A Comparative Analysis for $N = 21$ Shor's Algorithm

Student Muhammad Abdullah Mutahar
Roll No 2023 – 10 – 0113
Instructor Dr. Sabieh Anwar
Course PHY 612 — An Introduction to Quantum Information Science and Technologies

Abstract

We demonstrate the physical implementation of Shor's algorithm after building its theoretical background. The fully functional quantum computer to implement Shor's algorithm is predicted to compute the prime factors in polynomial time. Quantum Phase Estimation forms the crux of this algorithm, breaking down the task of computing prime factors to finding the period of the periodic function $f(x) = a^x \mod N$. Computing prime factors in polynomial time allows us to surpass public key encryption schemes such as the RSA cryptography, which are widely used to protect the data being shared between many two parties. After describing the RSA cryptosystem and how Shor's algorithm can be used it break this encryption mechanism, we compare the results achieved from different implementations of 'a' for $N = 21$.

Introduction

The advent of communication channels and transmission lines in the last century revolutionised the way information was transferred over larger distances. Different encryption systems and mechanisms were developed to secure the data that was being transferred. Before 1976, symmetric cryptosystems were largely used which relied on a single encryption key. This key must had to be exchanged securely over communication channels which was a major drawback.

Whitfield Diffie and Martin E. Hellman later introduced *public key cryptography* in 1976 [1], where they found a way of secure communication by using two keys, called the public key and the private key. This mechanism worked in the following manner. Suppose a person A has to send a message to a person B. B will have a set of predetermined public and private keys, where the public key will be shared with A. A will use it to encrypt the message and send it to B, where B will now use the private key to decrypt it.

This system is currently in place today because even though the public and private keys are connected, no one can easily guess the private key. Furthermore, public key cryptography needs a mathematical procedure that is straightforward to encrypt using the public key, but decrypting without the private key becomes computationally challenging and mathematically intractable on a classical computer. This mathematical term is called the trapdoor function. One example of commonly used public key cryptography which we would encounter next is the RSA mechanism and we would shortly see how the development of Shor's algorithm resulted in a possibility to break this encryption mechanism.

1 RSA Cryptosystem

R. L. Rivest, A. Shamir, and L. Adleman publicly implemented a technique for public key cryptography in 1977 [6], which came to be known as RSA cryptosystem. The technique has the following implementation composing of key generation and information transfer protocol. Any message we want to send first has to be cut down into pieces and converted into integers. The purpose of converting it into integers is to change strings into numbers for encryption.

1.1 Key Generation

RSA technique is fundamentally established on the hardship of factoring large integers into prime numbers. The steps for generating the keys are these:

1. Choose two integers, a and b, such that a and b are prime numbers. If you randomly choose a very large number, you can do a primality test to check whether it is prime or not because the primality test is less costly than going all the way from 1 and checking all the prime numbers and then picking a large prime number from there. Typically, the 2048 bit key is used in RSA, containing 617 decimal digits.

2. Construct a composite number n such that:

$$n = a \times b \tag{1}$$

3. Compute the Euler function $\phi(a, b)$ which is defined in the following way:

$$\phi(a, b) = (a - 1)(b - 1) \tag{2}$$

4. Select a number e which is co-prime with ϕ.

5. Compute the multiplicative inverse of e defined in the following way:

$$d \times e \equiv 1 \mod \phi \tag{3}$$

1.2 Transmission Protocol

Suppose there are two people Alice and Bob. Alice makes a set of two keys with (e, n) as her public key and (d, n) as her private key. Bob wants to send a message M to Alice. He acquires Alice's public key (e, n) and the following procedure is followed for transmission:

1. Bob encrypts his message M using the public key (e, n):

$$C \equiv M^e \mod n \tag{4}$$

2. He transmits the encrypted message C to Alice via a secure communication channel.

3. Alice decrypts the message C via the following manner:

$$C^d \mod n \equiv M \tag{5}$$

The success of the RSA encryption relies on the fact that prime factorization has a high time complexity and if the number is very large, the problem becomes intractable classically. Now we would see how Shor's algorithm can help us acheive this task.

2 Shor's Algorithm

Peter Shor, an American mathematician, developed this algorithm in 1994 which could be used to compute the prime factors of a large number N in polynomial time [7]. Shor's algortihm is essentially composed of a period finding algorithm as we would see. Suppose that we have a composite number N, whose prime factors are to be computed. The algorithm works in the following manner:

1. Choose a number a, which is co-prime with N.

2. Find the period of the function $f(x) = a^x \mod N$. This reduces to finding $f(r) = 1(\mod N)$, where r is the period of $f(x)$.

3. if r is even or $a^{r/2}$ has an integer value, then proceed further. Otherwise, choose a different a.

The crux of this algorithm relies in finding the period r, which if found correctly results in instant factorisation of N. The method is as follows:

$$\begin{aligned} \left[a^{r/2}\right]^2 &= 1(\mod N) \\ \left[a^{r/2}\right]^2 - 1 &= 0(\mod N) \\ \left(a^{r/2}+1\right)\left(a^{r/2}-1\right) &= 0(\mod N) \end{aligned} \tag{6}$$

This means, either $\left(a^{r/2}-1\right)$ or $\left(a^{r/2}+1\right)$ or both share factors with N. Therefore, factors of N can either of:

$$\gcd\left(a^{r/2}\pm 1, N\right) \tag{7}$$

All computations can be performed classically in polynomial time, however, the period finding algorithm has an exponential time complexity and the problem becomes intractable for very large N. Here the usefulness of quantum computers kicks in, in terms of quantum phase estimation carried out by inverse Quantum Fourier Transform (QFT). This allows for the period r to be found in polynomial time.

Let's further discuss how a quantum computer could physically implement this algorithm. The basic idea is to construct a quantum circuit which allows us to compute the values for $f(x) = a^x$, and then find its period. We can develop an oracle V whose action is defined as follows:

$$V|x\rangle|y\rangle = |x\rangle|ay\rangle \tag{8}$$

We develop our circuit such that $|x\rangle$ is contained on the first register (control register) which is composed of m qubits and $|ay\rangle$ is contained on the second register (work register) which composes of n qubits. We restrict n such that $n = \lceil \log_2 N \rceil$, which in turn restricts the maximum value of $|ay\rangle$ and the physical stored value is $|ay \mod N\rangle$. Repeated actions of V, x number of times would yield our desired output on the second register provided $|y\rangle = |1\rangle$. This is encapsulated by our intended oracle U:

$$U|x\rangle|1\rangle = V^x|x\rangle|1\rangle = |x\rangle|a^x \mod N\rangle \tag{9}$$

The periodicity of the function $f(x) = a^x \mod N$ implies that the repeated action of V eventually results in same set of $|a^x \mod N\rangle$ after $x > r$. If we take a sum of this cyclic set, it must be an eigenvalue of U since all the outputs would be present in that set. We define this as:

$$|V\rangle = \frac{1}{\sqrt{r}}\sum_{k=0}^{r-1}\left|a^k \mod N\right\rangle \tag{10}$$

This is an eigenvector of U with eigenvalue 1. We can assign phases proportional to k to each basis vector, and can also further associate these phases up to a factor s depending upon the eigenvector $|u_s\rangle$.

$$|u_s\rangle = \frac{1}{\sqrt{r}}\sum_{k=0}^{r-1} e^{-\frac{2\pi i s k}{r}}\left|a^k \mod N\right\rangle \tag{11}$$

This arbitrary eigenvector yields an eigenvalue of:

$$U|u_s\rangle = e^{\frac{2\pi i s}{r}}|u_s\rangle \tag{12}$$

In all $|u_s\rangle$, the basis state $|1\rangle$ has no phase attached to it as $k = 0$, however this is not true for the rest of the basis states. If we sum up the eigenvectors $|u_s\rangle$ of U, $|1\rangle$ would survive, however, the sum of the phases of all other basis states would add up to zero. This follows from the fact that sum of the roots of unity equal zero.

$$\frac{1}{\sqrt{r}}\sum_{s=0}^{r-1}|u_s\rangle = |1\rangle \tag{13}$$

Hence our initial step of taking $|y\rangle = |1\rangle$ was justified in order to achieve the desired outcome from the oracle U, as $|1\rangle$ additionally turns out to be the sum of eigenvectors of U, allowing us to transmit the phase $\frac{s}{r}$ of $|u_s\rangle$ to the control registers as part of the Quantum Phase Estimation protocol. This phase encoding the period r of the function $f(x) = a^x \mod N$ could then be extracted via IQFT.

2.1 Scheme of working

2.1.1 Initialization

Prepare $|0\rangle^{\otimes m} |0\rangle^{\otimes n}$ with $m = 2n$. Apply $H^{\otimes m}$ on the control register and the NOT gate on the n^{th} qubit on the work register, thus creating a superposition of 2^n states on the control register and $|1\rangle$ on the work register.

$$|0\rangle^{\otimes m} |0\rangle^{\otimes n} \rightarrow \frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} |x\rangle |1\rangle \tag{14}$$

2.1.2 Modular exponentiation function (MEF)

Apply the unitary operation U that implements the modular exponentiation function $a^x \mod N$ on the work register whenever the control register is in state $|x\rangle$:

$$\begin{aligned} \frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} |x\rangle |1\rangle \rightarrow \frac{1}{2^{n/2}} \sum_{x=0}^{2^n-1} |x\rangle |a^x \mod N\rangle \\ = \frac{1}{\sqrt{r2^n}} \sum_{s=0}^{r-1} \sum_{x=0}^{2^n-1} e^{i2\pi sx/r} |x\rangle |u_s\rangle \end{aligned} \tag{15}$$

2.1.3 Inverse Quantum Fourier Transform (QFT)

Apply the inverse quantum Fourier transform on the control register:

$$\frac{1}{\sqrt{r2^n}} \sum_{s=0}^{r-1} \sum_{x=0}^{2^n-1} e^{i2\pi sx/r} |x\rangle |u_s\rangle \rightarrow \frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} |\phi_s\rangle |u_s\rangle \tag{16}$$

2.1.4 Measurements

Measure the qubits of the control register in computational basis. The inverse QFT yields peaks in probability of the states $|\phi_s\rangle$ where $\phi_s \approx 2^n s/r$. There is a high probability of obtaining the location of the these peaks after only a few runs. The number of qubits m determine the accuracy of ϕ_s.

2.1.5 Continued fractions

Compute $\phi = \phi_s/2^n$ and then apply continued fractions to ϕ with maximum denominator value of N in order to extract r from the convergents.

2.2 Quantum Circuit

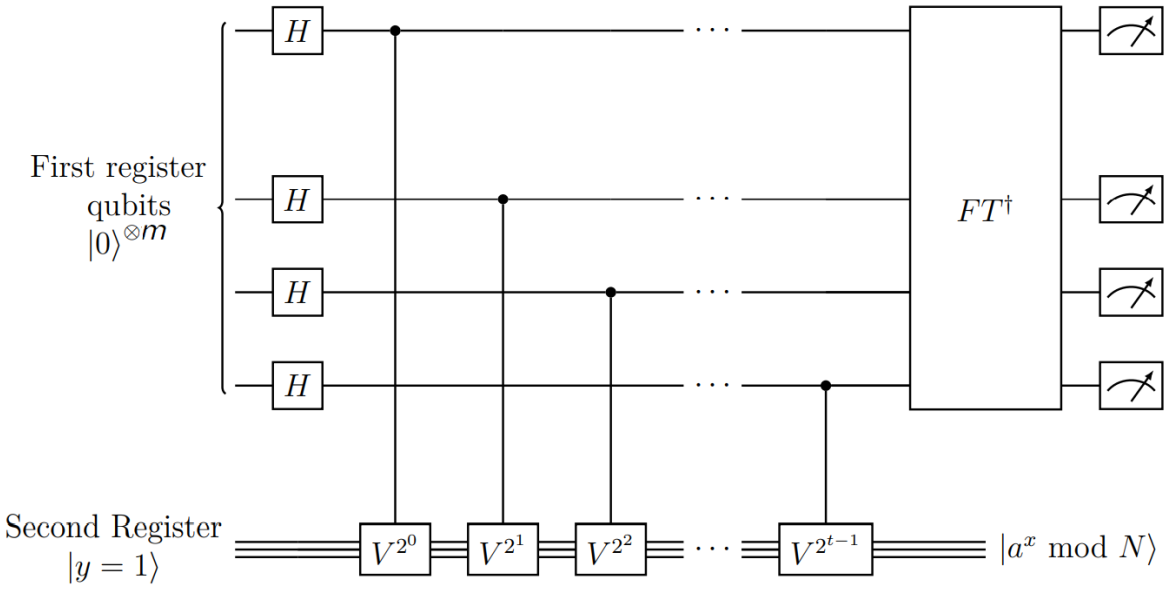


Figure 1: Circuit for period finding of the function $f(x) = a^x \mod N$

2.3 Implementation

We constructed several different physical implementations for $N = 21$ with different values of a. As $N = 21$ is a 5-*bit* number, we selected $n = 5$ number of bits on the work register, and $m = 10$ number of bits on the control register with each bit providing further precision. The rest of the structure of the quantum circuit for Shor's algorithm Fig. 1 is standard except for different implementations of the oracle U each specific to a unique value of a. The construction of the oracle U further splits up into implementations of V^{2^l} with $l = 0, 1, ..., n-1$.

$$\begin{aligned} x &= 2^0 \cdot x_0 + 2^1 \cdot x_1 + ... + 2^{n-1} \cdot x_{n-1} \quad \text{(binary conversion)} \\ U &= (V^{x_0})^{2^0} + (V^{x_1})^{2^1} + ... + (V^{x_{n-1}})^{2^{n-1}} \end{aligned} \tag{17}$$

Each implementation of V is described for different values of a as follows:

2.3.1 $N = 21, a = 4$

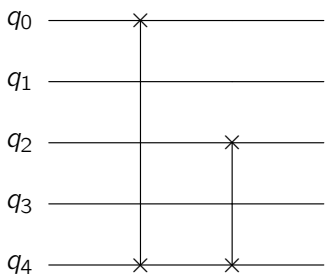
:

$$\begin{aligned} V\,|1\rangle &= |4 \mod 21\rangle = |00100\rangle \\ V^2\,|1\rangle &= |16 \mod 21\rangle = |10000\rangle \\ V^3\,|1\rangle &= |64 \mod 21\rangle = |00001\rangle \\ V^4\,|1\rangle &= |256 \mod 21\rangle = |00100\rangle \end{aligned}$$

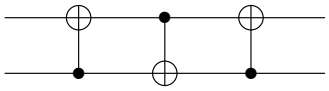
As we can notice a sequence where the qubit having a binary value 1 is switching between the first, third and the fifth qubit. We can use this to construct an implementation of V which swaps first qubit and the third, then third one and the fourth. Any sequence could be spotted and used to construct implementations of V. However, a sequence my not exist, or be difficult to realise and construct. In that case, a general strategy for V^{2^l} with $l = 0, 1, ..., n-1$ can be constructed. For the action of each V^{2^l} in the series, apply a NOT gate to each qubit whose value is different than $|1\rangle = |00001\rangle$.

Sequence Strategy:

Each successive implementation of V follows the following protocol:



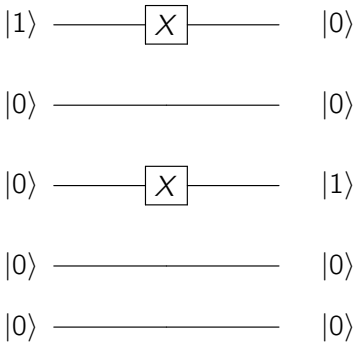
where the swap gate is implemented by:



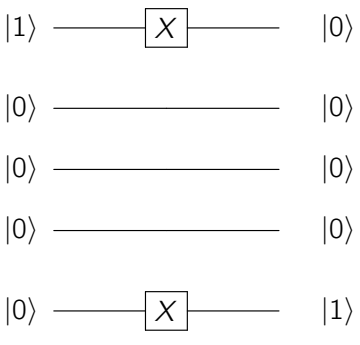
General Strategy:

Construct each implementation of V^{2^l}. First few are shown below:

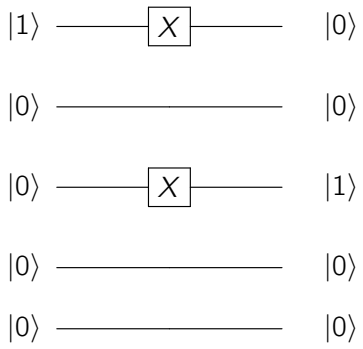
For V^1 such that $V|00001\rangle = |00100\rangle$:



For V^2 such that $V^2|00001\rangle = |10000\rangle$:



For V^4 such that $V^4|00001\rangle = |00100\rangle$:



This method can be carried out for all V^{2^l} with $l = 0, 1, ..., n-1$. We shall detail the sequence strategies we determined from now on-wards.

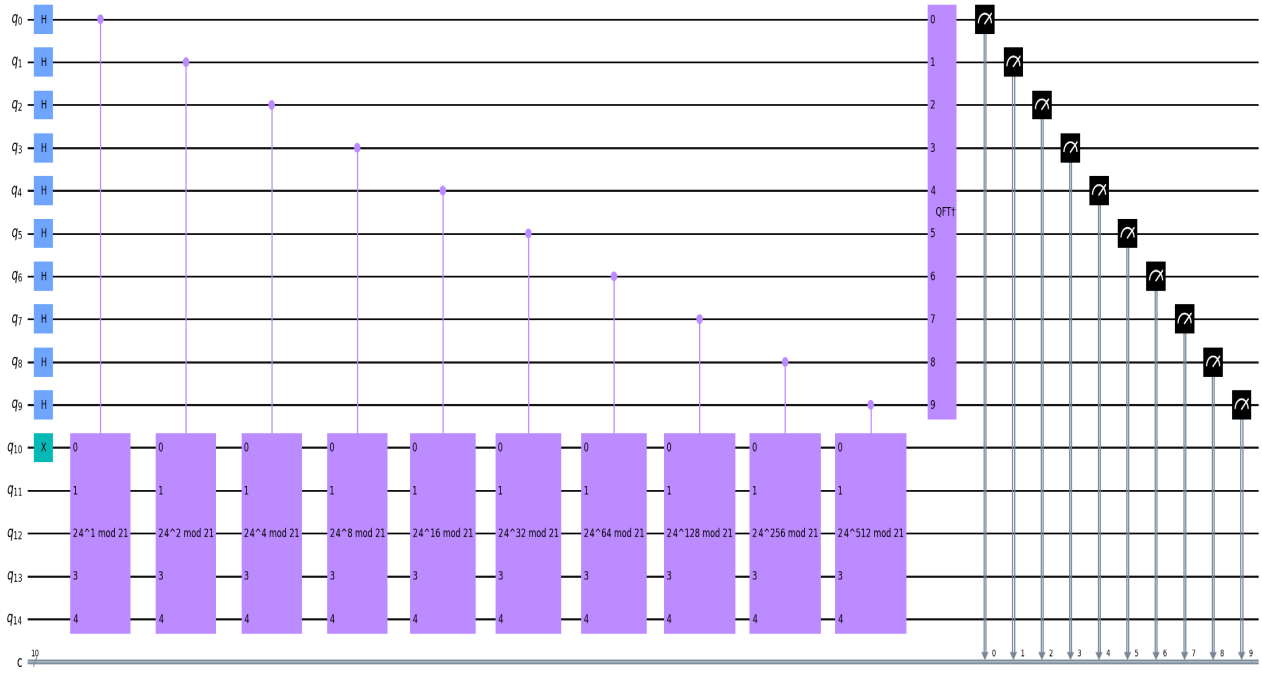
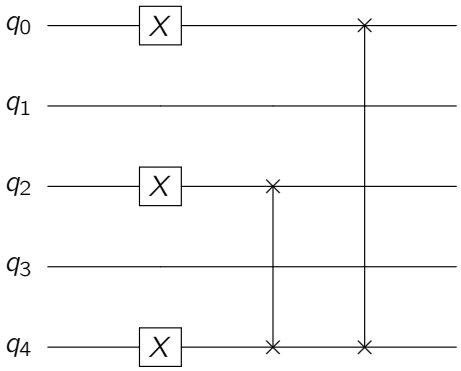


Figure 2: Circuit for implementing Shor's algorithm with $N = 21, a = 4, n = 5, m = 10$

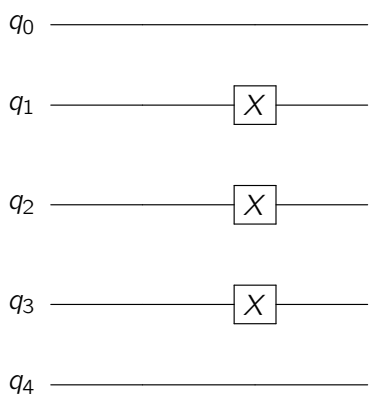
2.3.2 $N = 21, a = 5$

: Each successive implementation of V follows the following protocol:

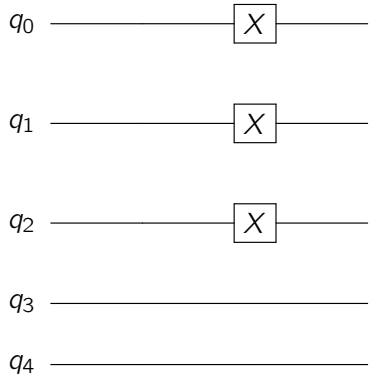


2.3.3 $N = 21, a = 6$

Each successive implementation follows the following protocol even powers of V:

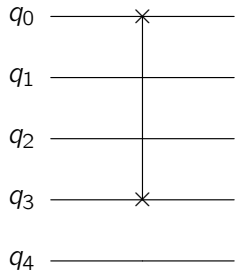


Each successive implementation follows the following protocol odd powers of V:



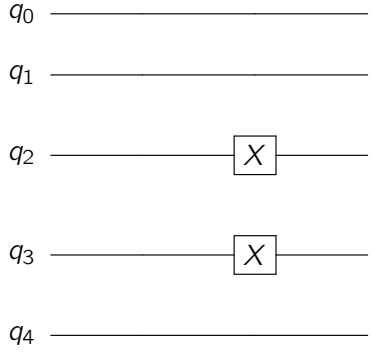
2.3.4 $N = 21, a = 8$

: Each successive implementation of V follows the following protocol:

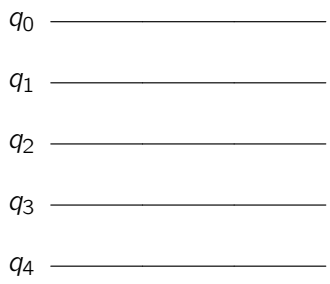


2.3.5 $N = 21, a = 13$

Each successive implementation follows the following protocol odd powers of V:



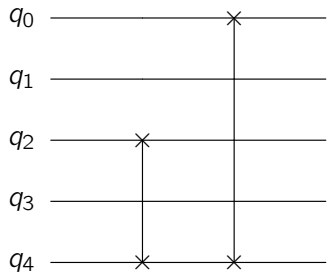
Each successive implementation follows the following protocol even powers of V:



Yes, that is correct. It does nothing for even powers.

2.3.6 $N = 21, a = 16$

Each successive implementation of V follows the following protocol:

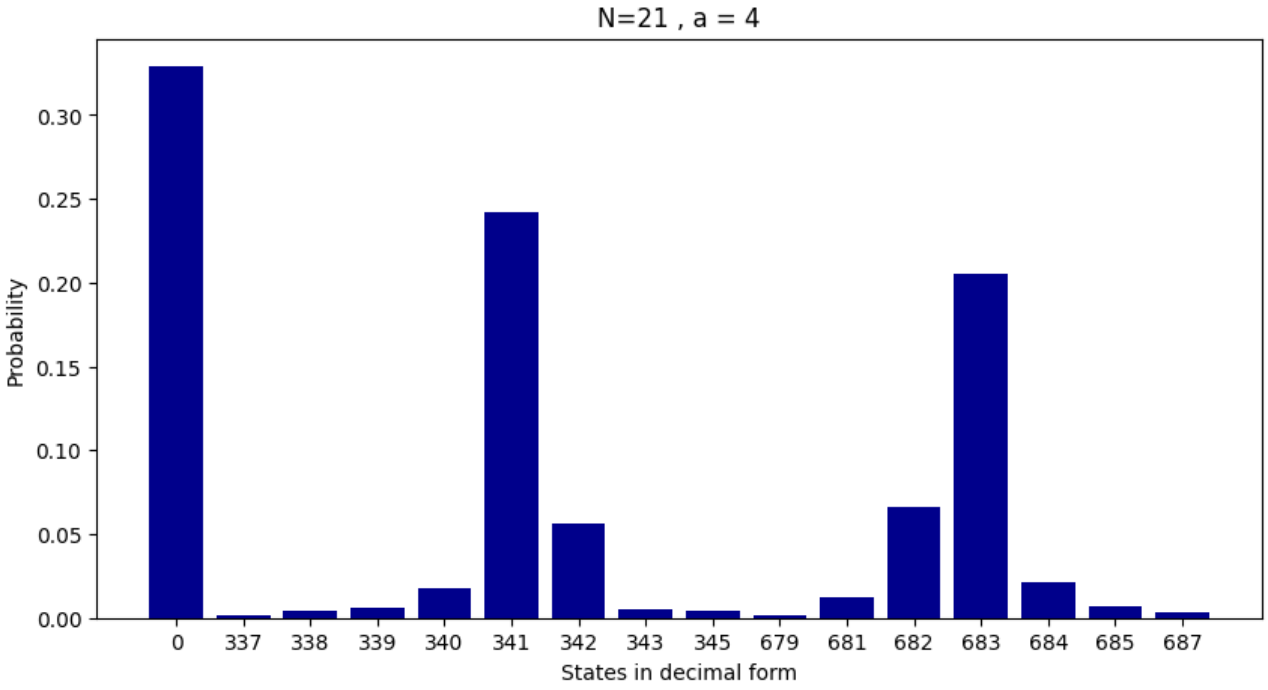


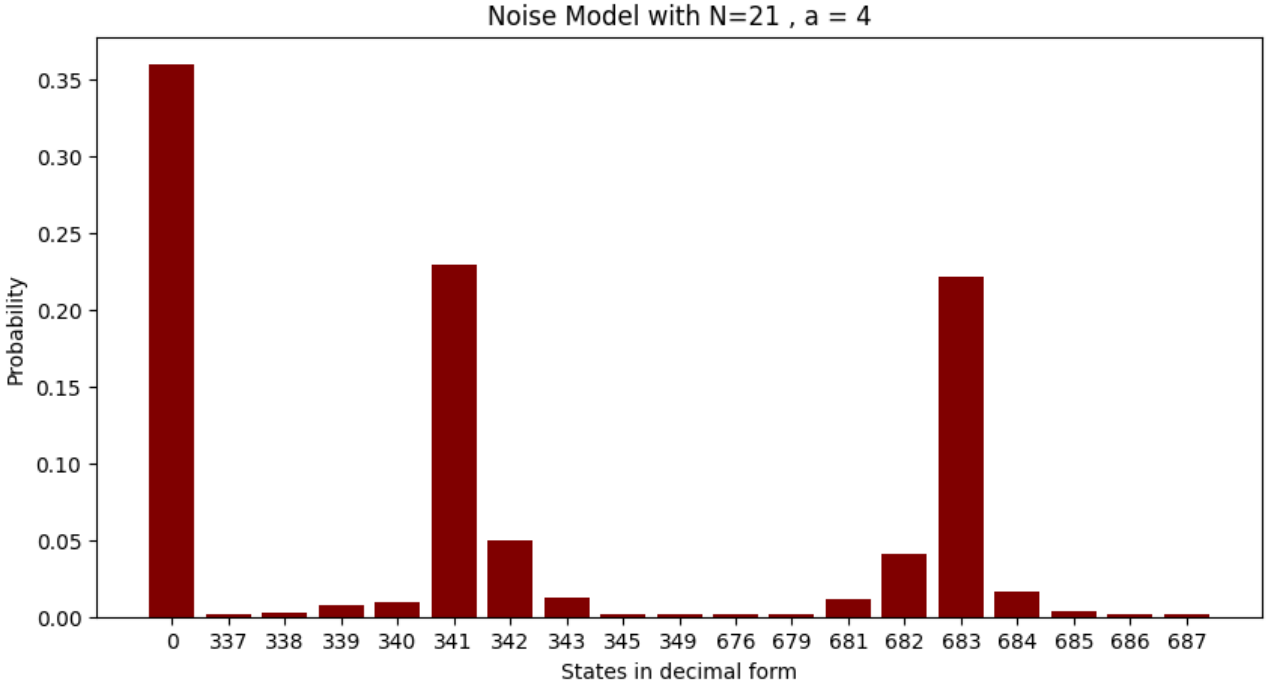
3 Results

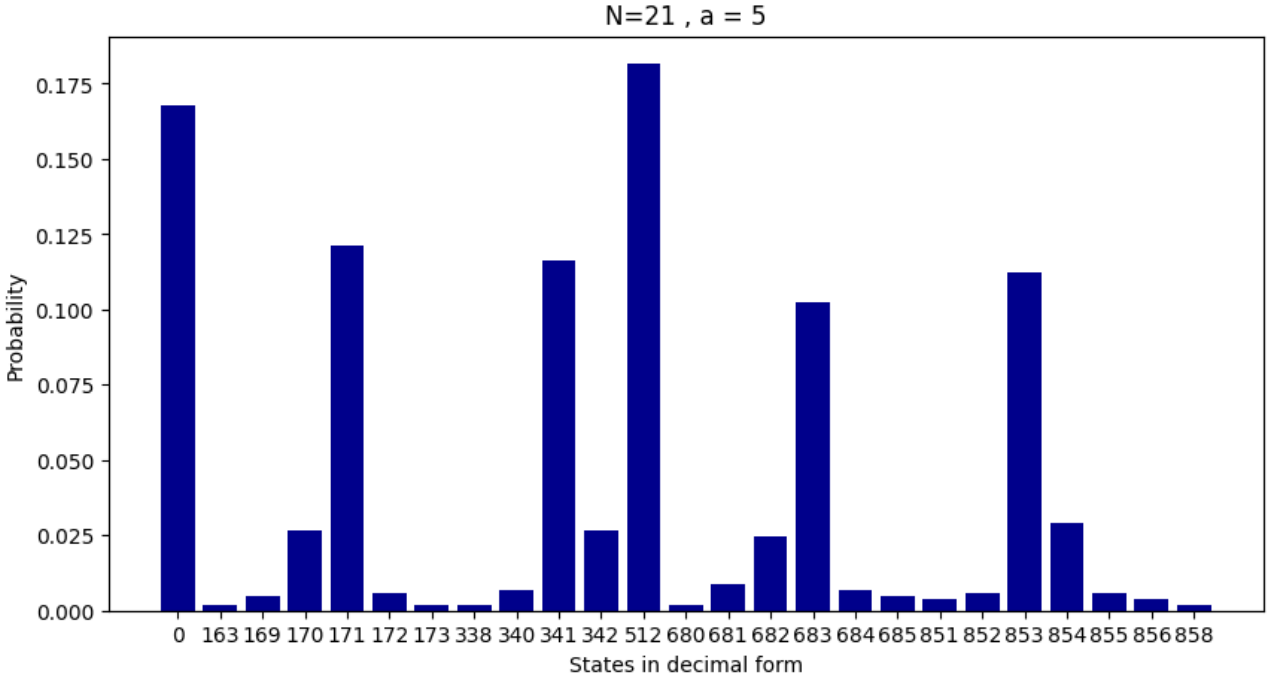
The output data was collected after simulating the quantum circuit Fig. 2 on *Qiskit* for all implementations of 'a' constructed in the previous section. As the number of qubits used in our quantum circuit are $n + m = 15$, it imposes a restriction for this circuit to implemented on a real quantum computer. However, to further simulate the realistic output results we incorporated a *noise bit flip* model. This model is described as follows:

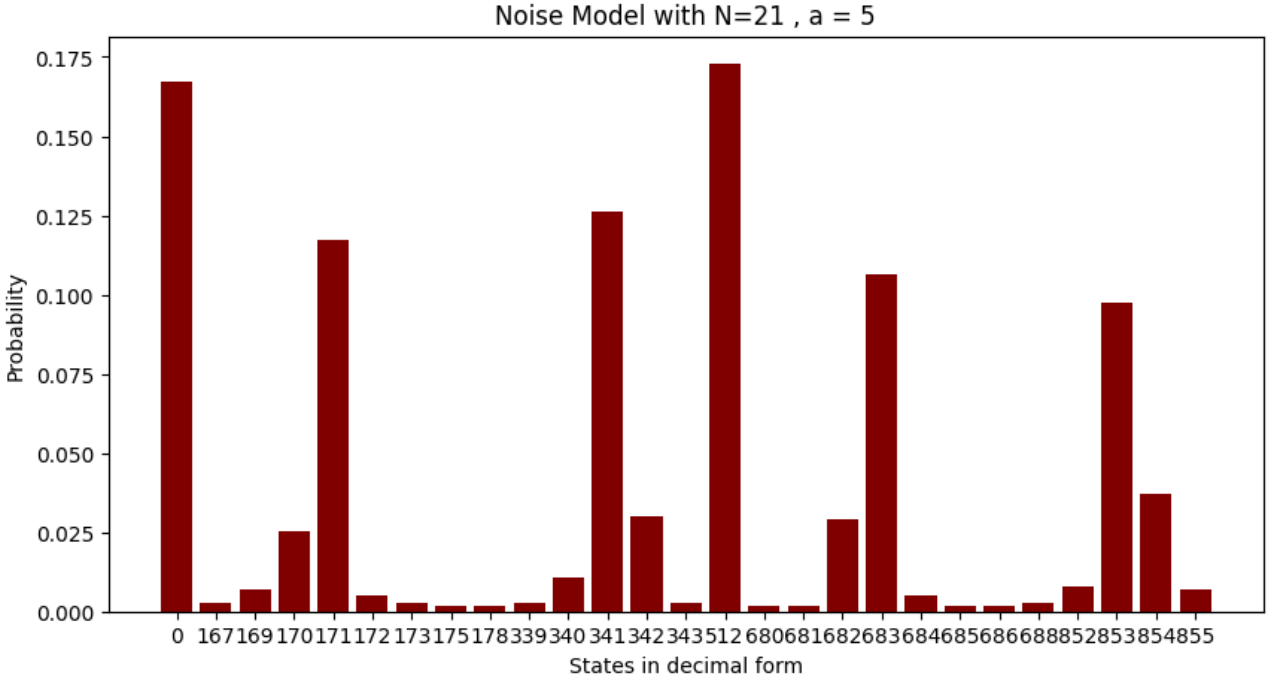
- When applying a single qubit gate, flip the state of the qubit with probability 0.05.
- When applying a 2-qubit gate apply single-qubit errors to each qubit.
- When resetting a qubit reset to 1 instead of 0 with probability 0.03.
- When measuring a qubit, flip the state of the qubit with probability 0.10.

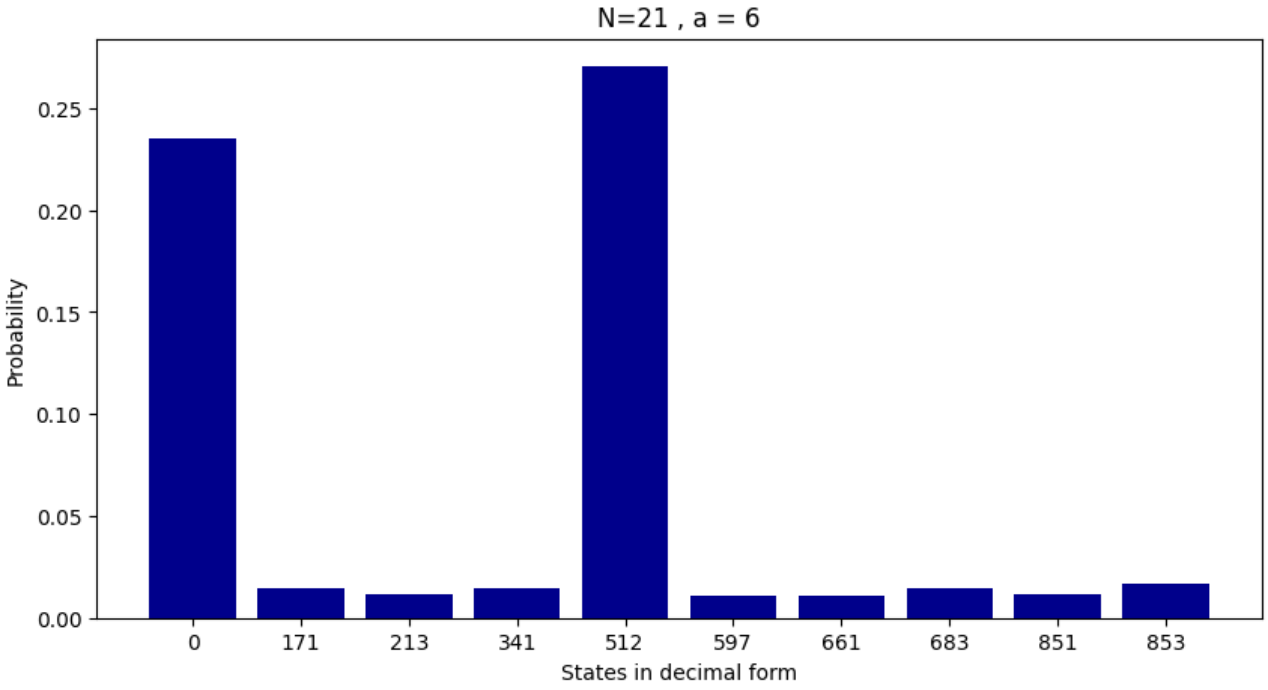
The output data in terms of counts generated verses each state in the control register was converted to probability of measuring each state in decimal form. This data is illustrated for both the ideal case, and with the noise bit flip model.

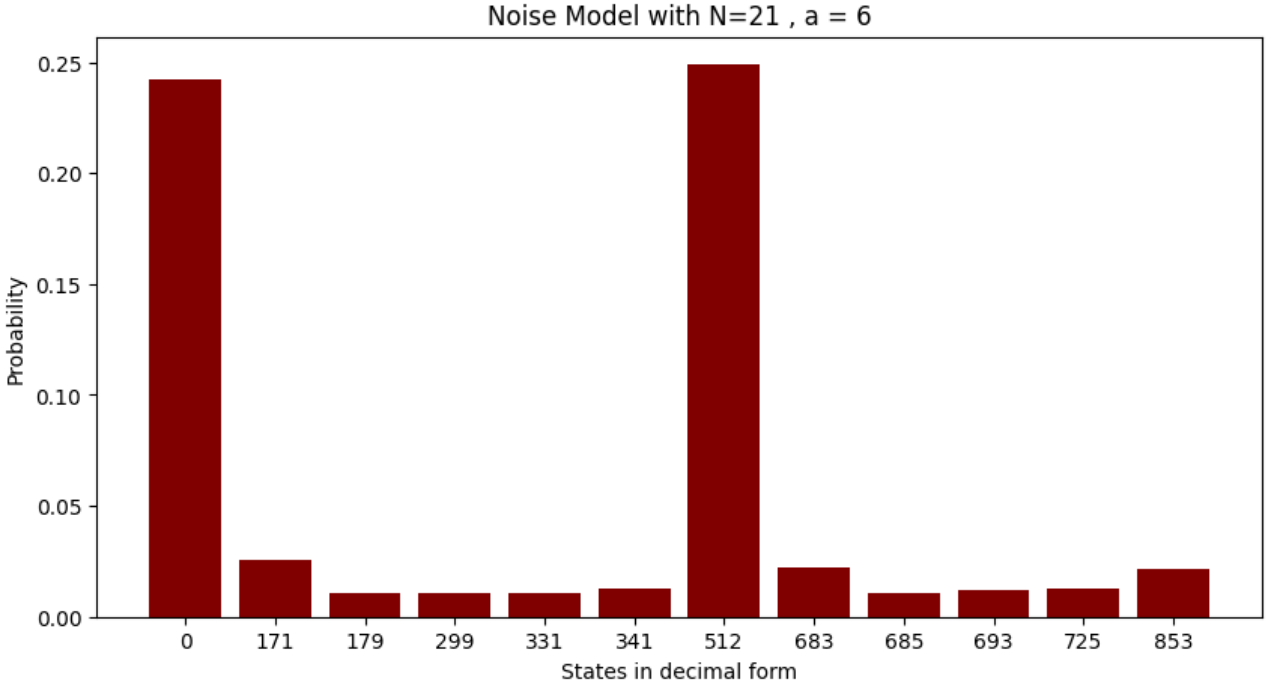

N=21 , a = 4
Probability
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0 337 338 339 340 341 342 343 345 679 681 682 683 684 685 687
States in decimal form

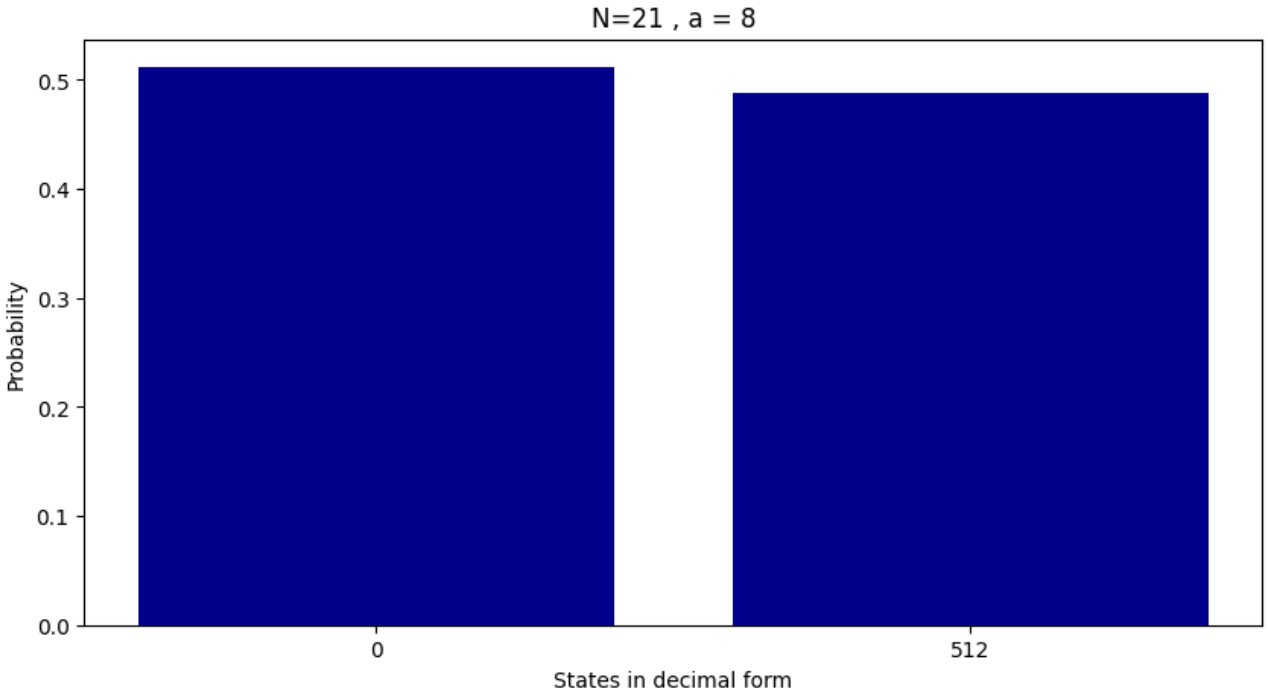

Noise Model with N=21 , a = 4
Probability
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0 337 338 339 340 341 342 343 345 349 676 679 681 682 683 684 685 686 687
States in decimal form

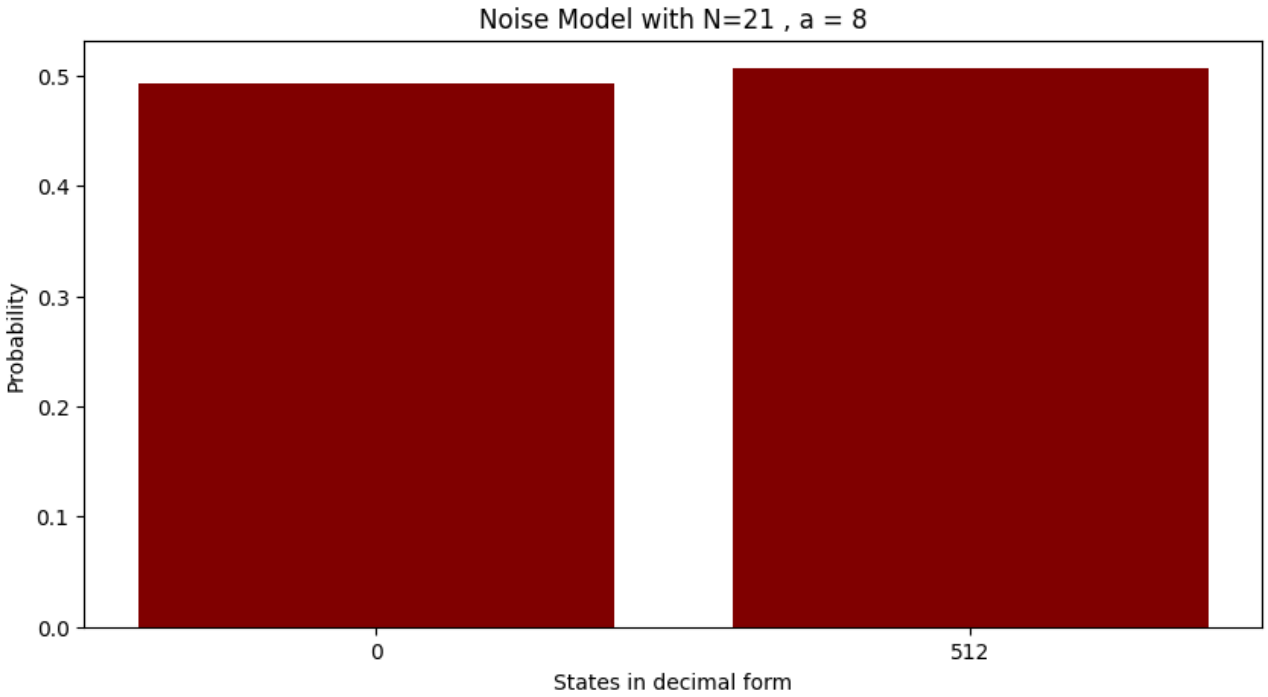

N=21 , a = 5
Probability
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0 163 169 170 171 172 173 338 340 341 342 512 680 681 682 683 684 685 851 852 853 854 855 856 858
States in decimal form

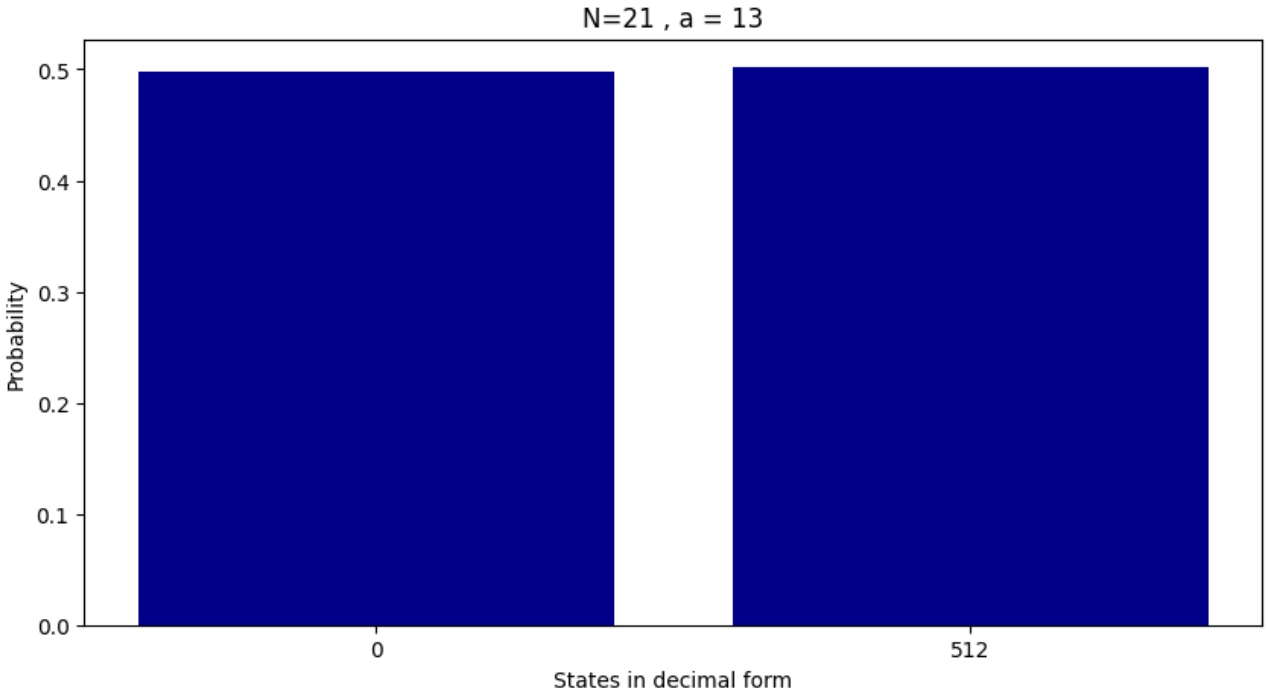

Noise Model with N=21 , a = 5
Probability
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
0 167 169 170 171 172 173 175 178 339 340 341 342 343 512 680 681 682 683 684 685 686 688 852 853 854 855
States in decimal form

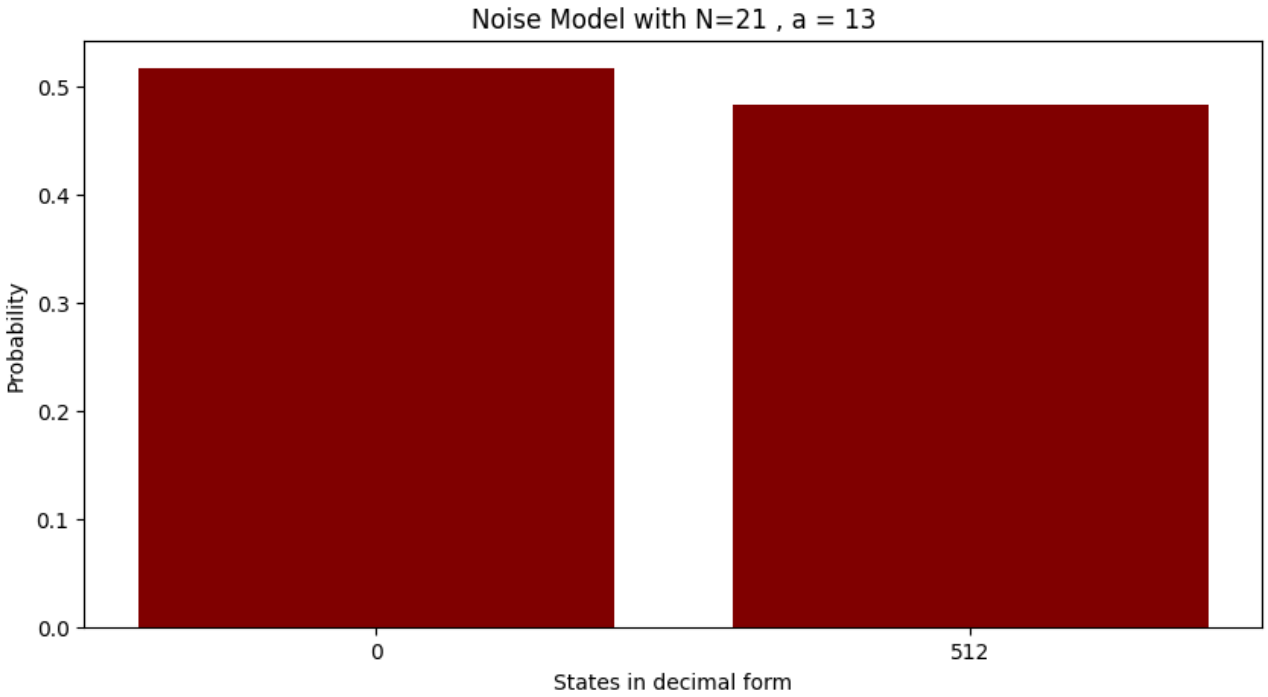

N=21 , a = 6
Probability
0.25
0.20
0.15
0.10
0.05
0.00
0
171
213
341
512
597
661
683
851
853
States in decimal form

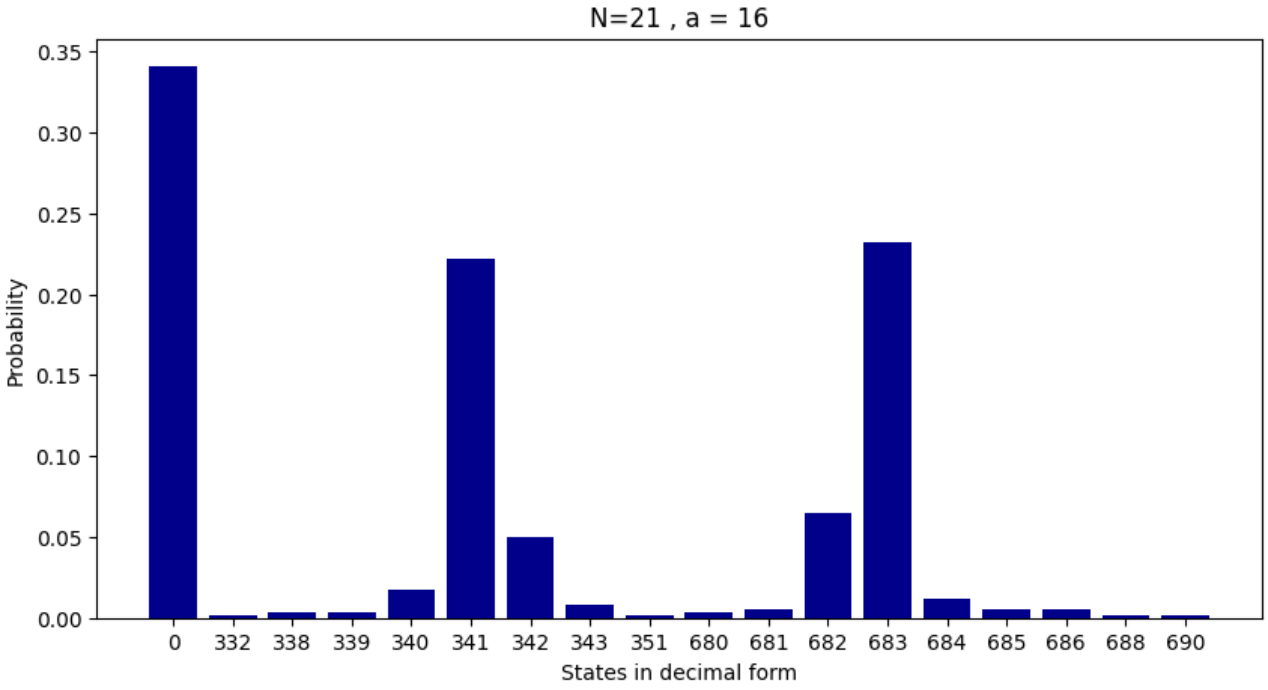

Noise Model with N=21 , a = 6
Probability
0.25
0.20
0.15
0.10
0.05
0.00
0
171
179
299
331
341
512
683
685
693
725
853
States in decimal form

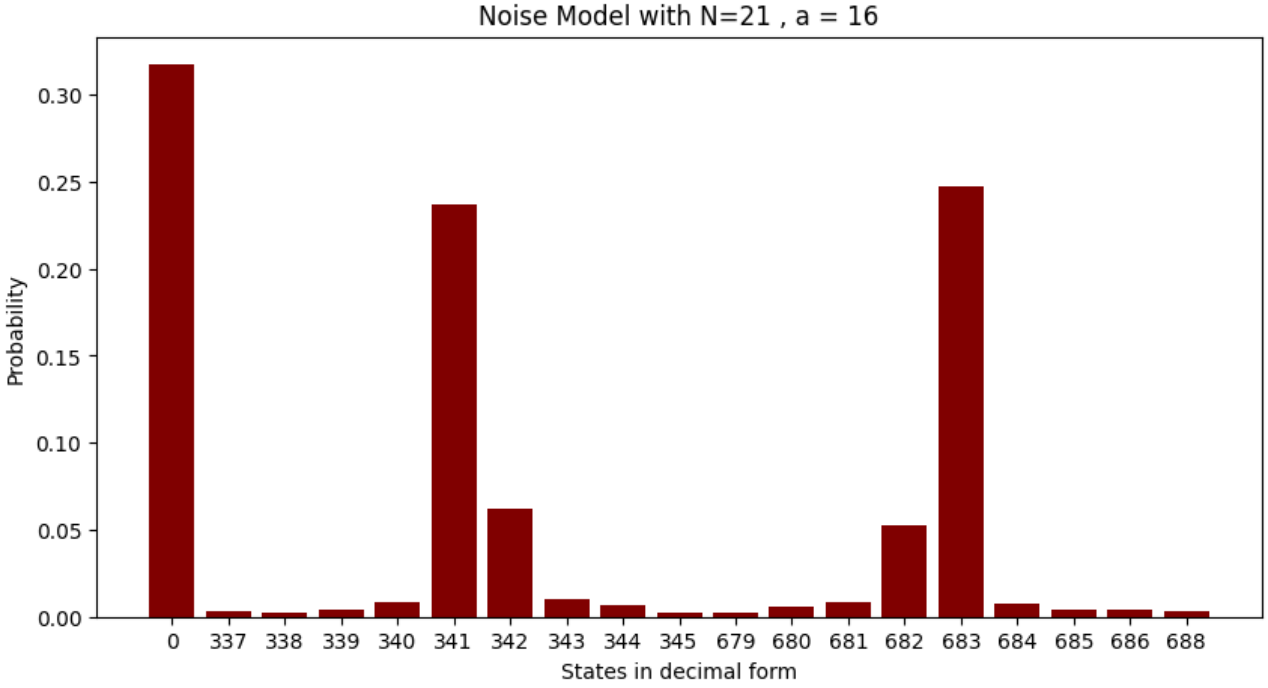

N=21 , a = 8
Probability
0.0
0.1
0.2
0.3
0.4
0.5
0
512
States in decimal form


Noise Model with N=21 , a = 8
Probability
0.0
0.1
0.2
0.3
0.4
0.5
0
512
States in decimal form


N=21 , a = 13
Probability
0.0
0.1
0.2
0.3
0.4
0.5
0
512
States in decimal form

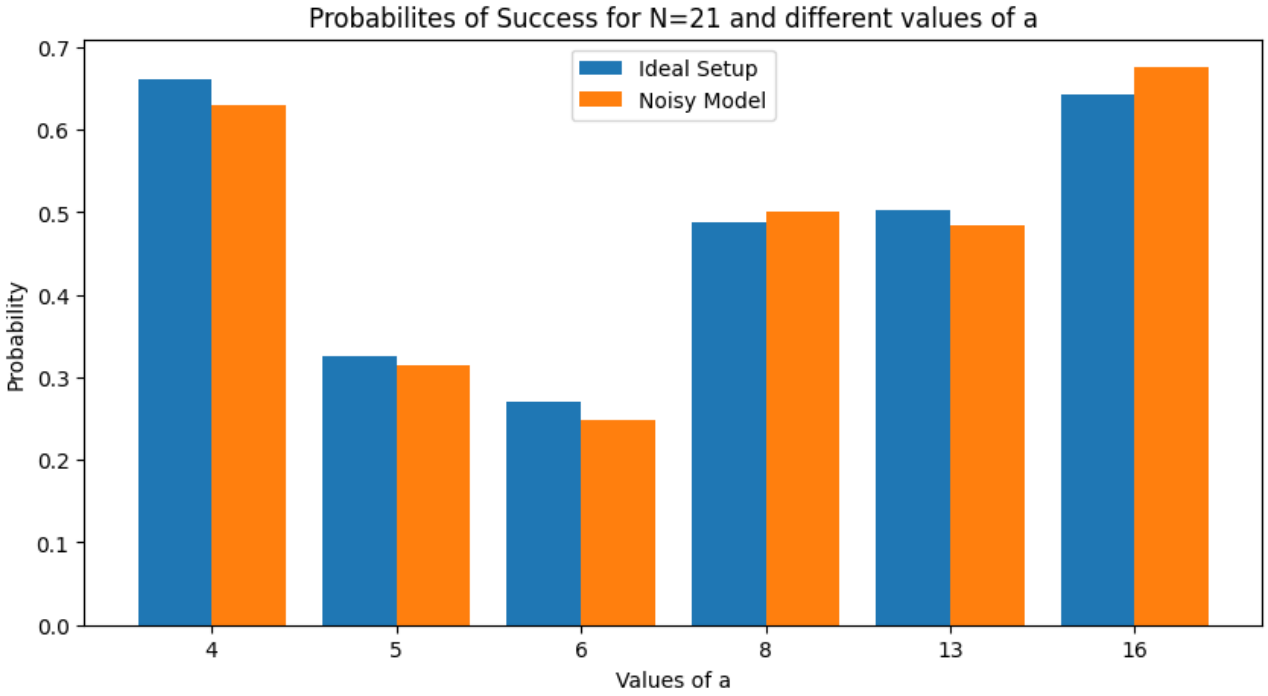

Noise Model with N=21 , a = 13
Probability
0.0
0.1
0.2
0.3
0.4
0.5
0
512
States in decimal form


N=21 , a = 16
Probability
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0 332 338 339 340 341 342 343 351 680 681 682 683 684 685 686 688 690
States in decimal form


Noise Model with N=21 , a = 16
Probability
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0 337 338 339 340 341 342 343 344 345 679 680 681 682 683 684 685 686 688
States in decimal form

3.1 Probability of successfully finding correct factors

We further compute the success probability for each implementation of 'a'. The data is illustrated as follows:



Discussion

The key result of our project is Fig. 3.1, which shows that the probability of successfully finding the correct prime factors is dependent upon the vale of 'a' chosen and its implementation constructed. We see that $a = 4$ and $a = 16$ yielded the highest success probability beyond 60% even after incorporating noise. This was followed by $a = 13$, $a = 8$, $a = 5$ with $a = 6$ yielding the least success probability. We could further link this success probability with the circuit implementations to draw possible factors affecting the success probability. The two major dependent factors which seem to play a crucial role in determining success probability are as follows:

1. The number of total gates used
2. The number of NOT gates used

The implementations of $a = 4$ and $a = 5$ which yielded the smallest success probability both utilised larger number of total gates and larger number of NOT gates as compared to rest of the implementations. Whereas, the implementations of $a = 4$ and $a = 13$ which yielded highest success probability utilised lesser number of total gates and zero number of NOT gates.

Conclusion

Starting off with the mathematical formulation of RSA cryptosystem and Shor's algorithm, we explained how Shor's algorithm can be used to break this encryption mechanism by computing prime factors in polynomial time - the key factor on which public key encryption relies. We further exemplified the general quantum circuit implementation of Shor's algorithm and constructed specific implementations for $N = 21$ and different values of 'a'. The results generated allowed us to analyse that success probability for Shor's algorithm implementation relies on the number of total gates used and the number of NOT gates used.

References

[1] Whitfield Diffie and Martin Hellman. "New directions in cryptography". In: *IEEE transactions on Information Theory* 22.6 (1976), pp. 644–654.

[2] David McMahon. *Quantum computing explained*. John Wiley & Sons, 2007.

[3] Hamed Mohammadbagherpoor et al. "Experimental challenges of implementing quantum phase estimation algorithms on ibm quantum computer". In: *arXiv preprint arXiv:1903.07605* (2019).

[4] Chris Monroe et al. "Demonstration of a fundamental quantum logic gate". In: *Physical review letters 75.25* (1995), p. 4714.

[5] Michael A Nielsen and Isaac Chuang. *Quantum computation and quantum information*. 2002

[6] Ronald L Rivest, Adi Shamir, and Leonard Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[7] Peter W Shor. "Algorithms for quantum computation: discrete logarithms and factoring". In: *Proceedings 35th annual symposium on foundations of computer science. Ieee*. 1994, pp. 124–134.

[8] Yaakov S Weinstein et al. "Implementation of the quantum Fourier transform". In: *Physical review letters* 86.9 (2001), p. 1889.

[9] Skosana, U., Tame, M. "Demonstration of Shor's factoring algorithm for N = 21 on IBM quantum processors". *Sci Rep 11, 16599* (2021).

Qiskit Implementation

```
import matplotlib.pyplot as plt
import numpy as np
from qiskit import QuantumCircuit, Aer, transpile
from qiskit.visualization import plot_histogram
from math import gcd
from numpy.random import randint
import pandas as pd
from fractions import Fraction
from qiskit.quantum_info import Kraus, SuperOp
from qiskit_aer import AerSimulator
from qiskit_aer.noise import (NoiseModel, QuantumError, ReadoutError,
    pauli_error, depolarizing_error, thermal_relaxation_error)
```

Listing 1: Imports

```
def c_amod21(a, power):
    """Controlled multiplication by a mod 15"""
    if a not in [4,5,6,8,13,16,17]:
        raise ValueError("'a' must be 4,5,6,8,13,16 or 17")
    U = QuantumCircuit(5)
    if a == 4:
        for _iteration in range(power):
            U.swap(0,4)
            U.swap(4,2)
    if a == 5:
        for _iteration in range(power):
            U.x(0)
            U.x(2)
            U.x(4)
            U.swap(4,2)
            U.swap(0,4)
    if a == 8:
```

```
        for _iteration in range(power):
            U.swap(0,3)
    if a == 17:
        for _iteration in range(power):
            U.swap(4,2)
            U.swap(0,4)
    if a == 13:
        if power%2 == 1:
            U.x(2)
            U.x(3)
    if a == 16:
        for _iteration in range(power):
            U.swap(4,2)
            U.swap(0,4)
    if a == 6:
        if power%2 == 1:
            U.x(0)
            U.x(1)
            U.x(2)
        else:
            U.x(1)
            U.x(2)
            U.x(3)
    U = U.to_gate()
    U.name = f"{a}^{power} mod 21"
    c_U = U.control()
    return c_U
```

Listing 2: Implementations for different values of 'a'

```
def qft_dagger(n):
    """n-qubit QFTdagger the first n qubits in circ"""
    qc = QuantumCircuit(n)       #create circuit
    # Don't forget the Swaps!
    for qubit in range(n//2):
        qc.swap(qubit, n-qubit-1)
    for j in range(n):
        for m in range(j):
            qc.cp(-np.pi/float(2**(j-m)), m, j)         # Applying controled phase
    gate with the specified phase
        qc.h(j)                                         # hadamard application
    qc.name = "QFT\dagger"                                  # what shows on the
    circuit
    return qc
```

Listing 3: Quantum Fourier Transform

```
# Specify variables
N_COUNT = 10  # number of counting qubits / control qubits
#Enter Value of a from 4,5,6,8,13,16 or 17
a = 4
qc = QuantumCircuit(N_COUNT + 5, N_COUNT)
# Initialize counting qubits
# in state |+> by application of hadamard gate
for q in range(N_COUNT):
    qc.h(q)
# And auxiliary register in state |1> by Not gate
qc.x(N_COUNT)
# Do controlled-U operations
for q in range(N_COUNT):
    qc.append(c_amod21(a, 2**q),
              [q] + [i+N_COUNT for i in range(5)])
# Do inverse-QFT
qc.append(qft_dagger(N_COUNT), range(N_COUNT))
# Measure circuit
```

```
qc.measure(range(N_COUNT), range(N_COUNT))
qc.draw(fold=-1)  # -1 means 'do not fold'
```

Listing 4: Assembling the quantum circuit

```
aer_sim = Aer.get_backend('aer_simulator')
t_qc = transpile(qc, aer_sim)
counts = aer_sim.run(t_qc).result().get_counts()
plot_histogram(counts)
# Just to unpack the data in 'counts'
mylist = list(counts.items())
ctt = pd.DataFrame(mylist)
avg = np.mean(ctt[1])
c = []                                          # stores counts
v = []                                          # stores values in decimals
period = []
for i in range(len(mylist)):
    if mylist[i][1] >= avg:
        c.append(mylist[i][1])
        v.append(int(mylist[i][0], 2))
# Finding period by the method of continued fractions
for i in range(len(v)):
    frac = Fraction(v[i]/1024).limit_denominator(21)
    s, r = frac.numerator, frac.denominator
    period.append(r)
for i in range(len(v)):
    if v[i]!=0:
        print('Peak on the value of ', v[i] ,' with counts ',c[i], ' corresponds to
    period of ', period[i])
# Finding the probability of correct period i.e correct prime factors found
if a == 4:
    correct_period = 3
if a == 5:
    correct_period = 6
if a == 8:
    correct_period = 2
if a == 13:
    correct_period = 2
if a == 17:
    correct_period = 2
if a == 16:
    correct_period = 3
if a == 6:
    correct_period = 2
mylist = list(counts.items())
ctt = pd.DataFrame(mylist)
avg = np.mean(ctt[1])
c = []
v = []
period = []
for i in range(len(mylist)):
    c.append(mylist[i][1])
    v.append(int(mylist[i][0], 2))
for i in range(len(v)):
    frac = Fraction(v[i]/1024).limit_denominator(21)
    s, r = frac.numerator, frac.denominator
    period.append(r)
s=0
for i in range(len(period)):
    if period[i] == correct_period:
        s += c[i]
# for i in range(len(v)):
#     if v[i] == 0:
#         zero_counts = c[i]
# if len(v) == 2:
```

```
#     print('Probability of finding the correct period: 100 %')
print('Probability of finding the correct period: ', (s*100)/1024 , '%')
for i in counts:
    measured_value = int(i[::-1], 2)
    if measured_value % 2 != 0:
        #print("Measured value not even")
        continue #measured value should be even as we are doing a^(r/2) mod N and r/2 should be int
    x = int(np.power(4, measured_value/2) % N)
    if (x + 1) % N == 0:
        continue
    factor_one = gcd(x + 1, N)
    factor_two = gcd(x - 1, N)
    if factor_one == N:
        continue
    if factor_two == N:
        continue
    if factor_one == 1 and factor_two == 1:
        continue
    if factor_one != 1:
        factor_two = N//factor_one
    if factor_two != 1:
        factor_one = N//factor_two
    print("Measured value = ", measured_value, " leads to the factors =", factor_one, factor_two)
```

Listing 5: Generating Results

```
import numpy as np
from qiskit import QuantumCircuit, transpile
from qiskit.quantum_info import Kraus, SuperOp
from qiskit_aer import AerSimulator
from qiskit.tools.visualization import plot_histogram
# Import from Qiskit Aer noise module
from qiskit_aer.noise import (NoiseModel, QuantumError, ReadoutError,
    pauli_error, depolarizing_error, thermal_relaxation_error)
# Example error probabilities
p_reset = 0.5
p_meas = 0.5
p_gate1 = 0.5
# QuantumError objects
error_reset = pauli_error([('X', p_reset), ('I', 1 - p_reset)])
error_meas = pauli_error([('X',p_meas), ('I', 1 - p_meas)])
error_gate1 = pauli_error([('X',p_gate1), ('I', 1 - p_gate1)])
error_gate2 = error_gate1.tensor(error_gate1)
# Add errors to noise model
noise_bit_flip = NoiseModel()
noise_bit_flip.add_all_qubit_quantum_error(error_reset, "reset")
noise_bit_flip.add_all_qubit_quantum_error(error_meas, "measure")
noise_bit_flip.add_all_qubit_quantum_error(error_gate1, ["u1", "u2", "u3"])
noise_bit_flip.add_all_qubit_quantum_error(error_gate2, ["cx"])
print(noise_bit_flip)
# Create noisy simulator backend
sim_noise = AerSimulator(noise_model=noise_bit_flip)
qc_noise = transpile(qc, sim_noise)
noise_counts = aer_sim.run(qc_noise).result().get_counts()
plot_histogram(noise_counts)
```

Listing 6: Noise Incorporation

```
data = pd.DataFrame(counts.items(), columns =['Value', 'Counts'])
data_srt = data.sort_values(by = 'Value')
data_srt = data_srt.reset_index(drop=True)
# mylist = list(data_srt)
# print(data_srt)
```

```
state = []
prob = []
for i in range(len(data_srt)):
    if data_srt.loc[i][1] > 1:
        state.append(str (int(data_srt.loc[i][0], 2) ))
        prob.append(data_srt.loc[i][1]/1024)
fig = plt.figure(figsize = (10, 5))
# creating the bar plot
plt.bar(state, prob, color ='darkblue')
plt.xlabel("States in decimal form")
plt.ylabel("Probability")
plt.title("N=21 , a = 17")
plt.show()
fig.savefig('a_17', bbox_inches='tight')
noise_data = pd.DataFrame(noise_counts.items(), columns =['Value', 'Counts'])
noise_data_srt = noise_data.sort_values(by = 'Value')
noise_data_srt = noise_data_srt.reset_index(drop=True)
n_state = []
n_prob = []
for i in range(len(noise_data_srt)):
    if noise_data_srt.loc[i][1] > 1:
        n_state.append(str (int(noise_data_srt.loc[i][0], 2) ))
        n_prob.append(noise_data_srt.loc[i][1]/1024)
fig = plt.figure(figsize = (10, 5))
# creating the bar plot
plt.bar(n_state, n_prob, color ='maroon')
plt.xlabel("States in decimal form")
plt.ylabel("Probability")
plt.title("Noise Model with N=21 , a = 17")
plt.show()
fig.savefig('noise_a_17', bbox_inches='tight')
mylist = list(noise_counts.items())
ctt = pd.DataFrame(mylist)
avg = np.mean(ctt[1])
c = []
v = []
period = []
for i in range(len(mylist)):
    c.append(mylist[i][1])
    v.append(int(mylist[i][0], 2))
# print(c)
# print(v)
for i in range(len(v)):
    frac = Fraction(v[i]/1024).limit_denominator(21)
    s, r = frac.numerator, frac.denominator
    period.append(r)
s=0
for i in range(len(period)):
    if period[i] == correct_period:
        s += c[i]
# for i in range(len(v)):
#     if v[i] == 0:
#         zero_counts = c[i]
# if len(v) == 2:
#     print('Probability of finding the correct period: 100 %')
print('Probability of finding the correct period: ', (s*100)/1024 , '%')
#print(zero_counts)
p_n_17 = s/1024
prob_a = [p_a_4, p_a_5, p_a_6, p_a_8, p_a_13, p_a_16]
prob_a_n = [p_n_4, p_n_5, p_n_6, p_n_8, p_n_13, p_n_16]
a_value = [4, 5, 6, 8, 13, 16]
# print(p_a_4, p_n_4)
# print(p_a_5, p_n_5)
# print(p_a_6, p_n_6)
# print(p_a_8, p_n_8)
```

```
# print(p_a_13, p_n_13)
# print(p_a_16, p_n_16)
X_axis = np.arange(len(a_value))
fig = plt.figure(figsize = (10, 5))
plt.bar(X_axis - 0.2, prob_a, 0.4, label = 'Ideal Setup')
plt.bar(X_axis + 0.2, prob_a_n, 0.4, label = 'Noisy Model')
plt.xticks(X_axis, a_value)
plt.xlabel("Values of a")
plt.ylabel("Probability")
plt.title("Probabilites of Success for N=21 and different values of a")
plt.legend()
plt.show()
fig.savefig('ideal_vs_noise', bbox_inches='tight')
```

Listing 7: Generating Noise Results