CentraleSupélec

Applications of Quantum Calculus: Shor's algorithm and probability distributions

2nd year Long Project

Students:
Ahmed BEN AISSA
Elie MOKBEL
Henrique MIYAMOTO
Pierre MINSSEN

Supervisor:
Prof. Benoît VALIRON

June 11, 2019

Contents

1 Introduction 2

2 Basic concepts 3
2.1 Qubits 3
2.2 Bloch sphere 3
2.3 Measurements 4
2.4 Unitary operations 4
2.5 Tensor product 4
2.6 Many qubits systems 5
2.7 Quantum entanglement 6
2.8 Quantum circuits 6

3 Shor's algorithm 7
3.1 Factorisation 7
3.2 Order finding 8
3.3 Quantum Fourier transform 9
3.4 Quantum phase estimation 10
3.5 Continued fraction expansion 12

4 Oracle circuit 14
4.1 Adder gate 14
4.2 Modular adder gate 15
4.3 Controlled multiplier gate 16
4.4 Controlled U gate 17

5 Results and discussion 18
5.1 Factorisation 18
5.2 Complexity Analysis 18
5.3 Discussion 19

6 Probability superpositions 21
6.1 Proposed implementation 22
6.2 Discussion 24

7 Final considerations 26

A Python codes 27

References 39

1 Introduction

TO DO

2 Basic concepts

2.1 Qubits

Classical computers operate on strings of bits (0 or 1) and produce other strings of bits. Classical data is supposed to be clonable, erasable, readable and not supposed to change when left untouched.

In quantum computation, on the other hand, the bits are replaced by *quantum bits* or *qubits*, which are unitary elements of the 2-dimensional complex Hilbert space $\mathbb{C}^2$. We choose the orthonormal basis called *computational basis*

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \text{ and } |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

A general qubit can be seen a superposition of states $|0\rangle$ and $|1\rangle$

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \tag{2.1}$$

where $\alpha, \beta \in \mathbb{C}$.

The Hilbert space $\mathbb{C}^2$ is provided with an *inner product* $\langle\varphi|\psi\rangle = |\varphi\rangle^\dagger|\psi\rangle = \sum_i \overline{\varphi}_i\psi_i$, which allows one to the define the *norm* of a state $\||\psi\rangle\| = \sqrt{\langle\psi|\psi\rangle}$ and *orthogonality* between two states when $\langle\varphi|\psi\rangle = 0$.

2.2 Bloch sphere

The Bloch sphere is a representation of quantum states on S^2. Let us consider the qubit in state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ and the polar representations $\alpha = |\alpha|e^{i\gamma} = \cos\frac{\theta}{2}e^{i\gamma}$ and $\beta = |\beta|e^{i(\gamma+\varphi)} = \sin\frac{\theta}{2}e^{i(\gamma+\varphi)}$. Then, we can write $|\psi\rangle = e^{i\gamma}\left(\cos\frac{\theta}{2}|0\rangle + e^{i\varphi}\sin\frac{\theta}{2}|1\rangle\right)$. Neglecting the global phase factor $e^{i\gamma}$[1], we have the mapping

$$(\theta,\varphi) \mapsto \left(\cos\frac{\theta}{2},\ e^{i\varphi}\sin\frac{\theta}{2}\right), \tag{2.2}$$

with $\theta \in [0\,\pi]$ and $\varphi \in [0, 2\pi[$.

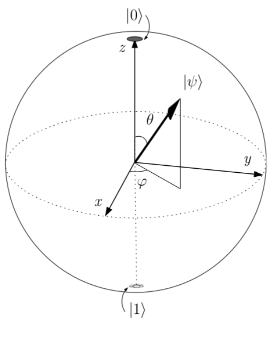


Figure 2.1: Bloch sphere [2].

[1]One reason for doing so is that this factor does not change the modulus squared of amplitudes $|\alpha|^2$ and $|\beta|^2$ [2].

2.3 Measurements

It is a probabilistic operation that allows one to recover some classical information. The measurement of $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ returns $|0\rangle$ with probability $|\alpha|^2$ and $|1\rangle$ with probability $|\beta|^2$. It also alters the state of a qubit and forces it to collapse to state $|0\rangle$ or $|1\rangle$, respectively. In this case, we say the measurement was done against the computational basis $\{|0\rangle, |1\rangle\}$.

2.4 Unitary operations

The temporal evolution of an isolated quantum system is described by linear transformations, represented by matrices. Transformations that map unitary vector onto unitary vectors are called *unitary transformations U* and can be defined by the following property:

$$U^\dagger U = UU^\dagger = I,$$

where $U^\dagger = \left(\overline{U}\right)^T$ is the adjoint matrix and I is the identity. These are reversible operations.

Some usual gates are NOT, Hadamard, phase-shift and phase-flip:

- NOT
$$N = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$
- Hadamard
$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$
- Phase-shift
$$V_\theta = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}$$
- Phase-flip[2]
$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

2.5 Tensor product

The tensor product between two states

$$|\psi\rangle = \begin{pmatrix} \psi_1 \\ \vdots \\ \psi_m \end{pmatrix} \text{ and } |\varphi\rangle = \begin{pmatrix} \varphi_1 \\ \vdots \\ \varphi_p \end{pmatrix}$$

[2]Note that, in fact, $Z = V_\pi$.

is computed as

$$|\psi\rangle \otimes |\varphi\rangle = \begin{pmatrix} \psi_1\varphi_1 \\ \vdots \\ \psi_1\varphi_p \\ \psi_2\varphi_1 \\ \vdots \\ \psi_2\varphi_p \\ \vdots \\ \psi_m\varphi_1 \\ \vdots \\ \psi_m\varphi_p \end{pmatrix}.$$

In general, given two matrices $A \in \mathbb{C}^{m\times n}$ and $B \in \mathbb{C}^{p\times q}$, the tensor product is the matrix $A \otimes B \in \mathbb{C}^{mp\times nq}$ given by

$$A \otimes B = \begin{pmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{pmatrix} \tag{2.3}$$

where a_{ij} is the (i,j)-element of A.

Given two linear transformations A and B, we can define a new linear mapping by

$$(A \otimes B)(|u\rangle \otimes |v\rangle) = A|u\rangle \otimes B|v\rangle. \tag{2.4}$$

Notation:

- We can indistinguishably write $|\psi\rangle \otimes |\varphi\rangle = |\psi\rangle|\varphi\rangle = |\psi,\varphi\rangle = |\psi\varphi\rangle$.
- $|\psi\rangle^{\otimes n} = \underbrace{|\psi\rangle \otimes \cdots |\psi\rangle}_{n \text{ times}}$ and $A^{\otimes n} = \underbrace{A \otimes \cdots A}_{n \text{ times}}$.

2.6 Many qubits systems

The state of a 2-qubit is an element of the tensor product space $\mathbb{C}^4 = \mathbb{C}^2 \otimes \mathbb{C}^2$, which is spanned by

$$|00\rangle = \begin{pmatrix}1\\0\\0\\0\end{pmatrix}, \quad |10\rangle = \begin{pmatrix}0\\1\\0\\0\end{pmatrix}, \quad |01\rangle = \begin{pmatrix}0\\0\\1\\0\end{pmatrix}, \quad |11\rangle = \begin{pmatrix}0\\0\\0\\1\end{pmatrix}.$$

A generic state of 2 qubits it therefore of the form

$$|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle$$

with $|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1$.

In general, writing states as the decimal number correspoding to the binary representation (e.g. $|11\rangle \to |3\rangle$), a n-qubit state is described as

$$|\psi\rangle = \sum_{i=0}^{2^n-1} \alpha_i |i\rangle, \text{ with } \sum_{i=0}^{2^n-1} |\alpha_i|^2 = 1.$$

As a generalisation of the 1-qubit case, the measurement of a n-qubit state changes it and forces it to collapse to one of the possible $|i\rangle$ states, each of which is measured with probability $|\alpha_i|^2$.

Among unitary operations available to 2-qubits states, we have the swap gate X and the control-not gate N_C:

$$X = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad N_C = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

The control-not gate changes the state of the second qubit only if the first qubit is in the state $|1\rangle$. It implements the mapping $|xy\rangle \mapsto |x\rangle \otimes |x \oplus y\rangle$.

The Toffoli gate acts on 3 qubits and is a "control-control-not": it implements the function $|xyz\rangle \mapsto |xy\rangle \otimes |z \oplus xy\rangle$, i.e., it changes the state of the last qubit if the two first qubits are in state $|11\rangle$.

2.7 Quantum entanglement

Consider the states $|\psi\rangle = a|0\rangle + b|1\rangle$ and $|\varphi\rangle = c|0\rangle + d|1\rangle$. Their tensor product is $|\psi\varphi\rangle = ac|00\rangle + ad|01\rangle + bc|10\rangle + bd|11\rangle$. It turns out that a general state is not on this form, unless it has $\alpha\delta = \beta\gamma$.

We say a quantum state is *entangled* when it cannot be written as a tensor product of two other states. For example, $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ cannot have such a decomposition, and hence is an entangled state.

2.8 Quantum circuits

Quantum circuits are graphical representations of a procedure, i.e., a sequence of logical operations performed on a system. Unlike classical circuits, the wires must not be regarded as physical connections and their components are not available "on the shelf".

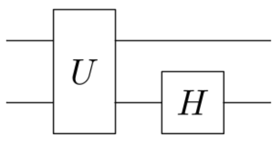


Figure 2.2: Example of quantum circuit representing $|\psi\rangle \mapsto (I \otimes H)(U|\psi\rangle)$ [1].

3 Shor's algorithm

The advantage of quantum algorithms over classical ones appears when using some quantum property such as entanglement or the interference, brought by complex coefficients.

In fact, quantum algorithms are based on a few "real" quantum constructions, such as quantum Fourier transform, quantum walk and amplitude amplification. The rest is composed of classical analysis and possibly an *oracle*: a quantum circuit corresponding to a reversible operation.

Some interesting problems in algebra and number theory reduce to the problem of order finding. For example, the problem of factorising an integer number, which is addressed by Shor's algorithm [4].

3.1 Factorisation

The objective of the *factorisation problem* is to factorise a big number N into prime numbers. Note that at least $n = \lceil \log_2 N \rceil$ qubits are needed to store N and that n is the maximum number of prime factors. We will show how this problem reduces to the problem of finding the order of a randomly generated integer $x < N$ generated randomly.

If N is even, 2 is trivially a factor. In addition, if x and N have common factors, then $\gcd(x, N)$ gives a factor of N; so we focus on investigating the case when x and N are coprimes.

Definition 3.1. *The order of x modulo N is the least positive integer r such that*

$$x^r \equiv 1 \mod N. \tag{3.1}$$

Theorem 3.1 (Euler's Theorem)**.** *If x and N are coprime positive integers, then*

$$x^{\varphi(N)} \equiv 1 \mod N, \tag{3.2}$$

where $\varphi(N)$ is the Euler's totient function, i.e., it indicates the number of coprimes to N which are less or equal to it.

The *order finding problem* is to find r, given x and N coprimes. The algorithm is built up on the following two theorems.

Theorem 3.2. *Let N be a composite number stored with n qubits and x be non-trivial solution of the equation $x^2 \equiv 1 \mod N$ in the range $1 \leq x \leq N$, i.e., $x \not\equiv \pm 1 \mod N$. Then at least one of $\gcd(x+1, N)$ and $\gcd(x-1, N)$ is a non-trivial factor of N.*

Proof. Note that $x^2 \equiv 1 \mod N \Leftrightarrow x^2 - 1 \equiv 0 \mod N \Leftrightarrow (x+1)(x-1) \equiv 0 \mod N$, which means that N is a divisor of $(x+1)(x-1)$. If $1 < x < N-1$, then $0 < x-1 < x+1 < N$ and N cannot be a divisor of $(x+1)$ neither of $(x-1)$ separately. So both $(x+1)$ and $(x-1)$ must have factors of N. In this case, at least one of $\gcd(x+1, N)$ and $\gcd(x-1, N)$ produce a non-trivial factor of N[3]. □

[3]Such factor can be computed by using Euclid's algorithm.

Theorem 3.3. *Suppose $N = p_1^{\alpha_1}...p_m^{\alpha_m}$ is the prime factorisation of and odd composite positive integer. Let x be an integer chose uniformly at random such that $1 \leq x \leq N-1$ and $\gcd(x, N) = 1$. Let r be the order of x modulo N. Then*

$$\Pr\{r \textit{ is even and } x^{r/2} \not\equiv \pm 1 \mod N\} \geq 1 - \frac{1}{2^m}. \tag{3.3}$$

Proof. See [5], pp. 751-752[4]. □

The algorithm runs as follows.

Algorithm 1: Shor's algorithm for factorising N.

```
1 If $N$ is even, return the factor 2.
2 Randomly choose $x$ such that $1 \leq x \leq N-1$.
3 If $\gcd(x, N) \neq 1$, return the factor $\gcd(x, N)$.
4 Find the order $r$ of $x$ modulo $N$.
5 If $r$ is even and $x^{r/2} \not\equiv \pm 1 \mod N$, then $\gcd(x^{r/2}+1, N)$ and
    $\gcd(x^{r/2}-1, N)$ are non-trivial factors.
6 Else, restart the algorithm.
```

Some remarks:

- The method fails if N is the power of a prime number. But in this case there exists a a classical algorithm to solve the problem [2].
- The problem is finding the order of x modulo N, for which there is no efficient classical procedure available. This problem is addressed in the next part.

3.2 Order finding

The problem of order finding, i.e., given x and N coprimes, to find r such that $x^r \equiv 1 \mod N$ is related to the matrix eigendecomposition. A unitary U, being a Hermitian matrix, can be decomposed as

$$U = \sum_j \lambda_j u_j u_j^\dagger, \tag{3.4}$$

where u_j are orthonormal eigenvectors and λ_j are the associated eigenvalues.

Let us assume that $N = 2^n$ and consider the operation U_x on n qubits that implements the mapping

$$U_x : \quad |j\rangle \mapsto |j \cdot x \mod N\rangle. \tag{3.5}$$

The operator is unitary because x and N are coprimes and the image of $\{0, ..., N-1\}$ is the whole set. In particular, U_x^k sends $|j\rangle \mapsto |j \cdot x^k \mod N\rangle$, so if $x^r \equiv 1 \mod N$, the map U_x^r is the identity map.

[4]Actually, these authors use a slightly different bound for the probability: $1 - 1/2^{m-1}$.

The eigenstates of U are of the [5]

$$|u_s\rangle = \frac{1}{\sqrt{r}}\sum_{k=0}^{r-1}\exp\left(-\frac{2\pi i s k}{r}\right)|x^k \mod N\rangle \tag{3.6}$$

for $0 \le s \le r-1$. The eigenvalues are the r-th roots of the unity, having the form $e^{2\pi i s/r}$, since

$$U|u_s\rangle = \exp\left(\frac{2\pi i s}{r}\right)|u_s\rangle. \tag{3.7}$$

An algorithm for finding such eigenvalues should be enough to find the order r of x modulo N[6].

3.3 Quantum Fourier transform

The Fourier transform is an operation that can be performed faster in quantum computers. The quantum Fourier transform (QFT) [6] is defined in analogy with the discrete Fourier transform. It is the linear mapping:

$$\begin{aligned}\hat{f} \,:\, \mathbb{C}^N &\to \mathbb{C}^N \\ |j\rangle &\mapsto \frac{1}{\sqrt{N}}\sum_{k=0}^{N-1} e^{2\pi i jk/N}|k\rangle \end{aligned}\tag{3.8}$$

Let us consider $N = 2^n$ with orthonormal basis $\{|0\rangle, \dots, |2^n-1\rangle\}$. We shall use the *binary representation* $j =: j_1 j_2 \dots j_n$ to represent $j = \sum_{i=1}^n j_i 2^{n-i}$ and the *binary fraction* $j =: 0.j_1 j_2 \dots j_n$ to represent $j = \sum_{i=1}^n j_i 2^{-i}$.

It will be useful to consider an alternative form of the QFT which is indeed so important that could be considered its definition itself:

$$|j_1 \dots j_n\rangle \mapsto \frac{\left(|0\rangle + e^{2\pi i 0.j_n}|1\rangle\right)\otimes\left(|0\rangle + e^{2\pi i 0.j_{n-1}j_n}|1\rangle\right)\otimes\cdots\otimes\left(|0\rangle + e^{2\pi i 0.j_1\dots j_n}|1\rangle\right)}{2^{n/2}}. \tag{3.9}$$

The equivalence between the two expressions can be shown as follows [3]:

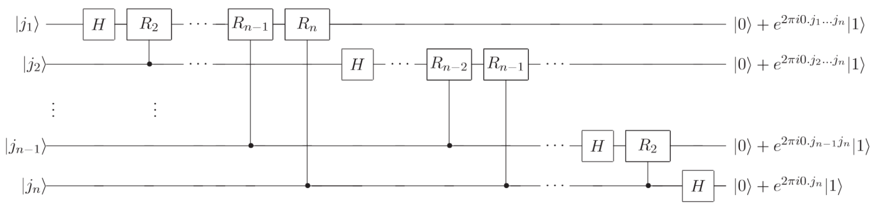
$$\begin{aligned}
&\frac{1}{2^{n/2}}\sum_{k=0}^{2^n-1} e^{2\pi i jk/2^n}|k\rangle \\
&= \frac{1}{2^{n/2}}\sum_{k_1=0}^{1}\cdots\sum_{k_n=0}^{1} e^{2\pi i j(\sum_{l=1}^n k_l/2^l)}|k_1\dots k_n\rangle \qquad \text{(write in binary representation)} \\
&= \frac{1}{2^{n/2}}\sum_{k_1=0}^{1}\cdots\sum_{k_n=0}^{1}\prod_{l=1}^{n}\otimes e^{2\pi i j k_l/2^l}|k_l\rangle \qquad \text{(tensor product decomposition)} \\
&= \frac{1}{2^{n/2}}\prod_{l=1}^{n}\otimes\left(\sum_{k_l=0}^{1} e^{2\pi i j k_l/2^l}|k_l\rangle\right) \qquad \text{(factorise the binary powers)} \\
&= \frac{1}{2^{n/2}}\prod_{l=1}^{n}\otimes\left(|0\rangle + e^{2\pi i j/2^l}|1\rangle\right).
\end{aligned}$$

[5]To simplify notation, hereafter we shall denote U_x by U simply.

[6]An alternative unitary could have been used: $V|j\rangle|k\rangle = |j\rangle|k + x^j \mod N\rangle$.

The product representation allows one to implement the quantum circuit for the QFT using Hadamard and R_k rotation gates, the latter being of the form[7]

$$R_k := \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{pmatrix}. \tag{3.10}$$



$|j_1\rangle$
$|j_2\rangle$
$|j_{n-1}\rangle$
$|j_n\rangle$
H
R_2
R_{n-1}
R_n
R_{n-2}
$|0\rangle + e^{2\pi i 0.j_1 \dots j_n}|1\rangle$
$|0\rangle + e^{2\pi i 0.j_2 \dots j_n}|1\rangle$
$|0\rangle + e^{2\pi i 0.j_{n-1} j_n}|1\rangle$
$|0\rangle + e^{2\pi i 0.j_n}|1\rangle$

Figure 3.1: Quantum circuit that implements the QFT [3].

To conclude this section, we remark that the QFT, as a linear operation, may be written in matrix form[7]:

$$F = \frac{1}{\sqrt{N}} \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{N-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(N-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{N-1} & \omega_n^{2(N-1)} & \omega_n^{3(N-1)} & \cdots & \omega_n^{(N-1)(N-1)} \end{pmatrix} \tag{3.11}$$

where $N = 2^n$ and $\omega_n := e^{2\pi i/2^n}$. This allows one to verify that $FF^\dagger = F^\dagger F = I$, therefore concluding that the Fourier transform is a unitary transformation.

3.4 Quantum phase estimation

Suppose a unitary operator U has an eigenvector $|u\rangle$ with eigenvalue $e^{2\pi i\varphi}$, where φ is unknown. The objective of the *phase estimation subroutine* is to estimate the value of φ. We assume we have oracles capable of preparing the state $|u\rangle$ and performing controlled-U^{2^j} operations.

The procedure uses two registers: the first contains t qubits initially in state $|0\rangle$. The number t depends on the desired accuracy for φ and on the probability we want the procedure to be successful. The second register begins with the state $|u\rangle$ and contains as many qubits as necessary to store it.

The first step is to apply the following circuit to the registers.

The final state of the first register is

$$\frac{1}{2^{t/2}}\left(|0\rangle + e^{2\pi i 2^{t-1}\varphi}|1\rangle\right)\left(|0\rangle + e^{2\pi i 2^{t-2}\varphi}|1\rangle\right)\dots\left(|0\rangle + e^{2\pi i 2^{0}\varphi}|1\rangle\right)$$
$$= \frac{1}{2^{t/2}}\sum_{k=0}^{2^t-1} e^{2\pi i\varphi k}|k\rangle.$$

[7]The attentive reader will notice that $R_k = V_{2\pi/2^k}$.

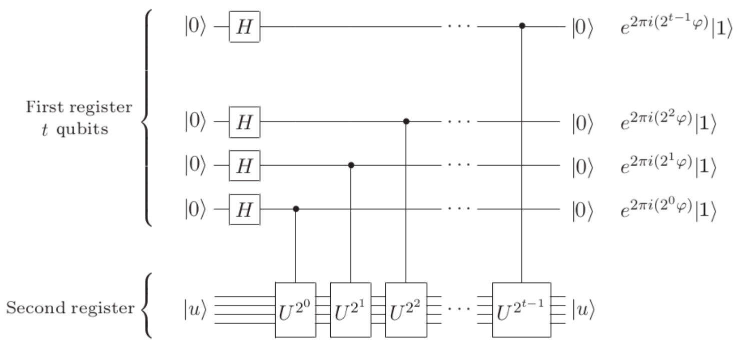

First register
t qubits
Second register
$|0\rangle$
H
$|u\rangle$
U^{2^0}
U^{2^1}
U^{2^2}
$U^{2^{t-1}}$
$|0\rangle\ e^{2\pi i(2^{t-1}\varphi)}|1\rangle$
$|0\rangle\ e^{2\pi i(2^{2}\varphi)}|1\rangle$
$|0\rangle\ e^{2\pi i(2^{1}\varphi)}|1\rangle$
$|0\rangle\ e^{2\pi i(2^{0}\varphi)}|1\rangle$

Figure 3.2: Quantum circuit for first step of phase estimation (normalisation has been omitted on the right side) [3].

The second step is to apply the inverse Fourier transform on the first register[8] and the third step is to measure it. Note that, if we write the previous expression using the binary fraction representation, we have

$$\frac{1}{2^{t/2}}\left(|0\rangle + e^{2\pi i 0.\varphi_t}|1\rangle\right)\left(|0\rangle + e^{2\pi i 0.\varphi_{t-1}\varphi_t}|1\rangle\right)\dots\left(|0\rangle + e^{2\pi i 0.\varphi_1\dots\varphi_t}|1\rangle\right).$$

Comparing it with the (product) expression for the Fourier transform, one can see that the result of applying the inverse Fourier transform is the state $|\varphi_1\dots\varphi_r\rangle$ and a measurement in the computational basis gives exactly an estimation for φ!

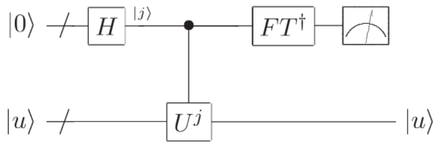


Figure 3.3: Schematic of the overall phase estimation subroutine [3] .

Summarising, the phase estimation subroutine gives an estimation $\tilde{\varphi}$ to the phase of an eigenvalue of a unitary U, which is precisely what we wanted to implement the order finding algorithm. The inverse Fourier transform performs

$$\frac{1}{2^{t/2}}\sum_{j=0}^{2^{t-1}} e^{2\pi i\varphi j}|j\rangle|u\rangle \mapsto |\tilde{\varphi}\rangle|u\rangle.$$

Theorem 3.4 (Accuracy of φ [3], pp. 223-224)**.** *To successfully obtain φ accurate to n bits with probability of success at least $1-\epsilon$, we choose*

$$t = n + \left\lceil \log\left(2 + \frac{1}{2\epsilon}\right)\right\rceil. \tag{3.12}$$

[8]To do so, we just have to invert the circuit for QFT on Figure 3.1.

3.5 Continued fraction expansion

Let us recapitulate what we have so far: we have reduced the factorisation problem to the order finding problem, which can be solved by calculating eigenvalues of a unitary U of the form $e^{2\pi is/r}$. The quantum phase estimation procedure uses the QFT to estimate the phase φ of an eigenvalue $e^{2\pi i\varphi}$. So we have an estimation

$$\tilde{\varphi} \approx \frac{s}{r}.$$

It misses one building block in order to solve our problem: how to retrieve r from $\tilde{\varphi}$. To do so, we shall use *continued fraction expansions.* The results on this topic can be deepened in [8].

Definition 3.2. *A finite simple continued fraction is an expression of the form*

$$a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{\ddots + \cfrac{1}{a_N}}}} \tag{3.13}$$

which is denoted by $[a_0, a_1, a_2, \cdots, a_N]$*, with* $a_i \in \mathbb{N}^* \ \forall i \in [\![1, N]\!]$*.*
We define the n-th convergent to this continued fraction as $[a_0, \ldots, a_n]$ *for* $0 \leq n \leq N$*.*

Theorem 3.5. *The n-th convergent may be written as a fraction*

$$[a_0, \ldots, a_n] = \frac{p_n}{q_n} \tag{3.14}$$

whose coefficients are given by the following recurrence relation:

$$\begin{array}{lll} p_0 := a_0 & p_1 := a_1 a_0 + 1 & p_n := a_n p_{n-1} + p_{n-2} \\ q_0 := 1 & q_1 := a & q_n := a_n q_{n-1} + q_{n-2} \end{array} \quad (2 \leq n \leq N).$$

Proof. See [8], pp. 166-167. □

The next two theorems will present the *continued fractions algorithm* to approximate $\tilde{\varphi}$ by a n-th convergent and explain why it suffices to solve our problem.

Theorem 3.6 (Continued fraction algorithm)**.** *Any rational number x can be represented by a finite simple continued fraction* $[a_0, \ldots, a_N]$*.*

Proof. See [8], pp. 173-174. □

The algorithm runs as follows.

Algorithm 2: Continued fraction algorithm for rational x.

```
1 Set $a_0 = \lfloor x \rfloor$. Then $x = a_0 + \xi_0$, with $\xi_0 \in [0, 1[$.
2 While $\xi_i \neq 0$: $a_{i+1} = \lfloor 1/\xi_i \rfloor$ and $1/\xi_i = a_{i+1} + \xi_{i+1}$, with $\xi_{i+1} \in [0, 1[$.
3 The continued fraction is $[a_0, \ldots, a_N]$, where $N$ is such that $\xi_N = 0$.
```

Theorem 3.7. *Suppose s/r is a rational number such that*

$$\left|\frac{s}{r} - \varphi\right| \le \frac{1}{2r^2}. \tag{3.15}$$

Then s/r is a convergent of the continued fraction for φ.

Proof. See [8], pp. 196-197. □

Since $\tilde{\varphi}$ is an approximation of s/r accurate to $n = \lceil \log_2 N \rceil$ qubits, the theorem applies. Summarising, given $\tilde{\varphi}$, we can use the continued fraction algorithm to find an irreducible fraction $s'/r' = s/r$. Then, r' is our candidate for the order. We check calculating $x^{r'} \mod N$; if the result is 1, we are done!

4 Oracle circuit

Now that we know the quantum algorithm to factorise a number, we can devote our attention to the implementation in terms of quantum circuit. Although we have already provided a circuit for the QFT, the oracle remains a mysterious black box. In this Section, we show how it can be implemented. In [9], the author presents a circuit implementation that aims to reduce the number of qubits needed, while using a polynomial number of elementary quantum gates.

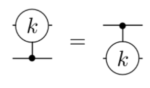
4.1 Adder gate

An initial version for the adder circuit is presented by [10]. Instead of basing on classical circuits which use at least $3n$ qubits to sum two n-qubits number, by composing carry gates and sum gates, the author presents an implementation based on the QFT. The idea, in order to sum a and b, is to compute the Quantum Fourier transform of a, $\phi(a)$, and then use b to compute $\phi(a+b)$. The inverse QFT is then applied to recover the desired result.

Using the following notation[9] to conditional rotation gates R_k, it is possible to describe the adder circuit as in Figure 4.1.

Conditional Rotation

$R_k =$



$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e(\frac{1}{2^k}) \end{bmatrix}$$

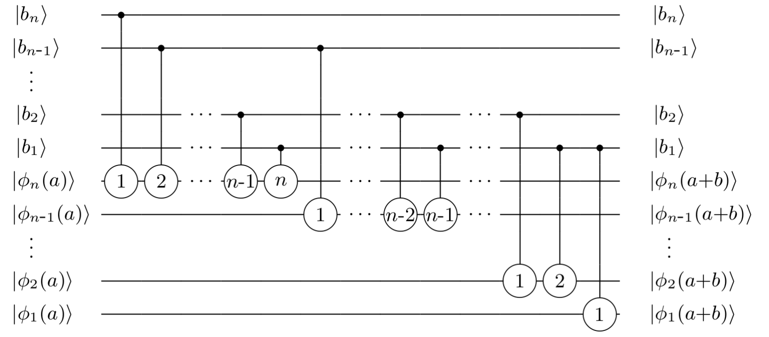


Figure 4.1: Quantum adder based on QFT, as presented in [10]. The QFT operator is denoted $\phi(\cdot)$.

The advantage of this method is that a quantum computer that can perform simultaneous calculations can decrease the time of execution, for instance, by executing all depth 1 rotations simultaneously, and then all depth 2 rotations and so on. In particular, in the case of using the Approximate Quantum Fourier

[9]Here, understand $e\left(\frac{1}{2^k}\right)$ as $\exp\left(\frac{2\pi i}{2^k}\right)$.

Transform (AQFT)[10], this quantum addition can be computed in $n \log_2 n$ operations.

To follow what happens in this circuit, we shall resume the notation introduced in the presentation of the QFT (§3).

$$\begin{aligned}
|\phi_n(a)\rangle &\to \frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i(0.a_n a_{n-1}\cdots a_1 + 0.b_n)}|1\rangle\right) && (R_1 \text{ rotation from } b_n)\\
&\to \frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i(0.a_n a_{n-1}\cdots a_1 + 0.b_n b_{n-1})}|1\rangle\right) && (R_2 \text{ rotation from } b_{n-1})\\
&\vdots\\
&\to \frac{1}{\sqrt{2}}\left(|0\rangle + e^{2\pi i(0.a_n a_{n-1}\cdots a_1 + 0.b_n b_{n-1}\cdots b_1)}|1\rangle\right) && (R_n \text{ rotation from } b_1)\\
&= |\phi(a+b)\rangle.
\end{aligned}$$

A slight variation of the implementation presented so far is introduced by [9], with the notation ϕ-ADD(a). As he points out, the objective of implementing this sum is to afterwards be able to retrieve the order modulo N of a (which plays the role of x in the previous notation). Thus, a is classical and the qubits that represent it may be replaced by classical bits. An additional qubit is added to prevent overflow.

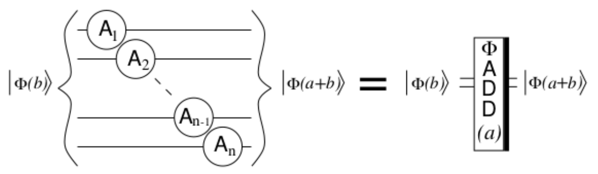


Figure 4.2: Circuit of ϕ-adder as presented in [9].

The reverse of this gate (denoted with the thick bar on the left) will be used to subtraction and comparison.

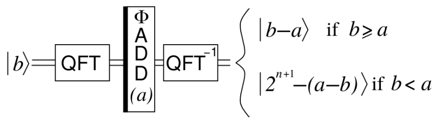


Figure 4.3: Circuit of reverse ϕ-ADD(a) [9].

4.2 Modular adder gate

The goal now is to use the ϕ-ADD(a) to build a adder modulo N. This means that we have to subtract N from the result if $a + b \geq N$. We will denote this block ϕ-ADD(a)-mod-N. The whole block is controlled by two qubits.

[10]This technique consists in reducing the rotations blocks as k gets large, as for large values of k, the rotation block approaches an identity gate ($R_k \approx I$). The optimal value of k is around $\log_2 n$ [10].

The circuit operation, depicted in Figure 4.4 is described as follows:

- The circuit begins with $|\phi(b)\rangle$ as input.
- After the ϕ-ADD(a) block, the value in the register is $\phi(a+b)$ (with no overflow, assuming there is an extra $|0\rangle$ qubit in $|b\rangle$).
- Then, we apply a reverse ϕ-ADDN, having $\phi(a+b-N)$ on the register. We apply this operation even without knowing if it was really necessary to subtract N from the previous result.
- Now we have to check if this operation was indeed necessary or not. But to access the most significant bit of it, we have to pass the whole value through an inverse QFT gate. The most significant bit is used to control a CNOT gate on the ancilla qubit $|0\rangle$.
- The QFT is reapplied, so we come back to $\phi(a+b-N)$.
- The value N is added back if the subtraction was unnecessary, so that $a+b<N$. The value in the register is now $\phi(a+b) \mod N$.
- Close to being satisfied, we just have to restore the ancilla to assure that our gate corresponds to a reversible operation. The trick is to note that $(a+b) \mod N \Leftrightarrow a+b<N$. So one can apply a circuit analogous to the previous one: subtract a, recover the most significant qubit of the result (which will be $|0\rangle$), invert it and use it on controlled-not over the ancilla, which will be restored to $|0\rangle$; at the end, restore a.

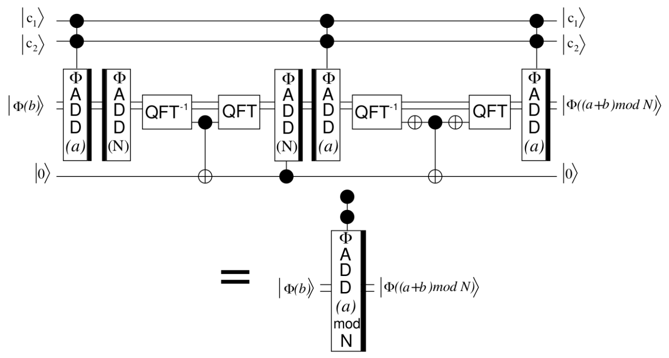


Figure 4.4: Circuit of ϕ-adder modulo N as presented in [9].

4.3 Controlled multiplier gate

Now, we can use the previous ϕ-ADD(a)-mod-N to build a controlled multiplied gate, denoted CMULT(a)-mod-N. This gate has input $|c\rangle|x\rangle|b\rangle$ and is to compute $|c\rangle|x\rangle|(b+ax) \mod N\rangle$ if $|c\rangle = |1\rangle$, or not to alter any qubit otherwise.

The implementation with ϕ-ADD(a)-mod-N is based on the fact that

$$(ax) \mod N = \sum_{i=0}^{n-1} 2^i a x_i \mod N \tag{4.1}$$

where the modulo N has to be applied on each step of the summation. Thus all we need are n successive ϕ-ADD($2^i a$)-mod-N gates.

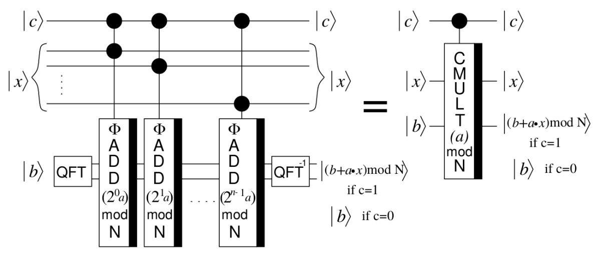


Figure 4.5: Circuit of CMULT(a)-mod-N as presented in [9].

4.4 Controlled U gate

In the previous step, we got a controlled gate that implements the mapping $|x\rangle|b\rangle \mapsto |x\rangle|b + (ax) \mod N\rangle$. However, in fact, we would like to implement $|x\rangle \mapsto |x\rangle|(ax) \mod N\rangle$, which corresponds to our U-gate unitary operation. The implementation is presented in Figure 4.6.

Let us follow the steps of the circuit:

- Start by applying the CMULT(a)-mod-N gate to the input $|c\rangle|x\rangle|0\rangle$.
- Apply a controlled-SWAP gate (controlled by $|c\rangle$ as well). Indeed, it is only necessary to apply it on n qubits, because the most significant one of $(ax) \mod N$ will always be $|0\rangle$ (to store the overflow of the ϕ-adder).
- Then, apply the inverse CMULT(a^{-1})-mod-N[11].

The sequence of performed operations is

$$|x\rangle|0\rangle \to |x\rangle|(ax) \mod N\rangle \to |(ax) \mod N\rangle|x\rangle \to$$
$$|(ax) \mod N\rangle|x - a^{-1}ax\rangle = |(ax) \mod N\rangle|0\rangle.$$

As a result, the controlled-U gate implements $|x\rangle|0\rangle \mapsto |(ax) \mod N\rangle|0\rangle$. To compute a $(\text{control} - U_a)^n$, it is enough to apply $\text{control} - U_{a^n}$ instead of applying the same gate multiple times.

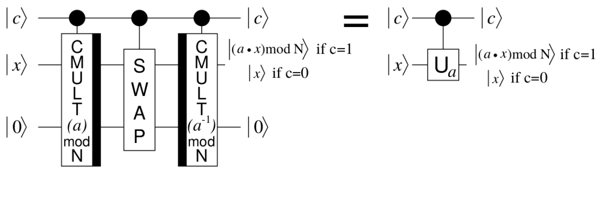


Figure 4.6: Circuit of controlled U-gate as presented in [9].

[11]The inverse modulo N of a, a^{-1}, can be computed via Euclid's algorithm.

5 Results and discussion

The implementations were made in Python using *ProjectQ*, which is an open-source software framework for quantum computing [11, 12]. Our codes are available on GitHub [13] and presented in Appendix A.

The following subsections present results and discussions on that implementation.

5.1 Factorisation

The quantum circuit for factorisation implements Shor's algorithm as described in §3 using the oracle as described in §4. ProjectQ has a feature that generates circuits as LaTeX images. Although the complete circuit and more complex blocks easily become too big to be presented here, some simpler blocks are showed in Figures 5.1, 5.2, 5.3.

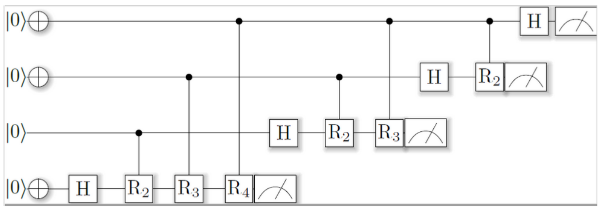


Figure 5.1: Circuit example for the QFT with 4 qubits.

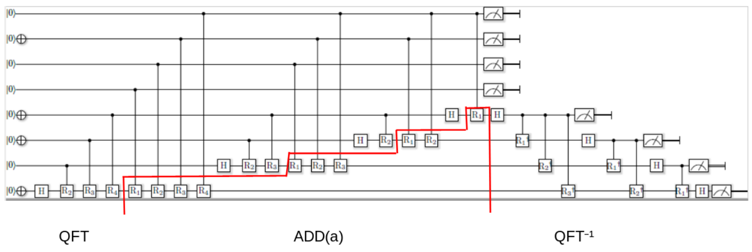


Figure 5.2: Circuit for the adder block, with possible overflow.

TO COMPLETE

5.2 Complexity Analysis

A complexity analysis is performed by [9], presenting number of qubits, order of number of gates and order of depth for each part of the circuit proposed by them (cf. §4) to factorise an integer N stored with $n = \lceil \log_2 N \rceil$ bits. The results are summarised in Table 5.1.

We have compared the number of gates of some of the circuits we have implemented with the result proposed above. Using MATLAB, we fitted each set of data to a polynomial function of the respective degree (Table 5.2, Figures 5.4, 5.5, 5.6). Note that, having used used the exact QFT approach in our

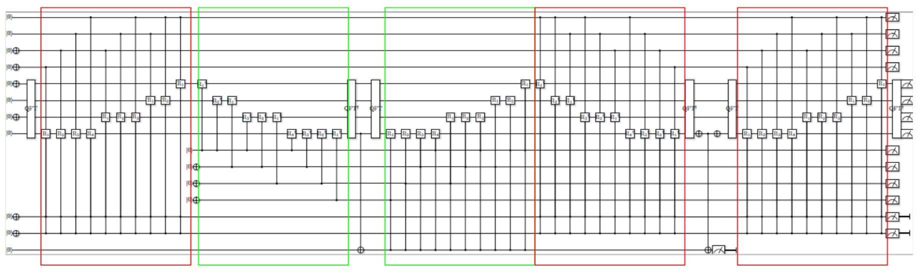

add *a*
subtract *N*
add *N* if control is 1 (*a*+*b*-*N* < 0)
subtract *a* to set ancilla to 0
add *a* to get *a*+*b* mod *N*

Figure 5.3: Circuit for modular adder block.

Table 5.1: Complexity analysis of circuit parts [9].

Circuit	Number of qubits	Order of gates	Order of depth
ϕ-ADD(a)	$n+1$	$O(n)$	$O(1)$
ϕ-ADD(a)-mod-N	$n+4$	$O(nk_{\max})$	$O(n)$
CMULT(a)-mod-N	$2n+3$	$O(n^2k_{\max})$	$O(n^2)$
U_a gate	$2n+3$	$O(n^2k_{\max})$	$O(n^2)$
Shor	$2n+3$	$O(n^3k_{\max})$	$O(n^3)$

When using the exact QFT, $k_{\max} = n$.

implementation, $k_{\max} = n$[12]. We have performed tests for $N = 2^k - 1, \quad k \in [\![3, 19]\!]$ and we counted the swap gates X used for qubits initialisation in the total number of gates.

Table 5.2: Order of number of gates and fitting polynomials in n.

Circuit	Order	Fitting polynomial
CMULT(a)-mod-N	$O(n^3)$	$2.5\ n^3 + 7.5\ n^2 + 18\ n + 11$
U_a gate	$O(n^3)$	$5\ n^3 + 15\ n^2 + 32\ n + 12$
Shor	$O(n^4)$	$10\ n^4 + 26\ n^2 - 16\ n + 2$

We observe that all resulting polynomials fit well the sets of data, matching the theoretical results in Table 5.1.

5.3 Discussion

TO DO

[12]This relation can be reduced with the approximate QFT technique up to $k_{\max} = O(\log(n/\epsilon))$, for any ϵ polynomial in $1/n$ [9].

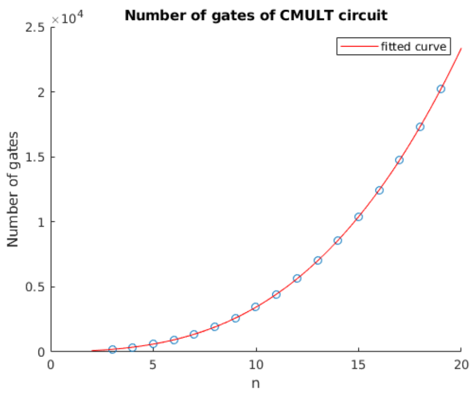


Figure 5.4: Number of gates of CMULT(a)-mod-N circuit as function of n.

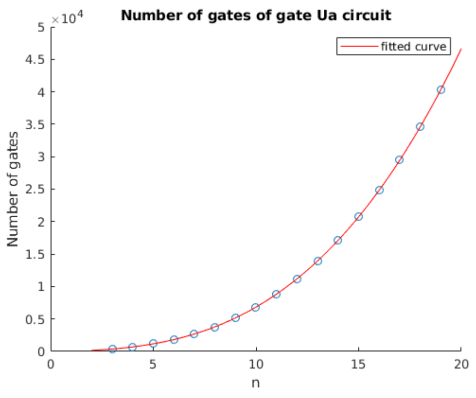


Figure 5.5: Number of gates of U_a gate circuit as function of n.

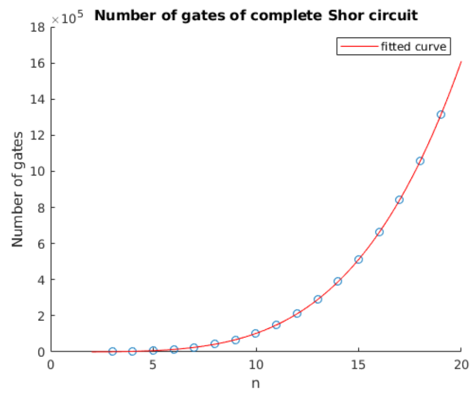


Figure 5.6: Number of gates of Shor's algorithm circuit as function of n.

6 Probability superpositions

After successfully implementing Shor's algorithm, we devoted our attention to the problem of creating quantum superpositions that correspond to efficiently integrable probability distributions, as presented in the paper *Creating superpositions that correspond to efficiently integrable probability distributions* [14].

The problem is as following: given a certain probability distribution $\{p_i\}$, to efficiently create a quantum superposition in the form

$$|\psi(\{p_i\})\rangle = \sum_i \sqrt{p_i}|i\rangle, \tag{6.1}$$

where the $|i\rangle$ form an orthonormal set of states. The most interesting case is when the index i ranges over an exponentially large set.

As far as we know, the problem of when such a state 6.1 can be created remains open. In [14], the authors show that if there is an efficient algorithm to integrate the probability distribution $p(x)$, then it is possible to efficiently create a superposition in the desired form for the discretised version $\{p_i\}$ of $p(x)$. Indeed, the class of log-concave distributions[13] can be efficiently integrated by Monte Carlo methods.

This method may have applications in non-uniform priors for quantum searching, sampling of non-log-concave distributions and estimating the magnitude of a Fourier component.

Iterative method. Let us consider a single variable probability distribution discretised over $N = 2^n$ points. The authors of [6] present an iterative method to create the desired superposition. Suppose the distribution is discretised over 2^m regions ($m < n$) and we already have the state

$$|\psi_m\rangle = \sum_{i=0}^{2^m-1} \sqrt{p_i^{(m)}}|i\rangle, \tag{6.2}$$

where $p_i^{(m)}$ is the probability that the random variable x lies in the i-th region of the discretisation.

Now, we want to get a discretisation in 2^{m+1} regions of $p(x)$. To do so, each region will be split in two by adding one qubit to the previous state. We want to achieve the evolution

$$\sqrt{p_i^{(m)}}|i\rangle \to \sqrt{\alpha_i}|i\rangle|0\rangle + \sqrt{\beta_i}|i\rangle|1\rangle, \tag{6.3}$$

where α_i and β_i are the probabilities of x to lie, respectively, in the left and the right half region of i. Then, the new state shall be

$$|\psi_{m+1}\rangle = \sum_{i=0}^{2^{m+1}-1} \sqrt{p_i^{(m+1)}}|i\rangle. \tag{6.4}$$

The idea is to repeat this process until $m = n$, when we will have created the desired superposition.

This method resembles the creation of a binary tree where the i-th node on the m-th level indicates the probability that the random variable x lies in the i-th of 2^m regions, as shall be discussed later (Figure 6).

[13] A probability distribution $p(x)$ is log-concave if $\frac{\partial^2}{\partial x^2} \log p(x) < 0$.

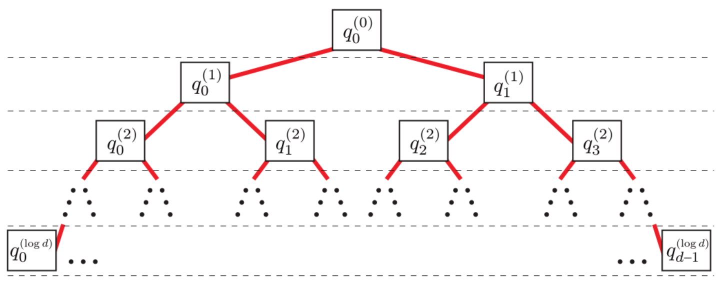


Figure 6.1: Binary tree of region probabilities. The discretised probability for the i-th region of m-th level is noted $q_i^{(m)}$. Here, $d = N = 2^n$.

Creating the evolution (6.3). Let us define x_L^i and x_R^i as the left and right boundaries of region i, respectively. Define also $x_M^i := (x_R^i - x_L^i)/2$ and the function

$$f(i) := \frac{\int_{x_L^i}^{x_M^i} p(x)dx}{\int_{x_L^i}^{x_R^i} p(x)dx}, \tag{6.5}$$

which represents the probability that x lies in the left half of region i.

The critical step of the method, where the challenge resides, is to construct a circuit that efficiently performs the computation

$$\sqrt{p_i^{(m)}}|i\rangle|0\cdots 0\rangle \to \sqrt{p_i^{(m)}}|i\rangle|\theta_i\rangle, \tag{6.6}$$

where $\theta_i := \sqrt{f(i)}$. Then, a controlled rotation of θ_i on the $m+1$-th qubit allows us to obtain

$$\sqrt{p_i^{(m)}}|i\rangle|\theta_i\rangle|0\rangle \to \sqrt{p_i^{(m)}}|i\rangle|\theta_i\rangle(\cos\theta_i|0\rangle + \sin\theta_i|1\rangle), \tag{6.7}$$

which can be identified with Equation (6.3), up to uncomputing $|\theta_i\rangle$.

6.1 Proposed implementation

In the following we describe an implementation of the aforementioned procedure, based on [15]. Consider Figure 6 and the notation $q_i := p_i$ for the discretised version of the probability distribution $p(x)$, we assume to have already calculated. We then have

$$q_i^{(k-1)} = q_{2i}^{(k)} + q_{2i+1}^{(k)} \tag{6.8}$$

for $k = 1, \ldots, n$ and $i = 0, \ldots, 2^k - 1$. The final value is $q_0^{(0)} = 1$, i.e., the sum of all probabilities. Initially, the state is

$$|\psi_0\rangle = |0\rangle_1|0\rangle_2\cdots|0\rangle_n.$$

In the first iteration, the goal is to prepare the state

$$|\psi_1\rangle = \left(\sqrt{q_0^{(1)}}|0\rangle_0 + \sqrt{q_1^{(1)}}|0\rangle_1\right)|0\rangle_2\cdots|0\rangle_n$$

which is achieved by applying the transformation $R(\theta)|0\rangle_1$, where

$$R(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

is the rotation transform by an angle $\theta_0^{(1)} = \arccos\sqrt{q_0^{(1)}}$.

After that we generalise for the following iterations: for the j-th iteration ($j = 2, \dots, n$), the qubits j and higher are still in state $|0\rangle$. So we apply a rotation to the j-th qubit of an angle

$$\theta_i^{(j)} = \arccos\sqrt{\frac{q_{2i}^{(j)}}{q_i^{(j-1)}}} \tag{6.9}$$

depending on the state $|i\rangle$ if the first $j-1$ qubits.

As we already stated, the critical step is to compute the value of the angle $\theta_i^{(j)}$ to apply the rotation. The authors of [15] propose the following scheme.

The Determine Angle Circuit (DAC) block computes the state equivalent to the angle $|\theta\rangle$ from the information $|q_{2i}^{(k)}\rangle$ and $|q_i^{(k-1)}\rangle$ (as in Equation 6.9), while handling some special cases, namely when $\theta = 0$ or $\theta = \pi/2$ (Figure 6.2).

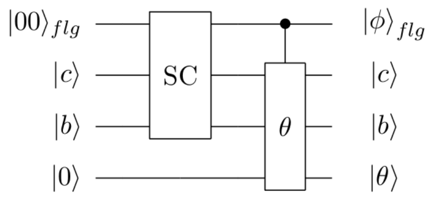


Figure 6.2: DAC circuit [15].

In the figure, $b = q_i^{(k-1)}$ and $c = q_{2i}^{(k)}$. Special cases are treated by SC circuit block according to Table 6.1.

Table 6.1: Special cases treatment.

Flag	Action
00	compute θ
01, 11	$\theta = 0$ ($b = c$)
10	$\theta = \pi/2$ ($c = 0$)

To perform those tests, the block SC is composed by two sub-blocks EQ, which merely tests whether two qubits are the same or not, outputting a flag that indicates the result (Figure 6.3).

Calculating θ. The problem is that, so far, no circuit was proposed to the critical step, which is calculating the angles θ_i (Equation 6.9, sub-block θ in Figure 6.2). This operation is specially difficult because it involves calculating a division, a square root and an arc-cosine.

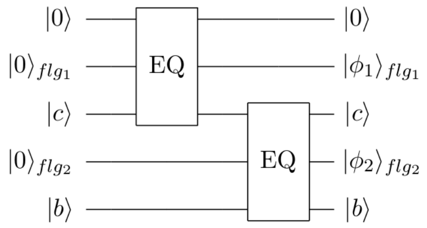


Figure 6.3: SC circuit [15].

The first two operations are addressed in the literature [16, 17]. For example [17] presents a method based on a two-stage Newton iteration. Suppose we want to compute $\sqrt{w}$: the first step computes $\hat{x}_{s1} \approx 1/w$, and then the second computes $\hat{y}_{s2} \approx 1/\hat{x}_{s1} \approx \sqrt{w}$ (Figure 6.4). While it is not our objective to detail such implementations, we invite the reader to further explore the original document. Henceforth we will simply use the fact that there exists a quantum circuit that compute the square root.

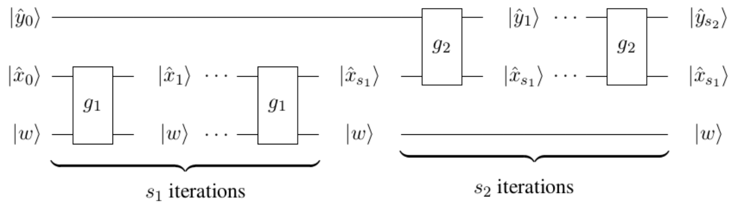


Figure 6.4: Circuit to calculate square root [17].

The most challenging part in the computation of θ (or at least the one that is not present in the literature, as far as we know) is the arc-cosine computation. Thus we propose a method to obtain using the polynomial series of arc-sine function

$$\arcsin(x) = x + \frac{x^3}{6} + O(x^3) \tag{6.10}$$

and a trigonometric relationship between arc-sine and arc-cosine:

$$\arcsin(x) + \arccos(x) = \frac{\pi}{2}. \tag{6.11}$$

The circuit that implements the calculus of arc-sine polynomial approximation up to order 3 is presented in Figure 6.5.

Figure 6.5: Circuit to calculate an approximation of $\arcsin(x)$.

6.2 Discussion

Complexity. An empirical complexity analysis is performed on the approximate arc-sine circuit, considering that this circuit is formed by blocks exponentiation, controlled multiplier, modular adder and QFT (Figure 6.5).

From the analysis in §5.2, we already know that CMULT(a)-mod-N has complexity $O(n^3)$. Modular adder and QFT blocks have at most the same complexity given that they are used to build CMULT(a)-mod-N. On the other hand, exponentiation is built as a loop that applies U_a-gate n times. As U_a-gate has complexity $O(n^3)$, exponential must be $O(n^4)$. Thus, the complete block for computing arc-sine must have complexity of order $O(n^4)$.

We used MATLAB to fit data total number of gates in the circuit as function of n, the number of qubits in the input (Figure 6.6) with $n \in [\![3, 9]\!]$. Fitting data to a fourth-degree polynomial we get

$$4.584\ n^4 + 31.07\ n^3 - 99.99\ n^2 + 605.6\ n - 790.1.$$

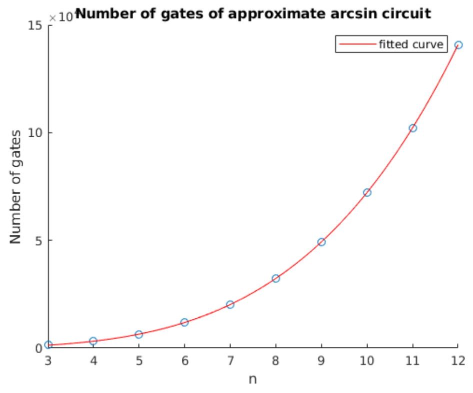


Figure 6.6: Circuit to calculate an approximation of arcsin(x).

Indeed, this is promising news for our implementation idea, because the final circuit may then be polynomial complex, i.e., there may be an efficient implementation for a circuit that solves the initial problem of creating superpositions that correspond to probability densities.

Representation. TO DO

7 Final considerations

TO DO

A Python codes

This section presents the Pyhton codes with ProjectQ library used to implement Shor's algorithm. Each circuit block described in §4 is implemented as a separate file.

```
from projectq.meta import Dagger
from projectq.ops import QFT
from homemade_code.phi_adder import phi_adder


def adder(eng, xa, xb):

    # On passe de a a phi(a) : QTF
    QFT | xa

    phi_adder(eng, xb, xa)

    # On passe de phi(a+b) Ã  a+b QFTˆ-1
    with Dagger(eng):
        QFT | xa
```

Listing 1: adder

```
from projectq.meta import (Control, Dagger)
from projectq.ops import (X, QFT, Measure)

from homemade_code.phi_adder import phi_adder
from homemade_code.inv_phi_adder import inv_phi_adder
from projectq.types import Qureg


def modularAdder(eng, xa: Qureg, x_phi_b: Qureg, xN: Qureg, c1, c2, aux
    ):
    """
    All input are Qubits
    if 2*N>b+a>N
    :param eng:
    :param xa:
    :param x_phi_b: phi(|b>) = phi(|b+a> [N])
    :param xN:
    :param c1: control bit 1
    :param c2: control bit 2
    :param aux: |0> --> |0>
    :return:
    """

    n = xN.__len__()

    # we need to compute a + b and subtract N if a + b >= N.
    with Control(eng, c1):
        with Control(eng, c2):
            phi_adder(eng, xa, x_phi_b)  # we get phi(a+b)
    with Dagger(eng):
        phi_adder(eng, xN, x_phi_b)  # we get phi(a+b-N)
    with Dagger(eng):
        QFT | x_phi_b

    MSB = x_phi_b[n-1]  # we need the most significant bit to evaluate
        a+b-N

    with Control(eng, MSB):
        X | aux

    QFT | x_phi_b

    with Control(eng, aux):
        phi_adder(eng, xN, x_phi_b)  # if a + b < N we add back the
            value N that we subtracted earlier.
    # we now have phi(a+b mod N)

    # these next steps are for restoring aux to 0 using (a + b)mod N >=
         a <=> a + b < N (same logic as before)
    with Control(eng, c1):
        with Control(eng, c2):
            with Dagger(eng):
                phi_adder(eng, xa, x_phi_b)
            #inv_phi_adder(eng, xa, x_phi_b)

    with Dagger(eng):
        QFT | x_phi_b

    MSB2 = x_phi_b[n-1]   # reminder x_phi_b is coded on n+1 bits

    X | MSB2

    with Control(eng, MSB2):
        X | aux

    X | MSB2

    QFT | x_phi_b

    with Control(eng, c1):
        with Control(eng, c2):
            phi_adder(eng, xa, x_phi_b)
```

Listing 2: modularAdder

```
from math import pi
from projectq.meta import Control
from projectq.ops import R


def phi_adder(eng, xa, x_phi_b):  # add a to phi(b) to get phi(a+b)
    n = len(x_phi_b)
    for i in range(n):
        N = n - i - 1
        for k in range(1, N+2):
            with Control(eng, xa[N-k+1]):
                R((2*pi) / (1 << k)) | x_phi_b[N]
```

Listing 3: phi_adder

```
from math import pi
from projectq.meta import (Control, Dagger)
from projectq.ops import R


def inv_phi_adder(eng, xa, x_phi_b): #add a to phi(b) to get phi(a+b)
    n = len(x_phi_b)
    #n = len(xa)
    for i in range(n):
        for k in range(i+1, 0, -1):
            with Control(eng, xa[i-k+1]):
                R(-(2*pi) / (1 << k)) | x_phi_b[i]
```

Listing 4: inv_phi_adder

```
from projectq.meta import Dagger
from projectq.ops import QFT

from homemade_code.modularAdder import modularAdder
from homemade_code.initialisation import initialisation,
    initialisation_n


def cMultModN_non_Dagger(eng, a, xb, xx, xN, aux, xc, N):
    """
    Cannot be Dagger as we allocate qubits on the fly
    But it is require less computation than create all the xa at
        initialisation
    |b> --> |b+(ax) mod N> if xc=1; else |b> -> |b>
    :param eng:
    :param a:
    :param xc: control bit
    :param aux: auxiliary
    :param xx: multiplier
    :param xb: modified qubit
    :param xN: Mod
    :return:
    """
    # b-->phi(b)
    QFT | xb
    n = len(xx) - 1

    for i in range(n):
        xa = initialisation_n(eng, ((2 ** i) * a)%N, n + 1) # both
            input of modularAdder must be <N
        # TODO define xa in a iterative way just by adding a new qubit
            0 as LSB
        modularAdder(eng, xa, xb, xN,  xx[i], xc, aux)

    with Dagger(eng):
        QFT | xb
```

Listing 5: cMultModN_non_Dagger

```
from projectq.meta import (Control, Dagger)
from projectq.ops import (All, Measure, QFT, X, Deallocate)

from homemade_code.modularAdder import modularAdder
from homemade_code.initialisation import initialisation,
    initialisation_n


def inv_cMultModN_non_Dagger(eng, a, xb, xx, xN, aux, xc, N):
    """
    |b> --> |b+(ax) mod N> if xc=1; else |b> -> |b>
    :param eng:
    :param a:
    :param xc: control bit
    :param aux: auxiliary
    :param xx: multiplier
    :param xb: modified qubit
    :param xN: Mod
    :return:
    """
    # b-->phi(b)
    QFT | xb
    n = len(xx) - 1
    for i in range(n-1, -1, -1):
        xa = initialisation_n(eng, ((2 ** i) * a)%N, n + 1)  # both
            input of modularAdder must be <N
        # TODO define xa in a iterative way just by adding a new qubit
            0 as LSB
        with Dagger(eng):
            modularAdder(eng, xa, xb, xN,  xx[i], xc, aux)
    with Dagger(eng):
        QFT | xb
```

Listing 6: inv_cMultModN_non_Dagger

```
from projectq.meta import (Control, Dagger)
from projectq.ops import Swap

from homemade_code.cMultModN_non_Dagger import cMultModN_non_Dagger
from homemade_code.inv_cMultModN_non_Dagger import
    inv_cMultModN_non_Dagger
'''
    ------------------------------------------------------------------------------------------
    '''


def gateUa(eng, a, inv_a, xx, xb, xN, aux, xc, N):
    """

    :param eng:
    :param a: int
    :param inv_a: inverse of a mod N
    :param xx: the modified bits : |x> -> |ax % N> if xc = 1; |x> else
    :param xb: equal to 0 before Ua and after Ua
    :param xN: qubits representing of N
    :param aux: ancillary reset at zero in each gate
    :param xc: control bit
    :param N: int N
    :return:
    """

    cMultModN_non_Dagger(eng, a, xb, xx, xN, aux, xc, N)

    with Control(eng, xc):
        Swap | (xb, xx)  # do work that way

    inv_cMultModN_non_Dagger(eng, inv_a, xb, xx, xN, aux, xc, N)
```

Listing 7: gateUa

```
from projectq.meta import (Control)
from projectq.ops import (H, R)
from math import pi


def qft(eng, xa):
    n = len(xa)
    for i in range(n):
        N = n - i - 1
        H | xa[N]
        for k in range(2, N + 2):
            with Control(eng, xa[N-k+1]):
                R((2*pi) / (1 << k)) | xa[N]
```

Listing 8: qft

```
from __future__ import print_function
from projectq.backends import CircuitDrawer
import math
import random
import sys
from fractions import Fraction
try:
    from math import gcd
except ImportError:
    from fractions import gcd

import projectq.libs.math
import projectq.setups.decompositions
from projectq.backends import Simulator, ResourceCounter
from projectq.cengines import (AutoReplacer, DecompositionRuleSet,
                               InstructionFilter, LocalOptimizer,
                               MainEngine, TagRemover)

from projectq.meta import (Control, Dagger)
from projectq.ops import (All, BasicMathGate, get_inverse, H, Measure,
    R,
                          Swap, X)


'''____________________________________________________________'''

def iqft(eng, xa):
    n = len(xa)
    for i in range(n):
        for k in range(i + 1, 1, -1):
            with Control(eng, xa[i-k+1]):
                R(-(2*math.pi) / (1 << k)) | xa[i]

        H | xa[i]
```

Listing 9: iqft

```
import math
from projectq.ops import X
from numpy import argmax


def int2bit(a):
    if a == 0:
        na = 1
    else:
        n_float = math.log(a, 2)
        na = math.ceil(n_float)
        if math.ceil(n_float) == n_float:
            na += 1
    La = [int(x) for x in bin(a)[2:]]
    La.reverse()
    return [La, na]


def meas2int(L: list):
    L.reverse()
    res = ''
    for i in range(len(L)):
        res += str(L[i])
    L.reverse()
    return int(res, 2)


def initialisation_n(eng, a, n):
    [L, na] = int2bit(a)
    eps = n-na
    for i in range(eps):
        L.append(0)

    # cas ou a > 2**n le dernier bit n'est pas Ã  zÃ©ro
    """
    if L[-1] != 0 and eps <= 0:
        print(a,n, L[-1])
        L[-1] = 0
    """

    xa = eng.allocate_qureg(n)
    for i in range(n):
        if L[i]:
            X | xa[i]
    return xa


def initialisation(eng, args):
    # TODO add flexibility to the inputs here only a list of int is
        accepted with *args
    # Be carefull it returns a fixed length qubits so generate don't
        ancilla with other qubits

    # Initialisation
    m = len(args)
    L = []
    N = []
    Xreg = []
    for i in range(m):
        [Lx, nx] = int2bit(args[i])
        L.append(Lx)
        N.append(nx)
    narg = argmax(N)
    n = N[narg]
    for i in range(m):
        eps = n-N[i]
        for _ in range(eps):
            L[i].append(0)
    for _ in range(m):
        Xreg.append(eng.allocate_qureg(n))

    # initialisation de a, b et N
    for j in range(m):
        for i in range(n):
            if L[j][i]:
                X | Xreg[j][i]

    return Xreg


def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)


def mod_inv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('modular␣inverse␣does␣not␣exist')
    else:
        return x % m
```

Listing 10: initialisation

```
from __future__ import print_function

import math
import random
import sys
from fractions import Fraction
try:
    from math import gcd
except ImportError:
    from fractions import gcd

from builtins import input

import projectq.libs.math
import projectq.setups.decompositions
from projectq.backends import Simulator, ResourceCounter
from projectq.cengines import (AutoReplacer, DecompositionRuleSet,
                               InstructionFilter, LocalOptimizer,
                               MainEngine, TagRemover)
from projectq.libs.math import (AddConstant, AddConstantModN,
                                MultiplyByConstantModN)
from projectq.meta import Control
from projectq.ops import (All, BasicMathGate, get_inverse, H, Measure,
    QFT, R,
                          Swap, X)

from homemade_code.gateUa import gateUa
from homemade_code.initialisation import mod_inv, initialisation_n

def run_shor(eng, N, a, verbose=True):
    """
    Runs the quantum subroutine of Shor's algorithm for factoring.
    The 1 controling qubits version
    [2, 3, 4, 5, 6, 7, 8]
    [1002,2926,6822,13802, 25257, 42548, 67868]
    Args:
        eng (MainEngine): Main compiler engine to use.
        N (int): Number to factor.
        a (int): Relative prime to use as a base for a^x mod N.
        verbose (bool): If True, display intermediate measurement
            results.

    Returns:
        r (float): Potential period of a.
    """
    n = int(math.ceil(math.log(N, 2)))
    x = eng.allocate_qureg(n)
    xN = initialisation_n(eng, N, n)
    xb = initialisation_n(eng, 0, n)
    aux = initialisation_n(eng, 0, 1)

    X | x[0]

    measurements = [0] * (2 * n)  # will hold the 2n measurement
        results

    ctrl_qubit = eng.allocate_qubit()
    # each iteration -> 454*log2(N) + c -> 2*n(454*log2(N) +c ) -> 908
        * log2(N)^2 -> O(n^2)
    # c = 3 ou 4
    # last interation measurment -> 3*n
    # donc 908*(n**2) + 3*n
    for k in range(2 * n):

        current_a = pow(a, 1 << (2 * n - 1 - k), N)
        # one iteration of 1-qubit QPE
        H | ctrl_qubit
        """
        with Control(eng, ctrl_qubit):
            MultiplyByConstantModN(current_a, N) | x
        """
        gateUa(eng, current_a, mod_inv(current_a, N), x, xb, xN, aux,
            ctrl_qubit, N)
        # nb of gate linear in log2(N) approx ~ 454*log2(N)
        # perform inverse QFT --> Rotations conditioned on previous
            outcomes
        """
        for i in range(k):
            if measurements[i]:
                R(-math.pi/(1 << (k - i))) | ctrl_qubit
        """
        H | ctrl_qubit

        # and measure
        Measure | ctrl_qubit
        eng.flush()
        measurements[k] = int(ctrl_qubit)
        if measurements[k]:
            X | ctrl_qubit

        if verbose:
            print("\033[95m{}\033[0m".format(measurements[k]), end="")
            sys.stdout.flush()
    Measure | aux
    All(Measure) | xN
    All(Measure) | xb
    All(Measure) | x
    # turn the measured values into a number in [0,1)
    y = sum([(measurements[2 * n - 1 - i]*1. / (1 << (i + 1)))
             for i in range(2 * n)])

    # continued fraction expansion to get denominator (the period?)
    r = Fraction(y).limit_denominator(N-1).denominator

    # return the (potential) period
    return r
```

```
# Filter function, which defines the gate set for the first
    optimization
# (don't decompose QFTs and iQFTs to make cancellation easier)
def high_level_gates(eng, cmd):
    g = cmd.gate
    if g == QFT or get_inverse(g) == QFT or g == Swap:
        return True
    if isinstance(g, BasicMathGate):
        return False
        if isinstance(g, AddConstant):
            return True
        elif isinstance(g, AddConstantModN):
            return True
        return False
    return eng.next_engine.is_available(cmd)


#if __name__ == "__main__":
def run():
    # build compilation engine list
    resource_counter = ResourceCounter()
    rule_set = DecompositionRuleSet(modules=[projectq.libs.math,
                                             projectq.setups.
                                                 decompositions])
    compilerengines = [AutoReplacer(rule_set),
                       InstructionFilter(high_level_gates),
                       TagRemover(),
                       LocalOptimizer(3),
                       AutoReplacer(rule_set),
                       TagRemover(),
                       LocalOptimizer(3),
                       resource_counter]

    # make the compiler and run the circuit on the simulator backend
    #eng = MainEngine(Simulator(), compilerengines)
    eng = MainEngine(resource_counter)
    # print welcome message and ask the user for the number to factor
    print("\n\t\033[37mprojectq\033[0m\n\t--------\n\tImplementation␣of
        ␣Shor"
          "\'s␣algorithm.", end="")
    N = int(input('\n\tNumber␣to␣factor:␣'))
    print("\n\tFactoring␣N␣=␣{}:␣\033[0m".format(N), end="")

    # choose a base at random:
    a = N
    while not gcd(a, N)==1:
        a = random.randint(2,N)

    print("\na␣is␣" + str(a))
    if not gcd(a, N) == 1:
        print("\n\n\t\033[92mOoops,␣we␣were␣lucky:␣Chose␣non␣relative␣
            prime"
              "␣by␣accident␣:)")
        print("\tFactor:␣{}\033[0m".format(gcd(a, N)))
    else:
        # run the quantum subroutine
        r = run_shor(eng, N, a, True)
        print("\n\nr␣found␣:␣" + str(r))
        # try to determine the factors

        if r % 2 != 0:
            r *= 2
        apowrhalf = pow(a, r >> 1, N)
        f1 = gcd(apowrhalf + 1, N)
        f2 = gcd(apowrhalf - 1, N)
        print("f1␣=␣{},␣f2␣=␣{}".format(f1, f2))
        if ((not f1 * f2 == N) and f1 * f2 > 1 and
                int(1. * N / (f1 * f2)) * f1 * f2 == N):
            f1, f2 = f1*f2, int(N/(f1*f2))
        if f1 * f2 == N and f1 > 1 and f2 > 1:
            print("\n\n\t\033[92mFactors␣found␣:-)␣:␣{}␣*␣{}␣=␣{}\033[0
                m"
                  .format(f1, f2, N))
        else:
            print("\n\n\t\033[91mBad␣luck:␣Found␣{}␣and␣{}\033[0m".
                format(f1,
                                                                          f2
                                                                              )
                                                                              )

        return resource_counter  # print resource usage
```

Listing 11: shor

```
from __future__ import print_function

import math
import random
import sys
from fractions import Fraction
try:
    from math import gcd
except ImportError:
    from fractions import gcd

from builtins import input

import projectq.libs.math
import projectq.setups.decompositions
from projectq.backends import Simulator, ResourceCounter
from projectq.cengines import (AutoReplacer, DecompositionRuleSet,
                               InstructionFilter, LocalOptimizer,
                               MainEngine, TagRemover)
from projectq.libs.math import (AddConstant, AddConstantModN,
                                MultiplyByConstantModN)
from projectq.meta import Control, Dagger
from projectq.ops import (All, BasicMathGate, get_inverse, H, Measure,
    QFT, R,
                          Swap, X)

from homemade_code.gateUa import gateUa
from homemade_code.initialisation import mod_inv, initialisation_n

def run_shor(eng, N, a, verbose=False):
    """
    Runs the quantum subroutine of Shor's algorithm for factoring. with
         2n control qubits

    Args:
        eng (MainEngine): Main compiler engine to use.
        N (int): Number to factor.
        a (int): Relative prime to use as a base for a^x mod N.
        verbose (bool): If True, display intermediate measurement
            results.

    Returns:
        r (float): Potential period of a.
    """
    n = int(math.ceil(math.log(N, 2)))

    x = eng.allocate_qureg(n)
    xN = initialisation_n(eng, N, n)
    xb = initialisation_n(eng, 0, n)
    aux = initialisation_n(eng, 0, 1)
    X | x[0]  # set x to 1

    measurements = [0] * (2 * n)  # will hold the 2n measurement
        results

    ctrl_qubit = eng.allocate_qureg(2*n)

    for k in range(2 * n):
        current_a = pow(a, 1 << k, N)
        # one iteration of 1-qubit QPE
        H | ctrl_qubit[k]
        gateUa(eng, current_a, mod_inv(current_a, N), x, xb, xN, aux,
            ctrl_qubit[k], N)

    with Dagger(eng):
        QFT | ctrl_qubit

    # and measure
    All(Measure) | ctrl_qubit
    eng.flush()
    for k in range(2*n):
        measurements[k] = int(ctrl_qubit[k])

    All(Measure) | x
    # turn the measured values into a number in [0,1)
    y = sum([(measurements[i]*1. / (1 << (i + 1)))
             for i in range(2 * n)])

    # continued fraction expansion to get denominator (the period?)
    r = Fraction(y).limit_denominator(N-1).denominator

    # return the (potential) period
    return r


if __name__ == "__main__":
    # build compilation engine list
    resource_counter = ResourceCounter()
    rule_set = DecompositionRuleSet(modules=[projectq.libs.math,
                                             projectq.setups.
                                                 decompositions])
    compilerengines = [AutoReplacer(rule_set),
                       TagRemover(),
                       LocalOptimizer(3),
                       AutoReplacer(rule_set),
                       TagRemover(),
                       LocalOptimizer(3),
                       resource_counter]

    # make the compiler and run the circuit on the simulator backend
    eng = MainEngine(Simulator(), compilerengines)

    # print welcome message and ask the user for the number to factor
    print("\n\t\033[37mprojectq\033[0m\n\t--------\n\tImplementation␣of
        ␣Shor"
          "\'s␣algorithm.", end="")
    N = int(input('\n\tNumber␣to␣factor:␣'))
    print("\n\tFactoring␣N␣=␣{}:␣\033[0m".format(N), end="")

    # choose a base at random:
    a = int(random.random()*N)
    print("\na␣is␣" + str(a))
    if not gcd(a, N) == 1:
        print("\n\n\t\033[92mOoops,␣we␣were␣lucky:␣Chose␣non␣relative␣
            prime"
              "␣by␣accident␣:)")
        print("\tFactor:␣{}\033[0m".format(gcd(a, N)))
    else:
        # run the quantum subroutine
        r = run_shor(eng, N, a, True)
        print(r)
        # try to determine the factors
        if r % 2 != 0:
            r *= 2
        apowrhalf = pow(a, r >> 1, N)
        f1 = gcd(apowrhalf + 1, N)
        f2 = gcd(apowrhalf - 1, N)
        if ((not f1 * f2 == N) and f1 * f2 > 1 and
                int(1. * N / (f1 * f2)) * f1 * f2 == N):
            f1, f2 = f1*f2, int(N/(f1*f2))
        if f1 * f2 == N and f1 > 1 and f2 > 1:
            print("\n\n\t\033[92mFactors␣found␣:-)␣:␣{}␣*␣{}␣=␣{}\033[0
                m"
                  .format(f1, f2, N))
        else:
            print("\n\n\t\033[91mBad␣luck:␣Found␣{}␣and␣{}\033[0m".
                format(f1,
                                                                        f2
                                                                            )
                                                                            )



        print(resource_counter)  # print resource usage
```

Listing 12: shor3n.

References

[1] Valiron, B. "Quantum computation: a tutorial". *New Generation Computing*, vol. 30, no. 4, pp. 271-296, oct. 2012.

[2] Portugal, R. et al. *Uma introdução à computação quântica.* São Carlos: SBMAC, 2004.

[3] Nielsen, Michael A. and Chuang, Isaac L. *Quantum computation and quantum information.* 10th anniversary edition. Cambridge: Cambridge University Press, 2010.

[4] Shor, Peter. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer". *SIAM Journal on Computing*, vol. 26, no. 5, pp. 1484-1509, 1997.

[5] Ekert, A. and Jozsa, R. "Quantum computation and Shor's factoring algorithm". *Reviews of Modern Physics*, vol. 68, no. 3, pp. 733-753, 1996.

[6] Griffiths, Robert B. and Niu, Chi-Sheng. "Semiclassical Fourier Transform for Quantum Computation". *Physycal Review Letters*, vol. 76, no. 17, pp. 3228-3231, 1996.

[7] Wikipedia. *Quantum Fourier transform.* Available in: `https://en.wikipedia.org/wiki/Quantum_Fourier_transform`. Access on: 18 dec. 2018.

[8] Hardy, G. H. and E. M. Wright. *An introduction to the theory of numbers.* 6th edition. Oxford : Oxford University Press, 2008.

[9] Beauregard, Stephane. "Circuit for Shor's algorithm using $2n + 3$ qubits". *Quantum Information and Computation*, vol. 3, no. 2 pp. 175-185, 2003.

[10] Draper, Thomas G. *Addition on a quantum computer.* Available in: `https://arxiv.org/abs/quant-ph/0008033`. Access on: 14 dec. 2018.

[11] ProjectQ. Availabe in: `https://projectq.ch/`. Access on: 28 nov. 2018.

[12] Steiger, Damian S.; Häner, Thomas and Troyer, Matthias. "ProjectQ: an open source software framework for quantum computing". *Quantum*, vol. 2, p. 49, jan. 2018.

[13] GitHub. *quantum-calculus.* Available in: `https://github.com/miyamotohk/quantum-calculus`. Access on: 11 jun. 2019.

[14] Grover, L. and Rudolph, T. *Creating superpositions that correspond to efficiently integrable probability distributions.* Available in: `https://arxiv.org/abs/quant-ph/0208112`. Access on: 9 may 2019.

[15] Chiang, Chen-Fu; Nagaj, Daniel; Wocjan, Pawel. "Efficient Circuits for Quantum Walks". *Quantum Information & Computation*, vol. 10, no. 5, pp. 420-434, 2010.

[16] Gokhale, Pranav. *Implementation of Square Root Function Using Quantum Circuits*. Princeton University, 2014. Available in: `http://www.undergraduatelibrary.org/2014/computer-sciences/implementation-square-root-function-using-quantum-circuits`. Access on: 19 jun. 2019.

[17] Hadfield, Stuart Andrew. *Quantum Algorithms for Scientific Computing and Approximate Optimization*. PhD Thesis, Columbia University, 2018. Available in: `https://arxiv.org/abs/1805.03265`. Access on: 19 jun. 2019.