

# Quantum Computation of the Prime Counting Function

Capstone final thesis  
30 May 2018

---

Name:	Peter-Jan Derks Amsterdam University College
Email:	<a href="mailto:peter-janderks@hotmail.com">peter-janderks@hotmail.com</a>
Student ID:	11065540
Major:	Sciences (Physics)
Expected graduation:	July 2018
Tutor:	Dr. Forrest Bradbury
Word Count:	7892
Supervisor:	Prof.dr. Kareljan Schoutens Institute for Theoretical Physics University of Amsterdam
Email:	<a href="mailto:c.j.m.schoutens@uva.nl">c.j.m.schoutens@uva.nl</a>
Reader:	Dr. Jasper van Wezel Institute for Theoretical Physics University of Amsterdam
Email:	<a href="mailto:j.vanwezel@uva.nl">j.vanwezel@uva.nl</a>

---



## Summary

In 2013, Latorre and Sierra proposed a quantum algorithm to compute the prime counting function with an error smaller than the error in the Riemann Hypothesis. In this thesis the proposed quantum circuit is partially designed and implemented to prepare the state of a  $n$  qubit quantum register in the superposition of all prime numbers less than  $2^n$ . This circuit uses Grover's algorithm with a quantum primality oracle, which is a quantum implementation of the Miller-Rabin primality test. It is shown that this circuit can be used by the phase estimation algorithm to approximate the prime counting function.

Keywords: *Prime counting function, Quantum programming, Quantum Fourier transform, Grover's Search algorithm, Number Theory*

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Nomenclature</b>	<b>4</b>
<b>3</b>	<b>Grover's Search algorithm</b>	<b>7</b>
3.1	Grover's algorithm . . . . .	7
3.1.1	The Oracle . . . . .	7
3.1.2	The Algorithm . . . . .	8
3.1.3	Performance of one Grover iteration . . . . .	9
3.1.4	Worked example . . . . .	11
3.1.5	Multiple Grover iterations . . . . .	11
<b>4</b>	<b>Grover primality oracle</b>	<b>14</b>
4.1	Miller-Rabin primality test . . . . .	14
4.2	ProjectQ . . . . .	20
4.2.1	Custom gates in project Q . . . . .	20
4.2.2	Five qubit result . . . . .	24
<b>5</b>	<b>Quantum Fourier transform and phase estimation</b>	<b>26</b>
5.1	Quantum Fourier transform . . . . .	26
5.1.1	Quantum circuit . . . . .	28
5.1.2	Complexity . . . . .	30
5.2	Phase estimation . . . . .	30
5.2.1	The algorithm . . . . .	31
<b>6</b>	<b>Quantum Counting</b>	<b>33</b>
6.1	Matrix representation of the Grover iteration . . . . .	33
6.2	Circuit in Quirk . . . . .	34
<b>7</b>	<b>Conclusion</b>	<b>36</b>
<b>8</b>	<b>Appendix A</b>	<b>38</b>
8.1	5 Qubit Prime State code . . . . .	38
8.2	Modular Increment and Decrement Gates . . . . .	50
8.3	Probability and Amplitude Functions . . . . .	52

# Chapter 1

## Introduction

In Number Theory, the prime counting function is defined as the number of primes less than or equal to  $x$ , denoted  $\pi(x)$ . In 2013, Latorre and Sierra proposed a quantum algorithm to efficiently compute  $\pi(x)$ . To calculate  $\pi(x)$  a quantum register needs to be prepared in the prime state defined by Latorre and Sierra as,

$$|P_n\rangle \equiv \frac{1}{\sqrt{\pi(2^n)}} \sum_{p \in \text{primes} < 2^n} |p\rangle. \quad (1.1)$$

A quantum register can be prepared in a state approximately equal to the prime state by using Grover's algorithm with a primality oracle. Generally, the quantum counting algorithm uses the Grover iteration in a phase estimation algorithm to calculate the number of solutions to a search problem. Therefore, the quantum counting algorithm can compute  $\pi x$ . The Riemann hypothesis is one of the six unsolved Millenial problems and often considered the most important problem in number theory. The Riemann hypothesis implies the following

$$|\pi(x) - Li(x)| < O(x^{\frac{1}{2}} \log(x)). \quad (1.2)$$

Where  $Li(x)$  stands for the logarithmic integral function and  $\pi(x)$  is the number of primes up to integer  $x$ . The quantum counting algorithm can calculate the number of primes with the following error [2]

$$|\tilde{\pi}(x) - \pi(x)| < \frac{2\pi}{c} \pi(x)^{\frac{1}{2}} + \frac{\pi^2}{c^2}. \quad (1.3)$$

Using the Prime Number Theorem, that states that  $\pi(x)$  is approximately equal to  $\frac{x}{\log(x)}$  this can be rewritten as

$$|\tilde{\pi}(x) - \pi(x)| < \frac{2\pi}{c} \frac{x^{\frac{1}{2}}}{\log x^{\frac{1}{2}}}. \quad (1.4)$$

Because the right side of this equation is smaller than the right side of (1.2):

$$\frac{x^{\frac{1}{2}}}{\log x^{\frac{1}{2}}} < O(x^{\frac{1}{2}} \log(x)) \quad (1.5)$$

the left side must also be smaller than the left side of (1.2). If this is not true for some value



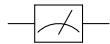


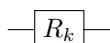



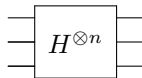
of  $x$  the Riemann hypothesis is falsified.

In this thesis the proposed algorithm to calculate  $\pi(x)$  is partially implemented. In Chapters 3 and 4 Grover's algorithm is explained, followed by an implementation of the algorithm to compute the prime state for  $n=4$  in Quirk and  $n=5$  in ProjectQ. In Chapters 5 and 6 the phase estimation algorithm is explained and followed by an attempted implementation to calculate  $\pi(7)$ . The main source for Chapters 3 and 5 are De Wolf's lecture notes and the book Quantum Computation and Quantum Information written by Nielsen and Chuang [3][7]. Several exercises from Nielsen and Chuang are included in this thesis. Chapter 4 includes code written with ProjectQ's software framework for quantum computing to simulate the computation of the prime state [5][9]. Although existing quantum computers use small amounts of qubits, software like ProjectQ is being developed for two main reasons. Firstly it is useful to simulate small quantum computers for calibration, validation, and benchmarking. Additionally, large quantum algorithms are complicated to calculate by hand, so by building a circuit and moving around gates insight can be gained into how a circuit works. All code and Quirk circuits included in this thesis can be found at <https://github.com/peter-janderks/bachelor-thesis>.

# Chapter 2

## Nomenclature

The algorithms in this thesis are visualized as quantum circuits with the following symbols:

		wire carrying a single qubit (time goes left to right)
		wire carrying n qubits
		projection onto $ 0\rangle$ and $ 1\rangle$ ,
	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	Z gate
	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	X gate
	$\begin{bmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{bmatrix}$	X gate
	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & u_{00} & u_{01} \\ 0 & 0 & u_{10} & u_{11} \end{bmatrix}$	controlled-U
	$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & u_{00} & u_{01} \\ 0 & 0 & u_{10} & u_{11} \end{bmatrix}$	anti-controlled-U
	$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$	Hadamard gate
	$\frac{1}{\sqrt{2}} \sum_{x_i, y_i} (-1)^{x_1 \cdot y_1}  x_1\rangle \langle y_1 $	Hadamard gate applied to n qubits

One of the most used gates in this thesis is the Hadamard gate. By explaining how the Hadamard gate acts on single qubit and multiple qubit states, it should become clear how the mathematical tools in linear algebra and quantum mechanics are used to describe quantum circuits.<sup>1</sup> The Hadamard operator on one qubit can be written as:

$$\begin{aligned} H &= \frac{1}{\sqrt{2}} [ |0\rangle + |1\rangle ] \langle 0| + (|0\rangle - |1\rangle) \langle 1| ] \\ H &= \frac{1}{\sqrt{2}} \sum_{x_i, y_i} (-1)^{x_i \cdot y_i} |x_i\rangle \langle y_i| \end{aligned} \quad (2.1)$$

This is equivalent to the matrix representation:

$$\begin{aligned} H &= \frac{1}{\sqrt{2}} \left[ \left( \begin{bmatrix} 1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right) \begin{bmatrix} 1 & 0 \end{bmatrix} + \left( \begin{bmatrix} 1 \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ 1 \end{bmatrix} \right) \begin{bmatrix} 0 & 1 \end{bmatrix} \right] \\ &= \frac{1}{\sqrt{2}} \left[ \begin{bmatrix} 1 & 0 \\ 1 & 0 \end{bmatrix} + \begin{bmatrix} 0 & 1 \\ 0 & -1 \end{bmatrix} \right] = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \end{aligned} \quad (2.2)$$

The Hadamard transform on  $n$  qubits,  $H^{\otimes n}$ , may be written as:

$$H^{\otimes n} = \bigotimes_{i=1}^n \frac{1}{\sqrt{2}} \left( \sum_{x_i, y_i} (-1)^{x_i \cdot y_i} |x_i\rangle \langle y_i| \right) \quad (2.3)$$

Here  $x_i \cdot y_i$  is multiplication mod 2 and therefore  $x \cdot y = \sum_{i=1}^n x_i \cdot y_i$ . By defining  $\otimes |x_i\rangle = |x\rangle$  and  $\otimes |y_i\rangle = |y\rangle$  we can eliminate the tensor product:

$$H^{\otimes n} = \frac{1}{\sqrt{2^n}} \sum_{x, y} (-1)^{x \cdot y} |x\rangle \langle y| \quad (2.4)$$

This formula is useful because it is quicker to compute the output of a Hadamard gate applied to multiple qubits than by working out the matrix representation. For example the Hadamard gate applied to  $n$  qubits gives

$$H^{\otimes n} |0^{\otimes n}\rangle = \left( \frac{1}{\sqrt{2^n}} \sum_{x, y} (-1)^{x \cdot y} |x\rangle \langle y| \right) |0^{\otimes n}\rangle \quad (2.5)$$

We only have to sum over  $\langle y| = \langle 0^{\otimes n}|$ :

---

<sup>1</sup>In the first three chapters of Quantum Computation and Quantum Information, Nielsen and Chuang introduce the necessary mathematical tools to understand quantum circuits, including matrix multiplications, eigenstates and eigenvalues and tensor products.

$$\begin{aligned}
 H^{\otimes n} |0^{\otimes n}\rangle &= \frac{1}{\sqrt{2^n}} \sum_x (-1)^{x \cdot 0} |x\rangle \langle 0^{\otimes n} | 0^{\otimes n}\rangle \\
 &= \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle
 \end{aligned} \tag{2.6}$$

The matrix representation for  $H^{\otimes 2}$  is:

$$H^{\otimes 2} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \tag{2.7}$$

## Chapter 3

# Grover's Search algorithm

Nielsen and Chang define three main classes of quantum algorithms; quantum search algorithms, quantum simulations and quantum algorithms based on the Fourier transform. In this Chapter Grover's search which is the most important quantum search algorithm is explained. In Chapter 5 the quantum Fourier transform, the main algorithm of the third class, will be explained.

### 3.1 Grover's algorithm

In 1996, Grover described a quantum algorithm which can efficiently solve a NP search problem [4]. The example given by Grover of such a search problem is the following; imagine you are trying to find someone's phone number in a phone directory in which the names are arranged in a completely random order. On a classical computer, in the worst case the person's name whose phone number you are looking for is the last name you check. If the phone book has  $N$  names with corresponding phone numbers it takes  $O(N)$  operations to find someone's phone number. Grover's quantum search algorithm is able to find this phone number with  $O(\sqrt{N})$  operations. This is done by adjusting the phases of qubits, such that several computations reinforce each other while others interfere randomly. The inputs to Grover's search algorithm are an oracle " $O$ " and  $n + 1$  qubits, of which  $n$  qubits are in the state  $|0\rangle$  and 1, called the "oracle qubit" is in the state  $|1\rangle$ .

#### 3.1.1 The Oracle

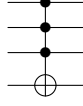
A search problem can be represented by a function which takes input  $x$  and returns 1 if  $x$  is a solution and 0 if  $x$  is not a solution. There are two possible implementations of this function on a quantum computer: a Boolean oracle and a phase-shift oracle. The action of both oracles can be described as:[7]

$$|x\rangle \xrightarrow{O} (-1)^{f(x)} |x\rangle \quad (3.1)$$

The amount of qubits needed for the phase-shift oracle is equal to the size of the state  $|x\rangle$ , while the Boolean oracle uses an extra oracle qubit. The following example illustrates how both oracles work. Take  $f(x) = 0$  for all  $x$  except  $x = |111\rangle$ . The oracles use the following gates,



Boolean oracle:



Phase-shift oracle:



In the Boolean oracle, the oracle qubit  $|q\rangle$  is flipped if  $f(x) = 1$  and unchanged if  $f(x) = 0$ :

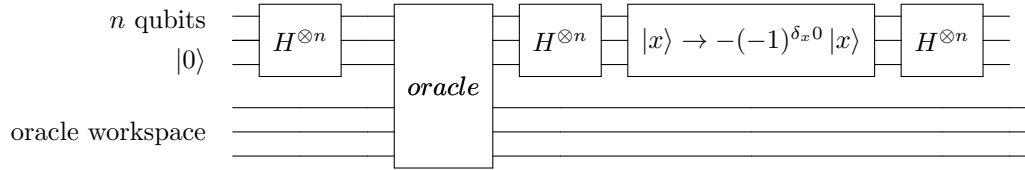
$$|x\rangle |q\rangle \xrightarrow{O} |x\rangle |q \oplus f(x)\rangle. \quad (3.2)$$

It is useful to apply a Hadamard gate on the oracle qubit such that it is in the state  $(|0\rangle - |1\rangle)/\sqrt{2}$  when the oracle is applied. When this is done the action of the Boolean oracle can be written as:

$$|x\rangle \left( \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right) \xrightarrow{O} (-1)^{f(x)} |x\rangle \left( \frac{|0\rangle - |1\rangle}{\sqrt{2}} \right). \quad (3.3)$$

This shows that the oracle qubit does not need to be measured to find out if  $|x\rangle$  is a solution and that both oracles perform the same operation.

### 3.1.2 The Algorithm



The quantum circuit for Grover's iteration in figure 3.1.2 uses two registers, an  $n$  qubit register and an oracle workspace. The  $n$  qubit register is initially in the state

$$|0^{\otimes n}\rangle. \quad (3.4)$$

Before applying the Grover iteration a Hadamard gate is applied to all qubits, such that the first  $n$  qubits are in a superposition of all possible states.

$$|\Psi\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle \quad (3.5)$$

Then the first step of Grover iteration, the search problem oracle, is applied. It performs a phase shift on every state which is a solution to the search problem  $|x\rangle \xrightarrow{O} (-1)^{f(x)} |x\rangle$ . Then a Hadamard transformation, a conditional phase shift and again a Hadamard transformation is applied. The state after each gate could be written out, but for calculations later on in this

section it is useful to write the Grover iteration as one operator. The phase shift operator shifts every state except  $|0\rangle$  with  $-1$ :

$$|x\rangle \rightarrow -(-1)^{\delta_{x0}} |x\rangle \quad (3.6)$$

The matrix of this operation can be decomposed as the unitary operator  $2|0^n\rangle\langle 0^n| - I^{\otimes n}$ :

$$\begin{aligned} & 2 \begin{bmatrix} 1 \\ \vdots \\ 0_n \end{bmatrix} \begin{bmatrix} 1 & \dots & 0_n \end{bmatrix} - \begin{bmatrix} 1 & \dots & 0_n \\ \vdots & \ddots & \vdots \\ 0_n & \dots & 1 \end{bmatrix} \\ &= \begin{bmatrix} 1 & \dots & \dots & 0_n \\ \vdots & -1 & & \vdots \\ \vdots & & \ddots & \vdots \\ 0_n & \dots & \dots & -1 \end{bmatrix} \end{aligned} \quad (3.7)$$

Using this representation on the phase shift operator, the combined effect of the Hadamard transform, phase shift operator and again the Hadamard transform can be written as:

$$\begin{aligned} & H^{\otimes n} (2|0\rangle\langle 0| - I) H^{\otimes n} \\ &= H^{\otimes n} (2|0\rangle\langle 0| H^{\otimes n} - I H^{\otimes n}) \\ &= 2H^{\otimes n} |0\rangle\langle \psi| - H^{\otimes n} I H^{\otimes n} \\ &= 2|\psi\rangle\langle \psi| - I \end{aligned} \quad (3.8)$$

Thus the Grover iteration may be written as

$$\rightarrow \left[ (2|\psi\rangle\langle \psi| - I) O \right] \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle \quad (3.9)$$

### 3.1.3 Performance of one Grover iteration

Define the state  $|\alpha\rangle$  as the sum over all states which are not a solution to the search problem

$$|\alpha\rangle = \sum_{x \in f^{-1}(0)} |x\rangle \quad (3.10)$$

And  $|\beta\rangle$  as the sum over all states which are a solution

$$|\beta\rangle = \sum_{x \in f^{-1}(1)} |x\rangle \quad (3.11)$$

Initially, the state  $|\psi\rangle$  is in a superposition of all possible states:

$$|\psi\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{2^n-1} |x\rangle = \frac{1}{\sqrt{N}} |\alpha\rangle + \frac{1}{\sqrt{N}} |\beta\rangle \quad (3.12)$$

As explained in section 3.1.1, the oracle shifts the phase of the solution as follows:

$$O|x\rangle = \frac{1}{\sqrt{2^n}} |\alpha\rangle - \frac{1}{\sqrt{2^n}} |\beta\rangle \quad (3.13)$$

This can be rewritten as:

$$O|x\rangle = |\psi\rangle - \frac{3}{\sqrt{2^n}} |\beta\rangle \quad (3.14)$$

By substituting this result into the next step of the Grover algorithm (equation 3.9) we find:

$$(2|\psi\rangle\langle\psi| - I) \left( |\psi\rangle - \frac{3}{\sqrt{2^n}} |\beta\rangle \right) \quad (3.15)$$

By eliminating the brackets this leads to:

$$2|\psi\rangle\langle\psi|\psi\rangle - I|\psi\rangle - \frac{4}{\sqrt{2^n}} \langle\psi|\beta\rangle + \frac{2}{\sqrt{2^n}} I|\beta\rangle \quad (3.16)$$

We can simplify both inner products.  $\langle\psi|\psi\rangle = 2^n \frac{1}{\sqrt{2^n}} \frac{1}{\sqrt{2^n}} = 1$ . Since each  $\beta$  is a basis vector, we can use the identity  $\langle\psi|\beta\rangle = \langle\beta|\psi\rangle = \frac{1}{\sqrt{2^n}}$ . This leaves us with:

$$\begin{aligned} & 2|\psi\rangle - I|\psi\rangle - \frac{4}{\sqrt{2^n}} \frac{M}{\sqrt{2^n}} + \frac{2}{\sqrt{2^n}} |\beta\rangle \\ & \left( 1 - \frac{4M}{2^n} \right) |\psi\rangle + \frac{2}{\sqrt{2^n}} |\beta\rangle \end{aligned} \quad (3.17)$$

Where  $M$  is the number of solutions. When equation 3.12 is substituted we find:

$$\left( \frac{1}{\sqrt{2^n}} - \frac{4M}{2^{\frac{3}{2}n}} \right) |\alpha\rangle + \left( \frac{3}{\sqrt{2^n}} - \frac{4M}{2^{\frac{3}{2}n}} \right) |\beta\rangle \quad (3.18)$$

From this equation the probability of finding a solution when the first  $n$  qubits are measured can be calculated. The probability of measuring a state which is not a solution:

$$\left( \frac{1}{\sqrt{2^n}} - \frac{4M}{2^{\frac{3}{2}n}} \right)^2 \quad (3.19)$$

The probability of measuring a state which is a solution:

$$\left( \frac{3}{\sqrt{2^n}} - \frac{4M}{2^{\frac{3}{2}n}} \right)^2 \quad (3.20)$$

### 3.1.4 Worked example

The quantum circuit in figure 3.1 implements Grover's algorithm for a search problem with solutions  $|011\rangle, |101\rangle$  and  $|111\rangle$ . This example is chosen because these solutions translate to the prime numbers 3, 5 and 7. These are all prime numbers that can be made with 3 qubits, excluding 2. If 2 were to be included, more than half of the states would be solutions to the search problem and more iterations are needed. [7].

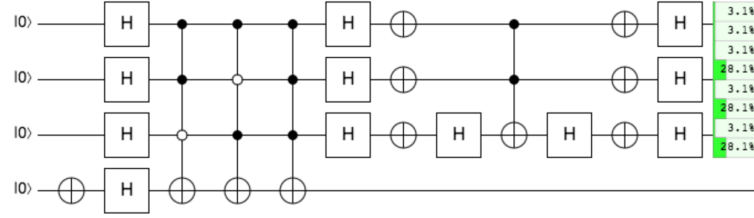


Figure 3.1: Quantum circuit of a Grover iteration for a search problem with solutions 3, 5 and 7. The table on the right shows that there is an 28.1 % chance of measuring the states  $|011\rangle, |101\rangle$  and  $|111\rangle$  and a 3.1 % chance of measuring the states  $|000\rangle, |001\rangle, |010\rangle, |100\rangle$  and  $|110\rangle$

If we calculate the probability of measuring a state which is solution and measuring a state which is not we find:

$$P_{\text{not a solution}} = \left( \frac{1}{\sqrt{2^3}} - \frac{4 \cdot 3}{2^{\frac{3}{2} \cdot 3}} \right)^2 = 0.03125$$

$$P_{\text{solution}} = \left( \frac{3}{\sqrt{2^3}} - \frac{4 \cdot 3}{2^{\frac{3}{2} \cdot 3}} \right)^2 = 0.28125$$
(3.21)

The measurement in figure 3.1 corresponds with these values.

### 3.1.5 Multiple Grover iterations

To find the probability to measure a solution after  $k$  Grover iterations define the state  $|\psi_k\rangle$  to be the state of the  $n$  qubits in the first register

$$|\psi_k\rangle = a_k |\alpha\rangle + b_k |\beta\rangle \quad (3.22)$$

where  $(b_k)^2$  is the probability of measuring a solution and  $(a_k)^2$  is the probability of measuring a state which is not a solution. The aim is to apply the Grover iterations such that  $b_k \approx 1$ .  $a_k$  and  $b_k$  change according to the following recursive relation [1]:

$$a_{k+1} = \frac{N-2M}{N}a_k - \frac{2M}{N}b_k$$

$$b_{k+1} = \frac{2(N-M)}{N}a_k + \frac{N-2M}{N}b_k \quad (3.23)$$

Using the derivation in section 3.1.3, this relation can be checked for  $a_1$  and  $b_1$  with  $a_0$  and  $b_0$  equal to  $\frac{1}{\sqrt{N}}$

$$\begin{aligned}
 a_1 &= \left( \frac{N-2M}{N} - \frac{2M}{N} \right) \frac{1}{\sqrt{N}} \\
 &= \frac{1}{\sqrt{N}} - \frac{4M}{N^{\frac{3}{2}}} \\
 b_1 &= \left( \frac{2N-M}{N} + \frac{N-2M}{N} \right) \frac{1}{\sqrt{N}} \\
 &= \frac{3}{\sqrt{N}} - \frac{4M}{N^{\frac{3}{2}}}
 \end{aligned} \tag{3.24}$$

Which is equivalent to the coefficients in equation 3.18. In general the recursive relation is solved by [1]:

$$\begin{aligned}
 a_k &= \frac{1}{\sqrt{N-M}} \cos((2k+1)\theta) \\
 b_k &= \frac{1}{\sqrt{M}} \sin((2k+1)\theta)
 \end{aligned} \tag{3.25}$$

where  $\sin \theta = \sqrt{\frac{M}{N}}$ .<sup>1</sup> The aim of applying multiple Grover iterations is decrease  $a_k$  to a small value. A perfect number of iterations  $\tilde{k}$  can be found by setting  $a_{\tilde{k}} = 0$ :

$$\begin{aligned}
 a_{\tilde{k}} &= \frac{1}{\sqrt{N-M}} \cos((2\tilde{k}+1)\theta) = 0 \\
 (2\tilde{k}+1)\theta &= \frac{\pi}{2} \\
 \tilde{k} &= \frac{\pi - 2\theta}{4\theta}
 \end{aligned} \tag{3.26}$$

Observe that  $\pi$  is an irrational number and that  $\theta = \frac{M}{N}$  is a rational number, and that therefore  $\tilde{k}$  is not an integer. The best choice for  $k$  is  $\lfloor \frac{\pi}{4\theta} \rfloor$ . The probability of measuring a state which is not a solution is derived by rewriting what we know about the error, namely  $|k - \tilde{k}| \leq 1/2$ , into the definition of  $a_k$ :

$$\begin{aligned}
 |(2k+1) - (2\tilde{k}+1)| &\leq 1 \\
 |(2k+1)\theta - (2\tilde{k}+1)\theta| &\leq \theta \\
 \left| (2k+1)\theta - \frac{\pi}{2} \right| &\leq \theta \\
 \left| \sin((2k+1)\theta) - \frac{\pi}{2} \right| &\leq |\sin \theta| \\
 |\cos((2k+1)\theta)| &\leq |\sin \theta| \\
 (N-M)a_k^2 &= \cos^2((2m+1)\theta) \leq \sin^2 \theta = \frac{M}{N}
 \end{aligned} \tag{3.27}$$

<sup>1</sup>In De Wolf's lecture notes on pages 48-49 and in Chuang and Nielsen's book on pages 252-253 a geometric argument is given which gives some intuition for these coefficients [3][7].

Thus the probability of measuring a state that is not a solution is negligible when  $M \ll N^2$ . Since  $\frac{\theta}{2} \geq \sin \frac{\theta}{2} = \sqrt{\frac{M}{N}}$ , the number of iterations required for attaining this probability is:

$$k \leq \frac{\pi}{4\theta} \leq \frac{\pi}{4} \sqrt{\frac{N}{M}}. \quad (3.28)$$

From this it can be concluded that with  $k = O(\sqrt{N/M})$  Grover iterations a solution to the search problem will be measured with high probability. This is a quadratic improvement over the  $O(N/M)$  oracle calls required by a classical algorithm.

---

<sup>2</sup>In this derivation I am still assuming the number of solutions is known. If  $M \ll N$  does not hold for a particular problem, extra qubits can be added to the search space such that  $M \ll N$ . This is an example of how quantum algorithms behave counterintuitive, as we expect it would be easier to find a solution when the search space is smaller, but for some cases the opposite is true.

## Chapter 4

# Grover primality oracle

The aim of the Grover primality oracle is to perform

$$|x\rangle \xrightarrow{O} (-1)^{f(x)} |x\rangle \quad (4.1)$$

where a function  $f(x) = 0$  if  $x$  is composite and  $f(x) = 1$  if  $x$  is prime. In this section a quantum implementation of the classical Miller-Rabin primality test is designed to be used as a Grover oracle. This oracle is used together with hadamard gates and a phase-shift oracle in Grover's algorithm to prepare the prime state.

First, the Grover primality oracle is built and used in Quirk to prepare the 4 qubit prime state. The circuit that is designed uses 14 qubits. For a 5 qubit prime state ProjectQ is used, as it is possible to use more than the 16 qubits which Quirk allows. In ProjectQ several gates needed to be created.

The circuit has two main differences with a usual grover iteration as described in chapter 2 and of which an example can be seen in figure 3.1. Firstly, the qubits are initialized in the state  $|1\rangle$ , not  $|0\rangle$ . This implies that the X-gates in the phase-shift oracle as seen in 3.1 are not needed. Secondly, it was found that when implementing the Miller-Rabin primality test, flipping states which are not solutions (composite numbers) would require less test-carrier qubits than the test-carrier qubits needed to flip solutions. Conveniently, the first qubit of all solutions (except the prime number 2) is 1, because all prime numbers are odd and therefore applying a Z gate at the end of the primality oracle causes that the phase of all primes is -1 and the phase of all composites is 1.

The circuit differs from the proposed circuit of Latorre and Sierra in two ways. Firstly, the figures in Latorre and Sierra show a boolean oracle, which requires more qubits than a phase shift oracle. Secondly, Latorre and Sierra's circuit requires more test carriers, as initially the prime numbers are flipped.

### 4.1 Miller-Rabin primality test

To determine if a number is prime or composite, two numbers  $s$  and  $d$  need to be found that satisfy  $x - 1 = 2^s d$ , where  $x$  and  $d$  are odd. Then a number called the witness must be chosen, denoted  $a$  and in the range  $1 \leq a < x$ . If the witness verifies the following test:

$$\begin{aligned} a^d &\not\equiv 1 \pmod{x} \\ a^{2^r d} &\not\equiv -1 \pmod{x} \end{aligned} \tag{4.2}$$

for all  $0 \leq r \leq s-1$ , then  $x$  is composite. If the test fails, the test needs to be rerun until either the test is verified or  $\log^2 x$  witnesses have been tested. If  $\log^2 x$  witnesses fail the test then  $x$  is prime according to the Generalized Riemann Hypothesis [6]. Firstly, to translate this test to a quantum algorithm a quantum circuit needs to be designed that finds  $s$  and  $d$ . Conveniently Latorre and Sierra found that  $s$  and  $d$  satisfy both  $x-1 = 2^s d$  and  $|x-1\rangle = |d\rangle |s\rangle$ . This is easiest to understand with an example; take the prime number  $x = 29$ .  $|x-1\rangle = |28\rangle = |11100\rangle$ .  $s$  is determined by the amount of trailing zeros when subtracting 1, in this example  $|s\rangle = |00\rangle$ : 2 trailing zeros.  $d$  is determined by the qubits leading the zeros, in this example  $|d\rangle = |111\rangle = 7$ . This choice of  $s$  and  $d$  satisfies the two equations  $29-1 = 2^2 7$  and  $|29-1\rangle = |111\rangle |00\rangle$ . If the number 2 were to be included in the input qubits, this method would have not been possible.

The essence of the quantum circuit is to perform the Miller-Rabin test in superposition. As can be seen in equation 4.2, the test is dependent on the value of  $s$ . This implies that the test can not be done on all  $x$  at the same time but a separate test must be ran for each value of  $s$ . These tests are done on a superpositions of all values of  $d$  for some  $s$ . In the quantum circuit an extra control bit, the top line in figure 4.1, is used to run the  $s$ -test on an input starting with the correct amount of zeros. For  $N = 4$ , the values of  $s$  for which tests need to be run are 1, 2 and 3. The first test  $s = 1$  is on all states that start with  $|11\rangle$ . These are the number 15, 11, 7 and 3. The second test  $s = 2$  is on the numbers 13 and 5 and the third test is on the numbers 1 and 9. The X gates are needed to prevent numbers getting tested multiple times. (If there were no X gates in front of the CNOT of the second test with  $s = 2$ , then 7 and 15 would again be tested).



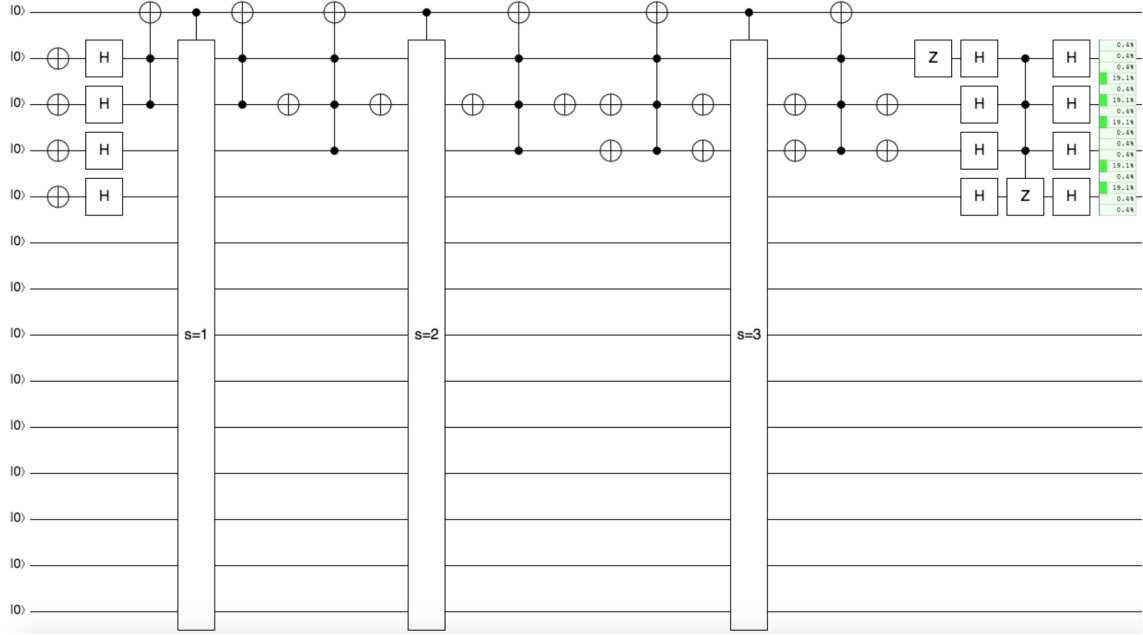


Figure 4.1: Grovers search algorithm for a four qubit prime state.

[6]

The  $s$ -test circuits consist of 13 qubits. Figure 4.2 shows the circuit for  $s=1$ . I will call the state of the first 4 qubits  $R$ , the state of qubits 5 through 7  $A$ , the state of qubits 8 through 11 the output and qubits 12 and 13 the test carriers. These names are chosen as they naturally follow from gates used in Quirk. The register  $R$  is in a superposition of the states :  $|0011\rangle$ ,  $|0101\rangle$ ,  $|0111\rangle$  and  $|1111\rangle$ . From the register  $R$  the value of  $d$  is copied into register  $A$  using CNOT gates. The output state is initialized to the state  $|0001\rangle$ , the witness is globally defined as 2 and the test carriers are both initialized to  $|1\rangle$ .

The first step of the Miller-Rabin primality test can be rewritten such that two available gates on Quirk can be used, namely the modular power multiplication gate  $B^A \bmod R$  and the modular decrement gate  $-1 \bmod R$ :

$$\begin{aligned} a^d &\not\equiv 1 \pmod{x} \\ a^{d \bmod x - 1} &\not\equiv 0 \end{aligned} \tag{4.3}$$

After applying these two gates the first test carrier is flipped if all four qubits in register  $R$  are 0. This is implemented by an X gate connected to four anti-control gates. For  $s = 1$ , the second step of the Miller-Rabin primality test can be rewritten such that only the modular increment

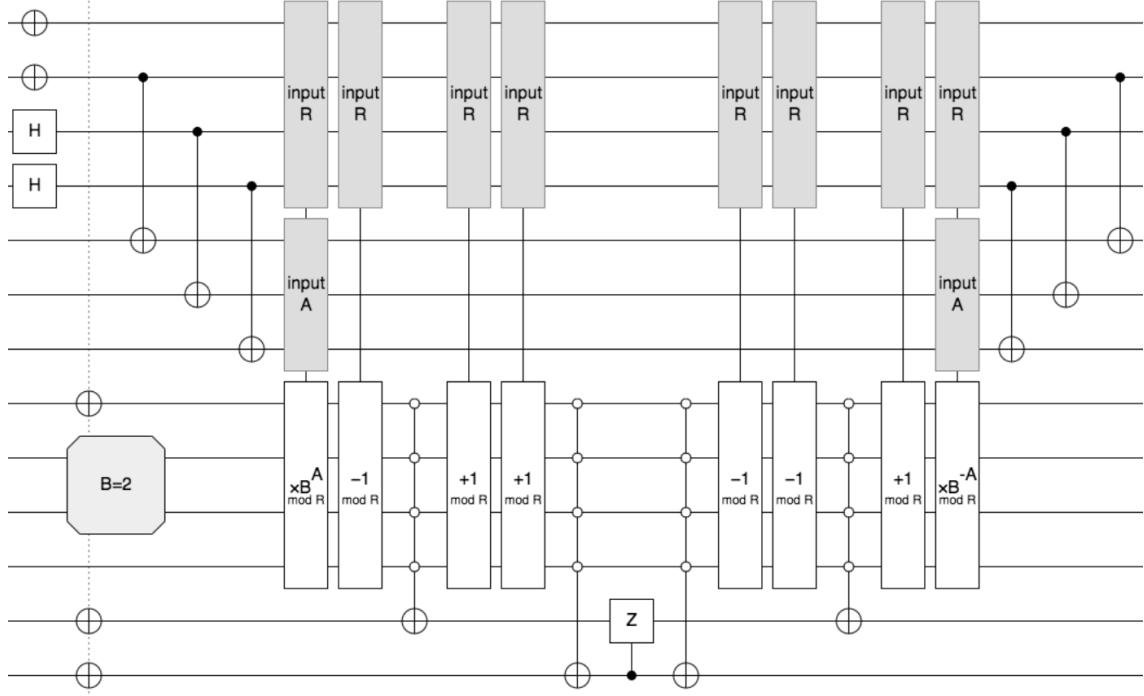
gate needs to be used:

$$\begin{aligned}
 a^{2^r d} &\not\equiv -1 \pmod{x} \\
 a^{2^0 d} \pmod{x} + 1 \pmod{x} &\not\equiv 0 \\
 a^d \pmod{x} - 1 \pmod{x} + 1 \pmod{x} + 1 \pmod{x} &\not\equiv 0
 \end{aligned} \tag{4.4}$$

Thus to perform the second step of the test two modular incrementation gates need to be applied. After this the second test carrier is flipped if all output bits are in the state 0. The following equations shows that only the phase of the state corresponding to 15 is changed by this circuit, because both test carriers are not flipped and remain in the state  $|1\rangle$ :

$$\begin{aligned}
 &\left. \begin{aligned} 2^1 \pmod{3} - 1 \pmod{3} &= 1 \\ 1 + 1 \pmod{3} + 1 \pmod{3} &= 0 \end{aligned} \right\} 3 \text{ is prime} \\
 &\left. \begin{aligned} 2^3 \pmod{7} - 1 \pmod{7} &= 0 \\ 0 + 1 \pmod{7} + 1 \pmod{7} &= 2 \end{aligned} \right\} 7 \text{ is prime} \\
 &\left. \begin{aligned} 2^5 \pmod{11} - 1 \pmod{11} &= 9 \\ 9 + 1 \pmod{11} + 1 \pmod{11} &= 0 \end{aligned} \right\} 11 \text{ is prime} \\
 &\left. \begin{aligned} 2^7 \pmod{15} - 1 \pmod{15} &= 7 \\ 7 + 1 \pmod{7} + 1 \pmod{7} &= 9 \end{aligned} \right\} 15 \text{ is composite}
 \end{aligned} \tag{4.5}$$

After the phase of the test-carriers is conditionally changed all qubits need to return to there input state. This is done by applying the inverse of each gate, starting with the last gates that have been applied before the phase shift and ending with the first gates of the circuit. For this to be possible all gates need to be reversible. Therefore the gates applying modular arithmetic functions do not perform the mathematical function they represent on all inputs. All gates only affect the input if it is smaller than the modulus. The modular increment gate  $+1 \pmod{R}$  adds 1 to the input and wraps  $R-1$  to 0. Its inverse, the modular decrement gate  $-1 \pmod{R}$  subtracts 1 from the input and wraps 0 to  $R-1$ . The modular power multiplication gate and modular power division gate do not change the input state if the modular multiplication would be irreversible. When a modular multiplication is irreversible is explained in section 4.2.1.


 Figure 4.2:  $s = 1$  circuit in Quirk

All three test circuits require 13 qubits, but the size of the state  $A$  and the amount of test carriers vary. In the  $s=2$  circuit in figure 4.3 state  $A$  consists of two qubits and three test carriers are used. The extra test carrier is needed because the second step of the Miller-Rabin test requires calculations for  $r = 0$  and  $r = 1$ . We can rewrite the calculation for  $r = 1$  such that we can again use the previously applied sequence of gates:

$$\begin{aligned}
 a^{2^1 d} &\not\equiv -1 \pmod{x} \\
 a^{2^d \bmod x + 1 \bmod x} &\not\equiv 0 \\
 (a^d \bmod x) a^d \bmod x + 1 \bmod x &\not\equiv 0 \\
 (a^d \bmod x - 1 \bmod x + 1 \bmod x + 1 \bmod -1 \bmod x) a^d \bmod x + 1 \bmod x &\not\equiv 0
 \end{aligned} \tag{4.6}$$

The  $s=2$  gate does not change the phase of either of its input states, 5 and 13. In the  $s=3$  circuit in figure 4.4 four test carriers are used. The test for  $r=2$  is  $a^{4d} \not\equiv -1 \pmod{x}$  and therefore the sequence of gates includes four modular multiplication gates.  $S=3$  changes the phase of both of its inputs, 1 and 9. When  $s=1$ ,  $s=2$  and  $s=3$  are used in a grover iteration, the probability display in Quirk in figure 4.1 shows that the probability of measuring 3, 5, 7, 11, 13 is 0.003906 and the probability of measuring a different state is 0.191406. These correspond with the probabilities

we would expect from the derivation in 3.1.3

$$P_{\text{not a solution}} = \left( \frac{1}{\sqrt{2^4}} - \frac{4 \cdot 5}{2^{\frac{3}{2} \cdot 4}} \right)^2 = 0.00390625$$

$$P_{\text{solution}} = \left( \frac{3}{\sqrt{2^3}} - \frac{4 \cdot 5}{2^{\frac{3}{2} \cdot 3}} \right)^2 = 0.19140625 \quad (4.7)$$

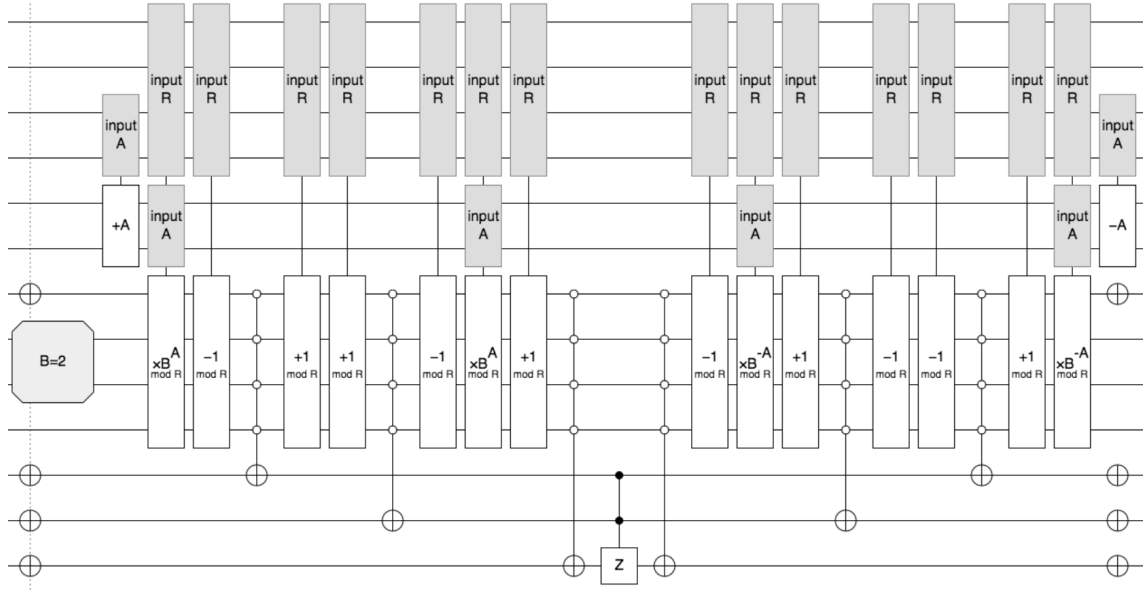
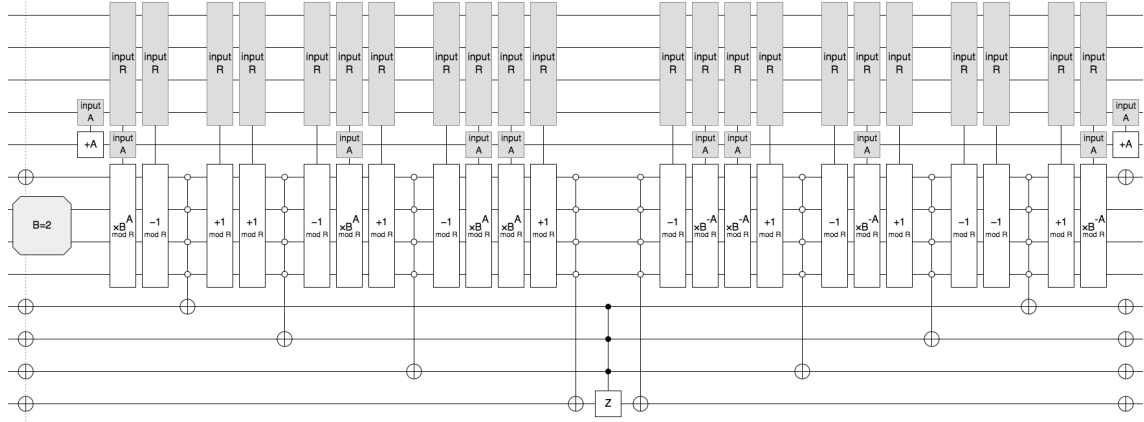


Figure 4.3:  $s = 2$  circuit in Quirk

Figure 4.4:  $s = 3$  circuit in Quirk

## 4.2 ProjectQ

The entire code for running the Grover primality oracle and preparing the 5 qubit prime state in ProjectQ can be found in appendix A. The main code file can be found in appendix A.1. It uses gates defined in the custom gates file, two of the gates defined in this file can be found in the next section and two gates in appendix A.2. At the end of the algorithm, a function `get_all_probabilities` and `get_all_amplitudes` is called, which is defined in the file `probabilities_and_amplitudes.py`, which can be found in appendix A.3. Information on the ProjectQ syntax and how to download ProjectQ can be found at <http://projectq.readthedocs.io/en/latest/tutorials.html>. The full code and test files can be found in my github repository <https://github.com/peter-janderks/bachelor-thesis>. The commenting in the file `customgates.py` is done in the same style as the comments in ProjectQ's source code, such that sections of the code can be added to the ProjectQ source code.

### 4.2.1 Custom gates in project Q

The `MultiplyModN` gate performs the function  $B^A \bmod R$ . In the code `B` is defined as the base, `A` the exponent and `R` is the modulus. The code uses the repeated squaring algorithm to efficiently compute modular exponentiation [8]:

---

```

1 ass MultiplyModN(BasicMathGate):
2     """
3     Takes three quantum registers as input, 'exponent', 'modulus' and 'val'.

```

---

```

4  The gate returns the exponent and modulus registers unchanged, and
5  multiplies the val register by base^exponent mod(modulus)
6  The numbers in quantum registers are stored from low- to high-bit, i.e.,
7  qunum[0] is the LSB.
8  """
9  def __init__(self, a):
10     """
11     Initializes the gate to the base to be used for modular
12     exponentiation.
13     Args:
14         base (int): base for the modular exponentiation.
15     It also initializes its base class, BasicMathGate, with the
16     corresponding function, so it can be emulated efficiently.
17
18     Example:
19         .. code-block:: python
20         MultiplyModN(2) | (qureg1, qureg2, qureg3)
21     """
22     def mod(exponent, modulus, val):
23         # does not change the val if it is bigger than the modulus (this
24         # would be irreversible) or if the modulus = 0 (division by 0 not
25         # possible)
26         if modulus == 0 or val >= modulus:
27             return (exponent, modulus, val)
28
29         else:
30             # repeated squaring algorithm
31             base = a
32             exp = exponent
33             while (exp > 0):
34                 # if exp is odd, multiply inverse with val
35                 if ((exp & 1) != 0):
36                     val = (val * base) % modulus
37                 # square the base
38                 base = (base * base) % modulus
39                 # binary shift
40                 exp >>= 1
41             return(exponent, modulus, val)

```

---

```

42     BasicMathGate.__init__(self,mod)
43     self.a = a

```

---

An example of the input given to the 'mod' function (line 22) when running the full circuit is  $2^{13} \bmod 27$ . This is part of the first step of the Miller-Rabin primality test for  $s=1$  and  $d = 13$ . The repeated squaring algorithm (lines 32-39) loops through the binary notation of the exponent, so it performs 4 iterations for this example. In each iteration, either the left most bit is 1, so the value is updated, or the left most bit is 0 and the value remains unchanged. In either case the base is squared. For  $2^{13} \bmod 27$  the functions works as follows: initially  $val = 1$ ,  $base = 2$  and  $exp = 13$ . After the first iteration  $val = 2$ ,  $base = 4$  and  $exp = 6$  (011). The second iteration does not change  $val$  because the left most qubit is 0, so at the end of the iteration  $base = 16$ ,  $val$  remains 2 and  $exp = 3$  (11). After the third iteration  $val = 5$ ,  $base = 13$  and  $exp = 1$ . The final iterations gives  $val = 11$ ,  $base = 7$  and  $exp = 0$ .  $Val = 11$  is the correct answer, because when  $2^{13} \bmod 27$  is directly calculated the answer is 11.

The inverse of the MultiplyModN gate, InverseMultiplyModN performs the function  $B^{-A} \bmod R$ , which can be rewritten as  $(B^{-1})^A \bmod R$ . To compute this the 'modular multiplicative inverse' of  $B$  needs to be calculated. The modular multiplicative inverse is defined as an integer  $C$  for which product  $BC$  is congruent with respect to the modulus  $R$ :

$$BC \equiv 1 \bmod R \quad (4.8)$$

The modular multiplicative inverse can be found using the extended Euclidean algorithm. The extended Euclidean algorithm determines the greatest common divisor (gcd) and the coefficients of Bézout's identity, which are integers  $x$  and  $y$  such that  $ax+by = \gcd(a,b)$ . If  $B$  has a modular multiplicative inverse this gcd must be 1. If this is the case, the inverse is equal to  $x \bmod R$ . If the inverse is found, fast modular exponentiation is used to calculate the modular power multiplication.

---

```

1  class InverseMultiplyModN(BasicMathGate):
2      """
3      Takes three quantum registers as input, 'exponent', 'modulus' and 'val'.
4      The gate returns the exponent and modulus registers unchanged, and
5      multiplies the val register by inverse^exponent mod(modulus). 'Inverse'
6      is the modular multiplicative inverse of 'base'.
7      The numbers in quantum registers are stored from low- to high-bit, i.e.,
8      qunum[0] is the LSB.
9      """
10     def __init__(self, a):
11         """

```

---

```

12     Initializes the gate to the base of which the inverse needs to be found
13     Args:
14         base (int): base of which the inverse needs to be found for modular
15             multiplication
16     It also initializes its base class, BasicMathGate, with the
17     corresponding function, so it can be emulated efficiently.
18     """
19     def extended_gcd(a, b):
20         # calculates the gcd and the coefficients of Bezout's identity
21         lastremainder, remainder = abs(a), abs(b)
22         x, lastx, y, lasty = 0, 1, 1, 0
23         while remainder != 0:
24             lastremainder, (quotient, remainder) = remainder, divmod(lastremainder, remainder)
25             x, lastx = lastx - quotient*x, x
26             y, lasty = lasty - quotient*y, y
27         return lastremainder, lastx, lasty
28
29     def find_inverse(value, mod):
30         # if the gcd is not equal to one, the value does not have an inverse.
31         # if it is, the inverse is the first Bezout coefficient modulus mod
32         (gcd, x, y) = extended_gcd(value, mod)
33         if gcd != 1:
34             return 'undefined'
35         return x % mod
36
37     def mod(exponent, modulus, val):
38         # does not change the val if it is bigger than the modulus (this
39         # would be irreversible) or if the modulus = 0 (division by 0 not
40         # possible)
41         base = a
42         if exponent == 0 or modulus == 0 or val >= modulus:
43             return (exponent, modulus, val)
44         else:
45             base = base
46             inverse = find_inverse(base, modulus)
47
48             if inverse == 'undefined':
49                 return (exponent, modulus, val)

```



---

```

50         exp = exponent
51
52         while (exp > 0):
53             # if exp is odd, multiply inverse with val
54             if ((exp & 1) != 0):
55                 val = (val * inverse) % modulus
56             # square the inverse (base)
57             inverse = (inverse * inverse) % modulus
58             # binary shift
59             exp >>= 1
60         return(exponent, modulus, val)
61     BasicMathGate.__init__(self, mod)
62     self.a = a

```

---

The code for the modular increment and decrement gates can be found in Appendix A.2.

### 4.2.2 Five qubit result

The code for the five qubit prime state returns the probabilities of measuring each state. The states that have a higher probability of being measured are 3, 5, 7, 11, 13, 17, 19, 21, 23, 27, 29 or 31. These are all prime numbers for  $\pi(x)$  excluding 2. The probability of measuring one of the primes is 0.09570312499999976. The probability of measuring a composite number is 0.001953124999999948. Thus it is  $\approx 49$  times more probable to measure a prime number than a composite number. The probabilities returned by the code are in correspondence with the equations in Section 4.2.1.

$$\begin{aligned}
 P_{\text{not a solution}} &= \left( \frac{1}{\sqrt{2^5}} - \frac{4 \cdot 10}{2^{\frac{5}{2} \cdot 5}} \right)^2 = 0.095703125 \\
 P_{\text{solution}} &= \left( \frac{3}{\sqrt{2^3}} - \frac{4 \cdot 5}{2^{\frac{3}{2} \cdot 3}} \right)^2 = 0.00195312499
 \end{aligned} \tag{4.9}$$

```

0.0019531249999999948 [0, 0, 0, 0, 0] 0
0.0019531249999999996 [1, 0, 0, 0, 0] 1
0.0019531249999999948 [0, 1, 0, 0, 0] 2
0.09570312499999976 [1, 1, 0, 0, 0] 3
0.0019531249999999948 [0, 0, 1, 0, 0] 4
0.09570312499999976 [1, 0, 1, 0, 0] 5
0.0019531249999999948 [0, 1, 1, 0, 0] 6
0.09570312499999972 [1, 1, 1, 0, 0] 7
0.0019531249999999948 [0, 0, 0, 1, 0] 8
0.0019531249999999991 [1, 0, 0, 1, 0] 9
0.0019531249999999948 [0, 1, 0, 1, 0] 10
0.09570312499999976 [1, 1, 0, 1, 0] 11
0.0019531249999999948 [0, 0, 1, 1, 0] 12
0.09570312499999972 [1, 0, 1, 1, 0] 13
0.0019531249999999948 [0, 1, 1, 1, 0] 14
0.0019531249999999972 [1, 1, 1, 1, 0] 15
0.0019531249999999948 [0, 0, 0, 0, 1] 16
0.09570312499999976 [1, 0, 0, 0, 1] 17
0.0019531249999999948 [0, 1, 0, 0, 1] 18
0.09570312499999976 [1, 1, 0, 0, 1] 19
0.0019531249999999948 [0, 0, 1, 0, 1] 20
0.0019531249999999972 [1, 0, 1, 0, 1] 21
0.0019531249999999948 [0, 1, 1, 0, 1] 22
0.09570312499999972 [1, 1, 1, 0, 1] 23
0.0019531249999999948 [0, 0, 0, 1, 1] 24
0.0019531249999999991 [1, 0, 0, 1, 1] 25
0.0019531249999999948 [0, 1, 0, 1, 1] 26
0.0019531249999999996 [1, 1, 0, 1, 1] 27
0.0019531249999999948 [0, 0, 1, 1, 1] 28
0.09570312499999972 [1, 0, 1, 1, 1] 29
0.0019531249999999948 [0, 1, 1, 1, 1] 30
0.09570312499999976 [1, 1, 1, 1, 1] 31

```

## Chapter 5

# Quantum Fourier transform and phase estimation

### 5.1 Quantum Fourier transform

The discrete Fourier transform takes as input a sequence of complex numbers  $\{x_n\} = x_0, x_1, \dots, x_{N-1}$  and outputs another sequence of complex numbers  $\{X_k\} = X_0, X_1, X_{N-1}$ , defined by:

$$\begin{aligned} X_k &= \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \\ &= \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x_n \omega_N^{kn} \end{aligned} \tag{5.1}$$

Where  $\omega_N^{ka}$  is the  $N^{th}$  root of unity. The quantum Fourier transform is the exact same transformation, on  $\{x_j |j\rangle\} = x_0 |0\rangle, x_1 |1\rangle, \dots, x_{N-1} |N-1\rangle$

$$\sum_{j=0}^{N-1} x_j |j\rangle = \sum_{k=0}^{N-1} X_k |k\rangle \tag{5.2}$$

Where  $x_j$  is the amplitude corresponding to state  $|j\rangle$  and  $N$  is equal to  $2^n$ , all possible combinations of  $n$  qubits. If the input is a basis state, meaning it is not in a superposition of different states and  $x_j = 1$  for some  $|j\rangle$  then equation 5.2 is equal to:

$$|j\rangle = \frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kj} |k\rangle \tag{5.3}$$

To give some intuition, the Fourier transform of the  $n^{th}$  qubit state  $|00\dots 0\rangle$  is:

$$\begin{aligned}
 |00\dots 0\rangle &= \sum_{k=0}^{N-1} e^{-\frac{2\pi i}{N} kn} |k\rangle \\
 |00\dots 0\rangle &= \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} e^{-\frac{2\pi i}{N} k0} |k\rangle \\
 &= \frac{1}{\sqrt{N}} \sum_{k=0}^{2^n-1} |k\rangle
 \end{aligned} \tag{5.4}$$

Which is an equal superposition of all states. To find a binary representation of state  $|j\rangle$  it is split up into bits:  $j_1, j_2 \dots j_n$ , with  $j = j_1 2^{n-1} + j_2 2^{n-2} + \dots + j_n 2^0$ . If  $j$  is a decimal number it is written as a binary fraction  $0.j_l j_{l+1} \dots j_m$  with  $j = j_l/2 + j_{l+1}/4 + \dots + j_m/2^{m-l+1}$ . The state  $|15.25\rangle$  is written as  $|1111.01\rangle$ . Equation 5.3 can be rewritten using the binary representation of  $|k\rangle : |k_1 \dots k_n\rangle$  and summing over the possible value (0 or 1) of each bit of  $k$ . In the exponential  $k/2^n$  is rewritten as  $\sum_{l=1}^{n-1} k_l 2^{-l}/2^n$  which can be reduced to  $\sum_{l=1}^n k_l 2^{-l}$ . Using this a product representation of the QFT can be found:

$$\begin{aligned}
 |j\rangle &\rightarrow \frac{1}{\sqrt{N}} \sum_{k_1=0}^1 \dots \sum_{k_n=0}^1 e^{-\frac{2\pi i}{N} j \sum_{l=1}^n k_l} |k_1 \dots k_n\rangle \\
 &= \frac{1}{\sqrt{N}} \sum_{k_1=0}^1 \dots \sum_{k_n=0}^1 \bigotimes_{l=1}^n e^{-\frac{2\pi i}{N} j k_l 2^{-l}} |k_l\rangle \\
 &= \frac{1}{\sqrt{N}} \bigotimes_{l=1}^n \left[ \sum_{k_l=0}^1 e^{-\frac{2\pi i}{N} j k_l 2^{-l}} |k_l\rangle \right] \\
 &= \frac{1}{\sqrt{N}} \bigotimes_{l=1}^n [|0\rangle + e^{2\pi i j 2^{-l}} |1\rangle]
 \end{aligned} \tag{5.5}$$

As an example, take  $n = 3$  such that the possible values of  $j$  are 0 up to and including 7. The first exponential term with  $l = 1$  can be rewritten using the binary representation of  $j$ :

$$e^{2\pi i j_1 j_2 j_3 / 2} = e^{2\pi i j_1 j_2 \cdot j_3} \tag{5.6}$$

Here the exponential term does not depend on  $j_1$  and  $j_2$ :

$$e^{2\pi i j_1 j_2} + e^{2\pi i 0 \cdot j_3} = e^{2\pi i 0 \cdot j_3} \tag{5.7}$$

The second exponential term depends on  $j_2$  and  $j_3$ :

$$e^{2\pi i j_1 j_2 j_3 / 4} = e^{2\pi i 0 \cdot j_2 j_3} \tag{5.8}$$

And the third exponential term depends on  $j_1$ ,  $j_2$  and  $j_3$ :

$$e^{2\pi i j_1 j_2 j_3 / 8} = e^{2\pi i 0 . j_1 j_2 j_3} \quad (5.9)$$

In general the exponential term does not depend on the  $n - l$  most significant bits:

$$e^{2\pi i j 2^{-l}} = e^{2\pi i 0 . j_{n-l+1} \dots j_n} \quad (5.10)$$

The product representation of the QFT for a  $n$  qubit input is <sup>1</sup>:

$$|j\rangle \rightarrow \frac{(|0\rangle + e^{2\pi i 0 . j_n} |1\rangle) \otimes (|0\rangle + e^{2\pi i 0 . j_{n-1} j_n} |1\rangle) \otimes (|0\rangle + e^{2\pi i 0 . j_1 j_2 \dots j_n} |1\rangle)}{\sqrt{N}} \quad (5.11)$$

### 5.1.1 Quantum circuit

The product representation (5.11) is useful as it shows per qubit what the output of an efficient circuit performing the quantum Fourier transform of the state  $|j_{q=1} \dots j_{q=n}\rangle$  should be. A sequence of gates starting with a Hadamard gate and followed by  $n - q$  phase gates is applied to each qubit in order from  $j_1$  to  $j_n$ . The circuit is shown in figure 5.1.1. The circuit starts with a Hadamard gate on the first qubit, which produces the state

$$\frac{1}{2} (|0\rangle + (-1)^{1 \cdot j_1} |1\rangle) |j_2 \dots j_n\rangle \quad (5.12)$$

Since  $(-1)^{j_1} = e^{2\pi i 0 . j_1}$  this is equivalent to

$$\frac{1}{2} (|0\rangle + e^{2\pi i 0 . j_1} |1\rangle) |j_2 \dots j_n\rangle \quad (5.13)$$

The matrices of the controlled rotational gates that are applied are

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{2\pi i / 2^k} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (5.14)$$

Each  $R_k$  gate adds an extra bit to the phase of the first qubits coefficient of  $|1\rangle$ . The first rotational gate applied in the quantum Fourier transform is controlled- $R_2$  on  $j_1$

$$|j_1\rangle = \frac{1}{\sqrt{2}} |0\rangle + e^{2\pi i 0 . j_1} |1\rangle \quad \begin{array}{c} \boxed{R_2} \\ \bullet \end{array} \quad \begin{array}{c} \text{---} \\ \text{---} \end{array}$$

<sup>1</sup>In literature, the tensor product symbol  $\otimes$  is often left out

If  $j_2 = |0\rangle$  the matrix representation of  $R_2$  times  $|j_1 j_2\rangle$  is

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{2\pi i/4} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ e^{2\pi i 0 \cdot j_1} \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ e^{2\pi i 0 \cdot j_1} \\ 0 \end{bmatrix} \quad (5.15)$$

If  $j_2 = |1\rangle$  the matrix representation of  $R_2$  times  $|j_1 j_2\rangle$  is:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{2\pi i/4} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ e^{2\pi i 0 \cdot j_1} \\ 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ e^{2\pi i(0 \cdot j_1 + 1/4)} \\ 0 \\ 1 \end{bmatrix} \quad (5.16)$$

So if  $j_2 = |0\rangle$ ,  $j_1$ 's state can be written as  $|0\rangle + e^{2\pi i(0 \cdot j_1 + 0/4)} |1\rangle$  and  $0/4 = 0.0j_2$ . If  $j_2 = |1\rangle$ ,  $j_1$  is in the state  $|0\rangle + e^{2\pi i 0 \cdot j_1 + 1/4} |1\rangle$  and  $1/4 = 0.0j_2$ . This shows the dependence of the state of  $j_2$  on  $j_1$ . After the controlled- $R_2$  gate the state of the qubits is

$$\frac{1}{\sqrt{2}} (|0\rangle + e^{2\pi i 0 \cdot j_1 j_2} |1\rangle) |j_2 \dots j_n\rangle \quad (5.17)$$

The next rotational gate, controlled- $R_3$  multiplies  $e^{2\pi i 0 \cdot j_1 j_2}$  with  $e^{2\pi i 1/8}$  if  $|j_3\rangle = 1$  or with  $e^{2\pi i 0/8}$  if  $|j_3\rangle = 0$ . This gives the state

$$\frac{1}{\sqrt{2}} (|0\rangle + e^{2\pi i 0 \cdot j_1 j_2 j_3} |1\rangle) |j_2 \dots j_n\rangle \quad (5.18)$$

After applying controlled- $R_4, R - 5$  through  $R_n$  the state of the qubits is

$$\frac{1}{2} (|0\rangle + e^{2\pi i 0 \cdot j_1 j_2 \dots j_n} |1\rangle) |j_2 \dots j_n\rangle \quad (5.19)$$

Next the same sequence of gates, a Hadamard gate now followed by  $n - 2$  controlled-rotational gates are applied to the second qubit. The Hadamard puts the second qubit in the state  $\frac{1}{\sqrt{2}} (|0\rangle + e^{2\pi i 0 \cdot j_2} |1\rangle)$  and the controlled-rotational gates produce the state  $\frac{1}{\sqrt{2}} (|0\rangle + e^{2\pi i 0 \cdot j_2} |1\rangle)$ . Thus after the sequence is applied to the second qubit the input is changed to the state

$$\frac{1}{2} (|0\rangle + e^{2\pi i 0 \cdot j_1 j_2 \dots j_n} |1\rangle) (|0\rangle + e^{2\pi i 0 \cdot j_2 \dots j_n} |1\rangle) |j_3 \dots j_n\rangle \quad (5.20)$$

After the sequence is applied to each qubit, the final state is

$$\frac{1}{2^{n/2}} (|0\rangle + e^{2\pi i 0 \cdot j_1 j_2 \dots j_n} |1\rangle) (|0\rangle + e^{2\pi i 0 \cdot j_2 \dots j_n} |1\rangle) \dots (|0\rangle + e^{2\pi i 0 \cdot j_n} |1\rangle) \quad (5.21)$$

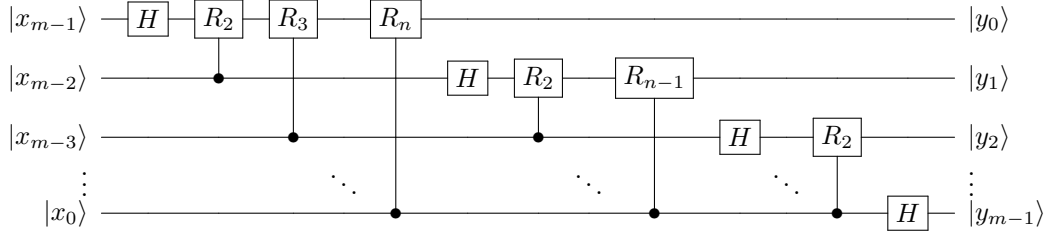


Figure 5.1: Circuit for the quantum Fourier transform. The swap gates at the end of the circuit are not shown.

This is equal to the product representation 5.11, except that the qubits are in reverse order. Thus swap operations, which consists of CNOT gates, are applied giving a final state equal to the product representation.

### 5.1.2 Complexity

The circuit in figure 5.1 computes the Fourier transform using  $\Theta(n^2)$  gates, while the best classical algorithm requires  $\Theta(n2^n)$  gates to compute the discrete Fourier transform [7]. Thus, it takes a classical computer exponentially more gates (operations) to calculate the discrete Fourier transform than it takes a quantum computer to calculate the quantum Fourier transform. Although they are the exact same transformation, the quantum Fourier transform can not be used for the same applications as the discrete Fourier transform due to two problems. To begin with, a fundamental property of quantum mechanics is that superpositions collapse when measured. Therefore the amplitude of the quantum Fourier transform can not be measured. Secondly, there is no efficient way to prepare the input qubits in a basis state  $|j_1, j_2 \dots j_n\rangle$  or a superposition  $\sum_{j=0}^{N-1} x_j |j\rangle$  corresponding to the data that needs to be transformed.

Even though the quantum Fourier transform can not be used to speed up real-world data processing, it is a powerful algorithm. This is shown in the next section where the quantum phase estimation algorithm is explained. It follows the general idea behind the design of many quantum algorithms; a function and final transformation is chosen such that useful global information about the function can be determined, which cannot be attained quickly on a classical computer. The final transformation is the inverse quantum Fourier transform and in chapter 5 the function used is the Grover iteration using the quantum primality oracle.

## 5.2 Phase estimation

The eigenvalue of a unitary operator  $U$  with an eigenvector  $|u\rangle$  can be written as  $e^{2\pi i\phi}$ . This can be proven as follows; let  $\lambda$  be an eigenvalue with eigenvector  $w \neq 0$  of a unitary matrix  $U$ . Using

the properties of a unitary matrix it can be shown that the modulus of  $\lambda = 1$

$$\begin{aligned}
 \langle w|w \rangle &= \langle UU^\dagger w|w \rangle \\
 &= \langle Uw|Uw \rangle \\
 &= \langle \lambda w|\lambda w \rangle \\
 &= \lambda \bar{\lambda} \langle w|w \rangle \\
 (1 - \lambda \bar{\lambda}) \langle w|w \rangle &= 0
 \end{aligned} \tag{5.22}$$

Since  $\langle w|w \rangle \neq 0$ ,  $\lambda \bar{\lambda} = |\lambda|^2 = 1$ . Thus all eigenvalues of a matrix  $\lambda = |\lambda|e^{i\theta}$  can be written in the form  $e^{i\theta}$  for some real  $\theta$ . The aim of the phase estimation algorithm is to estimate the phase  $\phi$  of a given unitary operator and corresponding eigenvector. Phase estimation should be seen as a subroutine that needs to be combined with other subroutines. In this section it is assumed that black boxes capable of preparing the eigenvector  $|u\rangle$  and controlled- $U^{2^j}$  are available.

### 5.2.1 The algorithm

The phase estimation circuit is shown in figure 5.2.1. The input consists of two quantum registers and the initial state is  $|0_1 \dots 0_t\rangle |u\rangle$ . The size of the first register is equal to the amount of bits needed to exactly express the phase of the unitary operation:  $\phi = 0.\phi_1 \dots \phi_t$ , assuming  $\phi$  may be expressed exactly in  $t$  bits. The size of the second register is equal to the size of the eigenvector  $|u\rangle$ .

The first step of the algorithm is to apply  $H^{\otimes t}$  to the first register to get

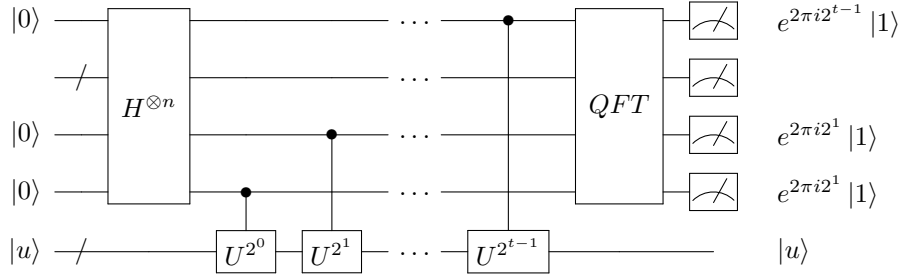
$$\frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle |u\rangle \tag{5.23}$$

The second step is applying  $t - 1$  controlled- $U$  gates, which act on the second register only if its corresponding control bit is  $|1\rangle$ . The action of first controlled- $U$  gate, applied to the last qubit of the first register, is

$$\begin{aligned}
 \frac{1}{\sqrt{2}} (|0\rangle + |1\rangle) |u\rangle &\xrightarrow{c-U} \frac{1}{\sqrt{2}} (|0\rangle |u\rangle + |1\rangle U |u\rangle) \\
 &= \frac{1}{\sqrt{2}} (|0\rangle |u\rangle + |1\rangle e^{2\pi i \phi} |u\rangle) \\
 &= \frac{1}{\sqrt{2}} (|0\rangle + e^{2\pi i \phi} |1\rangle) |u\rangle
 \end{aligned} \tag{5.24}$$

The eigenvalue of  $U^{2^x}$  is  $e^{2\pi i 2^x \phi}$ . Thus, the state of the first register after applying  $t-1$  controlled-





U gates as shown in figure 5.2.1 is

$$\frac{1}{\sqrt{2^t}} (|0\rangle + e^{2\pi i 2^{t-1} \phi} |1\rangle) \otimes (|0\rangle + e^{2\pi i 2^{t-2} \phi} |1\rangle) \otimes \dots \otimes (|0\rangle + e^{2\pi i 2^0 \phi} |1\rangle) \quad (5.25)$$

By representing  $\phi$  as a binary fraction we find

$$\begin{aligned} & \frac{1}{\sqrt{2^t}} (|0\rangle + e^{2\pi i 0.\phi_t} |1\rangle) \otimes (|0\rangle + e^{2\pi i 0.\phi_{t-1}\phi_t} |1\rangle) \otimes \dots \otimes (|0\rangle + e^{2\pi i 0.\phi_1\phi_2\dots\phi_t} |1\rangle) \\ &= \frac{1}{2^{t/2}} \sum_{k=0}^{2^t-1} e^{2\pi i \phi k} |k\rangle \end{aligned} \quad (5.26)$$

This is equal to the product representation of the quantum Fourier transform 5.11 applied to  $\phi$ . The last step of the algorithm is to apply the inverse Fourier transform transforming the state into exactly  $\phi$ . The final state of the two registers is  $|\phi\rangle |u\rangle$

## Chapter 6

# Quantum Counting

In this Chapter the Grover iteration (Chapter 2) will be combined with the phase estimation technique based upon the quantum Fourier transform (Chapter 4), to form the quantum counting algorithm. In the first section, I show that the Grover iteration can be written as a unitary matrix with eigenvalues  $e^{i\theta}$  and  $e^{i(2\pi-\theta)}$ . In section 6.2 the quantum counting algorithm using the grover iteration from section 3.1.4 is implemented in Quirk to calculate the prime counting function.

### 6.1 Matrix representation of the Grover iteration

To find the matrix representation of the Grover iteration the initial state is rewritten as the sum of two states. The first state sums over all states which are not solutions

$$|w\rangle = \frac{1}{\sqrt{N-M}} \sum_{x \in f^{-1}(0)} |x\rangle \quad (6.1)$$

The second state sums over all states which are solutions

$$|m\rangle = \frac{1}{\sqrt{M}} \sum_{x \in f^{-1}(1)} |x\rangle \quad (6.2)$$

The initial state is in the space spanned by  $|m\rangle$  and  $|w\rangle$ :

$$|\psi\rangle = \sqrt{\frac{N-M}{N}} |w\rangle + \sqrt{\frac{M}{N}} |m\rangle \quad (6.3)$$

with  $|m\rangle \perp |w\rangle$  because  $|x_i\rangle |x_j\rangle = \delta_{ij}$ . If  $\theta$  is chosen such that  $\sin^2 \frac{\theta}{2} = \frac{M}{N}$ , then

$$\cos^2 \frac{\theta}{2} = 1 - \sin^2 \frac{\theta}{2} = \frac{N-M}{N}. \quad (6.4)$$

Thus the initial state can be rewritten as

$$|\psi\rangle = \cos \frac{\theta}{2} |w\rangle + \sin \frac{\theta}{2} |m\rangle. \quad (6.5)$$

Using these coefficients a matrix representation of the Grover iteration in the  $|m\rangle, |w\rangle$  basis can be derived,

$$\begin{aligned}
G|w\rangle &= (2|\psi\rangle\langle\psi| - I)O|w\rangle \\
&= 2|\psi\rangle\langle\psi|w\rangle - |w\rangle \\
&= 2|\psi\rangle\cos\frac{\theta}{2} - |w\rangle \\
&= 2\cos\frac{\theta}{2}\left(\cos\frac{\theta}{2}|w\rangle + \sin\frac{\theta}{2}|m\rangle\right) - |w\rangle \\
&= \left(2\cos^2\frac{\theta}{2} - 1\right)|w\rangle - 2\sin\frac{\theta}{2}\cos\frac{\theta}{2}|m\rangle \\
&= \cos\theta|w\rangle + \sin\theta|m\rangle \\
G|m\rangle &= (2|\psi\rangle\langle\psi| - I)O|m\rangle \\
&= -2|\psi\rangle\langle\psi|m\rangle + |m\rangle \\
&= -2|\psi\rangle\sin\frac{\theta}{2} + |m\rangle \\
&= -2\sin\theta\left(\cos\frac{\theta}{2}|w\rangle + \sin\frac{\theta}{2}|m\rangle\right) + |m\rangle \\
&= -2\sin\frac{\theta}{2}\cos\frac{\theta}{2}|w\rangle + (1 - 2\sin^2\frac{\theta}{2})|m\rangle \\
&= -\sin\theta|w\rangle + \cos\theta|m\rangle
\end{aligned} \tag{6.6}$$

Thus the matrix representation of  $G$  in the basis  $|m\rangle, |w\rangle$  is

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{6.7}$$

## 6.2 Circuit in Quirk

In figure 6.1 the circuit for approximate quantum counting is shown. The circuit is included in this thesis to show how the circuit should be constructed. The measurement of this circuit is yet to be confirmed with what is expected from theory. Future research needs to be done to understand the probabilities. The Grover iteration in the circuit is from figure 3.1 in section 3.1.4. The number of qubits in each register is chosen such that following from a derivation from Chuang and Nielsen on pages 262-263 an estimate of  $\theta$  or  $2\pi - \theta$  accurate within  $|\Delta\theta| \leq 2^{-m}$  with probability at least  $5/6$  is expected. In future research it would be better to construct the quantum counting algorithm with ProjectQ than Quirk. It is time consuming to work with big custom gates in Quirk and once built the custom gates can not be changed. For the counting algorithm to be built in ProjectQ, first an inverse quantum Fourier transform circuit needs to be built.

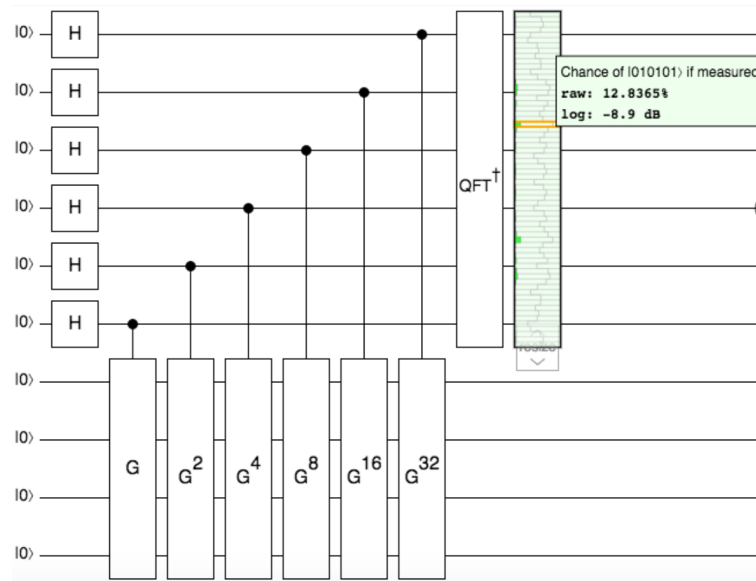


Figure 6.1:

## Chapter 7

# Conclusion

The classical Miller-Rabin primality test can be converted into a quantum primality oracle which can be used in Grover's algorithm to prepare a register in the prime state. In Quirk the 4 qubit prime state can be prepared using 14 qubits and in ProjectQ the 5 qubit prime state can be prepared using 17 qubits. The Grover iterations of these circuits can be written as a unitary matrix of which the phase of the eigenvalue can be estimated using the phase estimation algorithm. The phase estimation algorithm returns the prime counting function with an error which is smaller than that of the Riemann Hypothesis. The main focus of future research could be to code this phase estimation in ProjectQ using the code in this thesis and to derive the error bounds of probabilities to verify if the circuit works as expected. The time and memory complexity of the Grover iteration preparing the prime states remains to be analyzed in more detail. This could be possibly be done by expanding the circuit and using ProjectQ's resource counter.

# Bibliography

- [1] M. Boyer, G. Brassard, P. Høyer, and A. Tapp. Tight bounds on quantum searching. *Fortschritte der Physik*, 46(45):493–505, June 1998. ISSN 0015-8208. URL [arXiv:quant-ph/9605034](#).
- [2] G. Brassard, P. Hoyer, and A. Tapp. Quantum counting. *Automata, Languages And Programming*, 1443:820–831, May 1998. URL [arXiv:quant-ph/9805082](#).
- [3] R. de Wolf. Quantum computing: Lecture notes. 2016. URL <https://homepages.cwi.nl/~rdeewolf/qcnotes.pdf>.
- [4] L. K. Grover. A fast quantum mechanical algorithm for database search. May 1996. URL [arXiv:quant-ph/9605043](#).
- [5] T. Häner, D. S. Steiger, K. Svore, and M. Troyer. A software methodology for compiling quantum programs. April 2016. URL <https://arxiv.org/abs/1604.01401>.
- [6] J. I. Latorre and G. Sierra. Quantum computation of prime number functions. February 2013. URL <https://arxiv.org/pdf/1302.6245.pdf>.
- [7] M. A. Nielsen and I. I. Chuang. *Quantum computation and quantum information*. Cambridge University Press, 2000. ISBN 0521632358.
- [8] V. Shoup. Cambridge University Press.
- [9] D. S. Steiger, T. Häner, and M. Troyer. Projectq: an open source software framework for quantum computing. *Quantum - the open journal for quantum science*, 2, January 2018. ISSN 2521-327X. URL <https://doaj.org/article/2e2efd4f3a454a11abefffe748cf24f8>.

# Chapter 8

## Appendix A

### 8.1 5 Qubit Prime State code

---

```
1 from projectq.ops import C, BasicMathGate,H, Z, X, Measure, All
2 from projectq.meta import Loop, Compute, Uncompute, Control
3 from projectq.backends._sim._simulator import Simulator
4 from customgates import *
5 from probabilities_and_amplitudes import *
6
7 def s1(eng, qubits, R):
8     A = qubits[6:10]
9     output_reg = qubits[10:15]
10    test_carrier_1 = qubits[15]
11    test_carrier_2 = qubits[16]
12
13    C(X,1) | (R[1], A[0])
14    C(X,1) | (R[2], A[1])
15    C(X,1) | (R[3], A[2])
16    C(X,1) | (R[4], A[3])
17
18    X | output_reg[0]
19    X | test_carrier_1
20    X | test_carrier_2
21
22    MultiplyModN(2) | (A,R,output_reg)
23    modular_decrement_gate() | (R,output_reg)
24
25    with Compute(eng):
26        All(X) | output_reg
27    with Control(eng, output_reg):
28        X | test_carrier_1
```

```

29    Uncompute(eng)
30
31    modular_increment_gate() | (R,output_reg)
32    modular_increment_gate() | (R,output_reg)
33
34    with Compute(eng):
35        All(X) | output_reg
36    with Control(eng, output_reg):
37        X | test_carrier_2
38    Uncompute(eng)
39
40    C(Z,1) | (test_carrier_2,test_carrier_1)
41
42    with Compute(eng):
43        All(X) | output_reg
44    with Control(eng, output_reg):
45        X | test_carrier_2
46    Uncompute(eng)
47
48    modular_decrement_gate() | (R,output_reg)
49    modular_decrement_gate() | (R,output_reg)
50
51    with Compute(eng):
52        All(X) | output_reg
53    with Control(eng, output_reg):
54        X | test_carrier_1
55    Uncompute(eng)
56
57    modular_increment_gate() | (R,output_reg)
58    InverseMultiplyModN(2) | (A,R,output_reg)
59
60    C(X,1) | (R[4], A[3])
61    C(X,1) | (R[3], A[2])
62    C(X,1) | (R[2], A[1])
63    C(X,1) | (R[1], A[0])
64
65    X | output_reg[0]
66    X | test_carrier_1

```



```

67     X | test_carrier_2
68
69 def s2(eng,qubits,R):
70
71     A = qubits[6:9]
72     output_reg = qubits[9:14]
73     test_carrier_1 = qubits[14]
74     test_carrier_2 = qubits[15]
75     test_carrier_3 = qubits[16]
76
77     X | output_reg[0]
78     X | test_carrier_1
79     X | test_carrier_2
80     X | test_carrier_3
81
82     C(X,1) | (R[2], A[0])
83     C(X,1) | (R[3], A[1])
84     C(X,1) | (R[4], A[2])
85
86     MultiplyModN(2) | (A,R,output_reg)
87     modular_decrement_gate() | (R,output_reg)
88
89     with Compute(eng):
90         All(X) | output_reg
91     with Control(eng, output_reg):
92         X | test_carrier_1
93     Uncompute(eng)
94
95     modular_increment_gate() | (R,output_reg)
96     modular_increment_gate() | (R,output_reg)
97
98     with Compute(eng):
99         All(X) | output_reg
100    with Control(eng, output_reg):
101        X | test_carrier_2
102    Uncompute(eng)
103
104    modular_decrement_gate() | (R,output_reg)

```

```

105     MultiplyModN(2) | (A,R,output_reg)
106     modular_increment_gate() | (R,output_reg)
107
108     with Compute(eng):
109         All(X) | output_reg
110     with Control(eng, output_reg):
111         X | test_carrier_3
112     Uncompute(eng)
113
114     C(Z,2) | (test_carrier_1,test_carrier_2,test_carrier_3)
115
116     with Compute(eng):
117         All(X) | output_reg
118     with Control(eng, output_reg):
119         X | test_carrier_3
120     Uncompute(eng)
121
122     modular_decrement_gate() | (R, output_reg)
123     InverseMultiplyModN(2) | (A,R,output_reg)
124     modular_increment_gate() | (R,output_reg)
125
126     with Compute(eng):
127         All(X) | output_reg
128     with Control(eng, output_reg):
129         X | test_carrier_2
130     Uncompute(eng)
131
132     modular_decrement_gate() | (R, output_reg)
133     modular_decrement_gate() | (R, output_reg)
134
135     with Compute(eng):
136         All(X) | output_reg
137     with Control(eng, output_reg):
138         X | test_carrier_1
139     Uncompute(eng)
140
141     modular_increment_gate() | (R, output_reg)
142     InverseMultiplyModN(2) | (A,R,output_reg)

```

```

143
144     X | output_reg[0]
145     X | test_carrier_1
146     X | test_carrier_2
147     X | test_carrier_3
148     C(X,1) | (R[4], A[2])
149     C(X,1) | (R[3], A[1])
150     C(X,1) | (R[2], A[0])
151
152 def s3(eng, qubits, R):
153     A = qubits[6:8]
154     output_reg = qubits[8:13]
155     test_carrier_1 = qubits[13]
156     test_carrier_2 = qubits[14]
157     test_carrier_3 = qubits[15]
158     test_carrier_4 = qubits[16]
159
160     X | output_reg[0]
161     X | test_carrier_1
162     X | test_carrier_2
163     X | test_carrier_3
164     X | test_carrier_4
165
166     C(X,1) | (R[3], A[0])
167     C(X,1) | (R[4], A[1])
168
169     MultiplyModN(2) | (A,R,output_reg)
170     modular_decrement_gate() | (R, output_reg)
171
172     with Compute(eng):
173         All(X) | output_reg
174     with Control(eng, output_reg):
175         X | test_carrier_1
176     Uncompute(eng)
177
178     modular_increment_gate() | (R, output_reg)
179     modular_increment_gate() | (R, output_reg)
180

```

```

181     with Compute(eng):
182         All(X) | output_reg
183     with Control(eng, output_reg):
184         X | test_carrier_2
185     Uncompute(eng)
186
187     modular_decrement_gate() | (R,output_reg)
188     MultiplyModN(2) | (A,R,output_reg)
189     modular_increment_gate() | (R,output_reg)
190
191     with Compute(eng):
192         All(X) | output_reg
193     with Control(eng, output_reg):
194         X | test_carrier_3
195     Uncompute(eng)
196
197     modular_decrement_gate() | (R,output_reg)
198     MultiplyModN(4) | (A,R,output_reg)
199     modular_increment_gate() | (R,output_reg)
200
201     with Compute(eng):
202         All(X) | output_reg
203     with Control(eng, output_reg):
204         X | test_carrier_4
205     Uncompute(eng)
206
207     C(Z,3) | (test_carrier_1,test_carrier_2,test_carrier_3,test_carrier_4)
208
209     with Compute(eng):
210         All(X) | output_reg
211     with Control(eng, output_reg):
212         X | test_carrier_4
213     Uncompute(eng)
214
215     modular_decrement_gate() | (R,output_reg)
216     InverseMultiplyModN(4) | (A,R,output_reg)
217     modular_increment_gate() | (R,output_reg)
218

```

```

219     with Compute(eng):
220         All(X) | output_reg
221     with Control(eng, output_reg):
222         X | test_carrier_3
223     Uncompute(eng)
224
225     modular_decrement_gate() | (R,output_reg)
226     InverseMultiplyModN(2) | (A,R,output_reg)
227     modular_increment_gate() | (R,output_reg)
228
229     with Compute(eng):
230         All(X) | output_reg
231     with Control(eng, output_reg):
232         X | test_carrier_2
233     Uncompute(eng)
234
235     modular_decrement_gate() | (R,output_reg)
236     modular_decrement_gate() | (R,output_reg)
237
238     with Compute(eng):
239         All(X) | output_reg
240     with Control(eng, output_reg):
241         X | test_carrier_1
242     Uncompute(eng)
243
244     modular_increment_gate() | (R,output_reg)
245     InverseMultiplyModN(2) | (A,R,output_reg)
246
247     C(X,1) | (R[3], A[0])
248     C(X,1) | (R[4], A[1])
249
250     X | output_reg[0]
251     X | test_carrier_1
252     X | test_carrier_2
253     X | test_carrier_3
254     X | test_carrier_4
255
256 def s4(eng,qubits,R):

```

```

257     A = qubits[6]
258     output_reg = qubits[7:12]
259     test_carrier_1 = qubits[12]
260     test_carrier_2 = qubits[13]
261     test_carrier_3 = qubits[14]
262     test_carrier_4 = qubits[15]
263     test_carrier_5 = qubits[16]
264
265     X | output_reg[0]
266     X | test_carrier_1
267     X | test_carrier_2
268     X | test_carrier_3
269     X | test_carrier_4
270     X | test_carrier_5
271
272     C(X,1) | (R[4], A)
273
274     MultiplyModN(2) | (A,R,output_reg)
275     modular_decrement_gate() | (R, output_reg)
276
277     with Compute(eng):
278         All(X) | output_reg
279     with Control(eng, output_reg):
280         X | test_carrier_1
281     Uncompute(eng)
282
283     modular_increment_gate() | (R, output_reg)
284     modular_increment_gate() | (R, output_reg)
285
286     with Compute(eng):
287         All(X) | output_reg
288     with Control(eng, output_reg):
289         X | test_carrier_2
290     Uncompute(eng)
291
292     modular_decrement_gate() | (R,output_reg)
293     MultiplyModN(2) | (A,R,output_reg)
294     modular_increment_gate() | (R,output_reg)

```

```

295
296     with Compute(eng):
297         All(X) | output_reg
298     with Control(eng, output_reg):
299         X | test_carrier_3
300     Uncompute(eng)
301
302     modular_decrement_gate() | (R,output_reg)
303     MultiplyModN(4) | (A,R,output_reg)
304     modular_increment_gate() | (R,output_reg)
305
306     with Compute(eng):
307         All(X) | output_reg
308     with Control(eng, output_reg):
309         X | test_carrier_4
310     Uncompute(eng)
311
312     modular_decrement_gate() | (R,output_reg)
313     MultiplyModN(4) | (A,R,output_reg)
314     modular_increment_gate() | (R,output_reg)
315
316     with Compute(eng):
317         All(X) | output_reg
318     with Control(eng, output_reg):
319         X | test_carrier_5
320     Uncompute(eng)
321
322     C(Z,4) | (test_carrier_1,test_carrier_2,test_carrier_3,test_carrier_4, test_carrier_5)
323
324     with Compute(eng):
325         All(X) | output_reg
326     with Control(eng, output_reg):
327         X | test_carrier_5
328     Uncompute(eng)
329
330     modular_decrement_gate() | (R,output_reg)
331     InverseMultiplyModN(4) | (A,R,output_reg)
332     modular_increment_gate() | (R,output_reg)

```

```

333
334     with Compute(eng):
335         All(X) | output_reg
336     with Control(eng, output_reg):
337         X | test_carrier_4
338     Uncompute(eng)
339
340     modular_decrement_gate() | (R,output_reg)
341     InverseMultiplyModN(4) | (A,R,output_reg)
342     modular_increment_gate() | (R,output_reg)
343
344     with Compute(eng):
345         All(X) | output_reg
346     with Control(eng, output_reg):
347         X | test_carrier_3
348     Uncompute(eng)
349
350     modular_decrement_gate() | (R,output_reg)
351     InverseMultiplyModN(2) | (A,R,output_reg)
352     modular_increment_gate() | (R,output_reg)
353
354     with Compute(eng):
355         All(X) | output_reg
356     with Control(eng, output_reg):
357         X | test_carrier_2
358     Uncompute(eng)
359
360     modular_decrement_gate() | (R,output_reg)
361     modular_decrement_gate() | (R,output_reg)
362
363     with Compute(eng):
364         All(X) | output_reg
365     with Control(eng, output_reg):
366         X | test_carrier_1
367     Uncompute(eng)
368
369     modular_increment_gate() | (R,output_reg)
370     InverseMultiplyModN(2) | (A,R,output_reg)

```



```

371
372     X | output_reg[0]
373     X | test_carrier_1
374     X | test_carrier_2
375     X | test_carrier_3
376     X | test_carrier_4
377     X | test_carrier_5
378
379     C(X,1) | (R[4], A)
380
381 def grover_iteration(qubits, R):
382     control_qubit = qubits[0]
383
384     All(X) | R[0:5]
385     All(H) | R[0:5]
386
387     ####
388     C(X,2) | (R[0:2],control_qubit)
389     with Control(eng,control_qubit):
390         s1(eng, qubits,R)
391     C(X,2) | (R[0:2],control_qubit)
392     ####
393     X | R[1]
394     C(X,3) | (R[0:3], control_qubit)
395     X | R[1]
396
397     with Control(eng,control_qubit):
398         s2(eng, qubits,R)
399
400     X | R[1]
401     C(X,3) | (R[0:3], control_qubit)
402     X | R[1]
403     ###
404
405     All(X) | R[1:3]
406     C(X,4) | (R[0:4], control_qubit)
407     All(X) | R[1:3]
408

```

---

```

409     with Control(eng,control_qubit):
410         s3(eng, qubits, R)
411
412     All(X) | R[1:3]
413     C(X,4) | (R[0:4], control_qubit)
414     All(X) | R[1:3]
415
416     ###
417
418     All(X) | R[1:4]
419     C(X,4) | (R[0:4], control_qubit)
420     All(X) | R[1:4]
421
422     with Control(eng,control_qubit):
423         s4(eng, qubits, R)
424
425     All(X) | R[1:4]
426     C(X,4) | (R[0:4], control_qubit)
427     All(X) | R[1:4]
428
429     ###
430     Z | R[0]
431     All(H) | R[0:5]
432     C(Z,4) | (R[0:4],R[4])
433     All(H) | R[0:5]
434     return (qubits,R)
435
436 if __name__ == "__main__":
437     eng = MainEngine()
438     qubits = eng.allocate_quireg(17)
439     R = qubits[1:6]
440     (qubits,R) = grover_iteration(qubits,R)
441     eng.flush()
442     print(get_all_probabilities(eng,R))
443     print(get_all_amplitudes(eng,qubits))
444     Measure | qubits

```

---

## 8.2 Modular Increment and Decrement Gates

---

```
1 from projectq import *
2 from projectq.ops import C, BasicMathGate,H, Z, X, Measure, All
3 from projectq.meta import Loop, Compute, Uncompute, Control
4 from projectq.backends._sim._simulator import Simulator
5
6 class modular_increment_gate(BasicMathGate):
7     """
8     Takes two quantum registers as input 'modulus' and 'val'.
9     The gate returns the modulus register unchanged and adds +1 mod (modulus)
10    to the val register (in a way such that this addition is reversible)
11    The numbers in quantum registers are stored from low- to high-bit, i.e.,
12    qunum[0] is the LSB.
13    """
14    def __init__(self):
15        """
16        Initializes the gate to its base class, BasicMathGate, with the
17        corresponding function, so it can be emulated efficiently.
18        """
19        def mod(modulus, val):
20            if modulus == 0 or val >= modulus:
21                return(modulus, val)
22            else:
23                if val+1 == modulus:
24                    return(modulus, 0)
25                else:
26                    return(modulus, val+1)
27            BasicMathGate.__init__(self, mod)
28
29 class modular_decrement_gate(BasicMathGate):
30     """
31     Takes two quantum registers as input 'modulus' and 'val'.
32     The gate returns the modulus register unchanged and subtracts -1 mod (modulus)
33     from the val register (in a way such that this subtraction is reversible)
34     The numbers in quantum registers are stored from low- to high-bit, i.e.,
35     qunum[0] is the LSB.
```

```
36     """
37     def __init__(self):
38         """
39         Initializes the gate to its base class, BasicMathGate, with the
40         corresponding function, so it can be emulated efficiently.
41         """
42         def mod(modulus, val):
43             if modulus == 0 or val >= modulus:
44                 return(modulus, val)
45             if val == 0:
46                 return(modulus, modulus-1)
47             else:
48                 return(modulus, val-1)
49 BasicMathGate.__init__(self, mod)
```

---

## 8.3 Probility and Amplitude Functions

---

```

1 from projectq.backends._sim._simulator import Simulator
2
3 def get_all_probabilities(eng, qureg):
4     i = 0
5     y = len(qureg)
6     while i < (2**y):
7         qubit_list = [int(x) for x in list('{0:0b}'.format(i)).zfill(y)]
8         qubit_list = qubit_list[::-1]
9         l = eng.backend.get_probability(qubit_list, qureg)
10        if l != 0.0:
11            print(l, qubit_list, i)
12        i += 1
13
14 def get_all_amplitudes(eng, qureg):
15     i = 0
16     y = len(qureg)
17     while i < (2**y):
18         qubit_list = [int(x) for x in list('{0:0b}'.format(i)).zfill(y)]
19         l = eng.backend.get_amplitude(qubit_list, qureg)
20         if l != 0.0:
21             print(l, qubit_list, i)
22         i += 1

```

---