.wikipedia.org/wiki/Carmichael_function (visited on 21/03/2022).

[15] *Quantum fourier transform*. Wikipedia, 2019. URL: https://en.wikipedia.org/wiki/Quantum_Fourier_transform (visited on 21/03/2022).