Faktorisierung auf dem Quantencomputer Einführung und Implementierung

Einführung in die Funktionsweise von Quantencomputern und Implementierung zweier Programmbibliotheken in Q# mit arithmetischen Operationen und einer vollständigen Implementierung des Shor-Algorithmus

Maturaarbeit von	Joël Benjamin Huber
Betreut von	Christian Steiger
an der	Kantonsschule Freudenberg Zürich
abgegeben am	15. Dezember 2020

Zusammenfassung: *Die vorliegende Maturitätsarbeit beschäftigt sich mit Quantencomputern und ihrer Anwendung für das Faktorisierungsproblem. Zu diesem Zweck habe ich arithmetische Operationen genauer betrachtet, um zu verstehen, wie sie auf Quantencomputern implementiert werden können. In einem weiteren Schritt habe ich mich mit dem Shor-Algorithmus auseinandergesetzt, mit welchem sich Zahlen faktorisieren lassen. Dies alles habe ich in der Form von zwei Programmbibliotheken in der Quantenprogrammiersprache Q# und der klassischen Programmiersprache C# implementiert, mit dem Ziel, dass man diese Operationen später auch in anderen Projekten verwenden kann.*

This is only a foretaste of what is to come,
and only the shadow of what is going to be.

ALAN TURING

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Vorwort

Das Gebiet der Quantencomputer ist aktuell zweifellos eines der vielversprechendsten Forschungsgebiete. Quantencomputer würden es uns theoretisch erlauben, Berechnungen auszuführen, die auf klassischen Computern nicht effizient ausgeführt werden können. Diese mathematischen Konstrukte werden in naher Zukunft mit Sicherheit eine grosse Rolle spielen.

Immer wieder liest man in Zeitungs- und Journalartikeln über Realisierungen von Quantencomputern und immer wieder wurde von einem Durchbruch gesprochen. So zum Beispiel, als ein Forschungsteam von Google 2019 einen Artikel im Journal "Nature" publizierte, in welchem behauptet wurde, man habe die "Quantenüberlegenheit" erreicht ([1]), was gleich darauf von der IBM in ihrem Blog angezweifelt wurde ([14]).

In dieser Hinsicht ist es meines Erachtens ein guter Zeitpunkt, um sich vertieft mit dem Thema auseinanderzusetzen. Ich habe mich deshalb entschieden, mich auf diese Reise zu begeben und im Rahmen meiner Maturitätsarbeit zu versuchen zu verstehen, wie Quantencomputer funktionieren, um danach in einem nächsten Schritt arithmetische Operationen sowie den Shor-Algorithmus zu implementieren. Die grösste Schwierigkeit, mit der ich mich auf diesem Weg konfrontiert sah, war es, die Quantenalgorithmen zu debuggen.
In der Tat dauerte es jeweils eine Weile, bis mein Code funktionierte. Dieser war anfangs fehlerhaft und diese Fehler zu finden war eine Herausforderung. Dies lag unter anderem daran, dass Qubits komplizierte Konstrukte sind, deren momentane und geplante Zustände schwierig zu vergleichen waren. Somit konnte es Stunden in Anspruch nehmen, herauszufinden, wo sich der Fehler befand.

Rückblickend war es meiner Meinung nach nicht einfach, in dieses Gebiet einzusteigen. Meine Arbeit soll deshalb auch dazu dienen, zusammen mit der angeführten Fachliteratur anderen den Einstieg zu vereinfachen. Dazu werden allerdings Kentnisse der linearen Algebra vorausgesetzt, da diese unabdingbar sind, um die mathematische Struktur hinter den Quantensystemen zu verstehen. Zum Einstieg in die lineare Algebra habe ich das Buch "Lineare Algebra" von Gerd Fischer [6] studiert und empfehle dieses Buch, um sich die Grundlagen zu erarbeiten.

Ich habe mich einerseits damit auseinandergesetzt, wie arithmetische Operationen auf Quantencomputern programmiert werden können, andererseits, wie man damit Zahlen faktorisieren

kann. Was ich gelernt habe, habe ich in Code umgesetzt. Ich habe zwei Programmbibliotheken implementiert, die ich in Abschnitt 1.2 genauer beschreibe und in welchen man alle in dieser Arbeit betrachteten Operationen als Code finden kann. Zudem habe ich bei der Implementierung gezielt darauf geachtet, dass diese Bibliotheken auch in späteren Projekten gut verwendet werden können und ich habe vor, diese Bibliotheken in Zukunft kontinuierlich zu erweitern und Implementierungen anderer Quantenalgorithmen hinzuzufügen, so dass sie aktuell bleiben.

1.2 Die QInteger- und die QAlgorithm-Library und eine Implementierung des Faktorisierungsalgorithmus

Das Produkt dieser Maturitätsarbeit sind zwei Programmbibliotheken, welche Algorithmen für Quantencomputer bereitstellen. Diese Programmbibliotheken sind in der Programmiersprache Q# geschrieben, einer Quantenprogrammiersprache entwickelt von Microsoft. Das Quantum Development Kit (QDK) von Microsoft beinhaltet dabei nicht nur die Sprache Q#, sondern auch einen Simulator, auf welchem man den Code ausführen kann. Gleichzeitig ist es das Ziel der Sprache, dass man sie auf echten Quantencomputern einsetzen kann, sobald diese weit genug entwickelt sein werden.

Die beiden Bibliotheken sollen mathematische Operationen auf Quantencomputern vereinfachen. Hier ein kurzer Überblick über ihre Funktionen:

- Die *QInteger-Library* definiert mit dem Typ "QInt" die grundlegende Datenstruktur "Zahl" auf einem Quantencomputer. Zudem stellt sie arithmetische Operationen und nützliche Funktionen für den Typ "QInt" bereit.
- In der *QAlgorithms-Library* sind Algorithmen für Quantencomputer implementiert, darunter auch der quantenbasierte Teil von Shors Algorithmus zur Faktorisierung von Zahlen, welcher zweifelsohne eine der nennenswertesten Errungenschaften auf dem Gebiet der Quantencomputer ist.

Die beiden Bibliotheken stellen nützliche Funktionen für Quantencomputer bereit, welche auch in anderen Projekten weiterverwendet werden können. Den Code zu diesen Programmbibliotheken kann man auf GitHub finden, wo er in Zukunft einige Aktualisierungen erfahren wird. Auf GitHub ist er unter folgendem Link verfügbar:

https://github.com/johutha/QInteger-QAlgorithms

Ich habe Q# gewählt, da es aktuell eine der populärsten Quantenprogrammiersprachen ist, über eine gute Dokumentation verfügt, gut unterstützt und regelmässig aktualisiert wird. Zudem kann der Compiler automatisch zu einer Quantenoperation ihre adjungierte oder kontrollierte Version generieren[1], was den Code kürzer und übersichtlicher macht.

Neben diesen beiden Bibliotheken findet man im GitHub-Repository weitere Projekte. In einem dieser Projekte ist der "Factorizer" implementiert, welcher zusammen mit den Quanten-Programmbibliotheken eine vollständige Implementierung von Shors Algorithmus bildet. Weiter

[1] Siehe auch 2.2.5

sind im Projekt noch andere Faktorisierungsalgorithmen programmiert, sodass man diese miteinander vergleichen kann.

Im Weiteren befinden sich im Repository ein Projekt für eine einfache Konsolen-Applikation, welche den “Factorizer“ benützt, sodass man diesen als Proof of Concept ausprobieren kann, und ein Projekt mit einem Zeitmesser, welcher misst, wie lange der “Factorizer “ benötigt.

Zudem befinden sich im Repository drei weitere Projekte, in welchen Unit-Tests implementiert sind. Unit-Tests testen die verschiedenen Komponenten einzeln, wodurch man einfacher Implementierungsfehler lokalisieren und Bugs finden kann.

Kapitel 2

Grundlagen

2.1 Lineare Algebra

Um mit Quantencomputern arbeiten zu können, braucht es Kenntnisse der linearen Algebra. Operationen, die man auf Quantencomputern implementiert, sind lineare Operationen auf Qubits. Ich werde an dieser Stelle eine kurze Zusammenfassung der nötigen Grundlagen geben, welche erforderlich sind, um die Quanten-Grundlagen und Shors Algorithmus zu verstehen. Dies wird jedoch nur eine kurze Zusammenfassung und keine Einführung in das Gebiet sein. Für eine Einführung verweise ich auf [6].

Vor der Zusammenfassung möchte ich kurz die in der Quantenmechanik gebräuchliche Notation für lineare Algebra einführen. Diese nennt sich Dirac- oder Bra-Ket-Notation.

2.1.1 Schreibweise

Einen Vektor schreibt man in der Quantenmechanik als $|\varphi\rangle$. Diese besondere Art von Klammern wird als *ket* bezeichnet. Für einen Vektor $|\varphi\rangle$ wird der dazugehörige duale Vektor als $\langle\varphi|$ bezeichnet. Diese zweite Klammer nennt man *bra*, sodass die beiden Klammern zusammen ein Bra-Ket bildet, was vom englischen Wort bracket herrührt. Somit lässt sich das Skalarprodukt von $|\psi\rangle$ und $|\varphi\rangle$ als $\langle\psi|\varphi\rangle$ darstellen.

2.1.2 Vektorräume

Um Quantensysteme mathematisch beschreiben zu können, brauchen wir Vektorräume. Genauer benötigen wir endlichdimensionale Vektorräume über $\mathbb{C}$, zusammen mit einem Skalarprodukt. Diese Vektorräume werden auch unitäre Vektorräume[1] genannt. Sie sind isomorph zu $\mathbb{C}^n$.

Sei V ein solcher Vektorraum.

Wir nennen eine Menge von Vektoren in V ein *Erzeugendensystem*, falls jeder Vektor in V als eine Linearkombination der Vektoren in jener Menge geschrieben werden kann.

Eine Menge von Vektoren $|v_0\rangle, |v_1\rangle, ... |v_{k-1}\rangle$ ist *linear unabhängig*, falls aus der Gleichung $a_0|v_0\rangle + a_1|v_1\rangle + ... + a_{k-1}|v_{k-1}\rangle = 0$ folgt, dass $a_0 = a_1 = ... = a_{k-1} = 0$ gilt. Dabei sind die Koeffizienten $a_0, a_1, ..., a_{n-1}$ komplexe Zahlen. Diese Aussage ist äquivalent zur Aussage, dass sich

[1] Im Allgemeinen braucht es sogenannte Hilberträume. Da man es im Zusammenhang mit Quantencomputern aber nur mit endlichdimensionalen Räumen zu tun hat, reichen unitäre Vektorräume

keiner der Vektoren $|v_i\rangle$ als Linearkombination der anderen Vektoren in der Menge darstellen lässt.

Eine *Basis* ist ein linear unabhängiges Erzeugendensystem.

2.1.3 Skalarprodukt

Die für die Beschreibung der Qubits benötigten unitären Vektorräumen sind mit einem Skalarprodukt ausgerüstet: Das heisst, es gibt eine Funktion $(\cdot,\cdot) : V \times V \to \mathbb{C}$ mit folgenden Eigenschaften:

1. Linear im zweiten Argument:
$$\sum_j \lambda_j \left(|\psi\rangle\,,\ |\varphi_j\rangle\right) = \left(|\psi\rangle\,,\ \sum_j \lambda_j\,|\varphi_j\rangle\right)$$
2. Hermitesch:
$$\left(|\psi\rangle\,,\ |\varphi\rangle\right) = \overline{\left(|\varphi\rangle\,,\ |\psi\rangle\right)}$$
3. Positiv definit:
$$\varphi \neq 0 \Rightarrow \left(|\varphi\rangle\,,\ |\varphi\rangle\right) > 0$$

Wie bereits angetönt, schreibt man dieses Produkt in der quantenmechanischen Notation als $\langle\psi|\varphi\rangle$. Ich habe oben jedoch die $(\cdot,\cdot)$-Schreibweise verwendet, da man damit die drei Bedingungen übersichtlicher darstellen kann.

Wir benutzen in dieser Arbeit das Standardskalarprodukt in $\mathbb{C}^n$ definiert durch:

$$\begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix} := \sum_{i=1}^{n} \overline{c_i} w_i$$

Es ist nicht schwierig zu sehen, dass das auf diese Weise definierte Skalarprodukt die oben genannten Bedingungen erfüllt.

Die *Norm* eines Vektors $|\varphi\rangle$ wird als $\| |\varphi\rangle \|$ geschrieben und ist definiert als $\| |\varphi\rangle \| = \sqrt{\langle\varphi|\varphi\rangle}$. Ein Vektor $|\varphi\rangle$ ist normiert, falls $\| |\varphi\rangle \| = 1$ gilt.

2.1.4 Lineare Operatoren

Sei v ein endlichdimensionaler, unitärer Vektorraum[2], seien $|v\rangle$ und $|w\rangle \in V$ und $\lambda \in \mathbb{C}$.

Definition: Ein *linerarer Operator*[3] L ist eine Funktion

$$\begin{array}{rccc} L: & V & \longrightarrow & V \\ & v & \mapsto & Lv \end{array}$$

welche die Bedingung der Linearität $L(\sum_i \lambda_i |v_i\rangle) = \sum_i \lambda_i L(|v_i\rangle)$ erfüllt.

[2]Die Definitionen dieses Abschnitts würden auch für allgemeinere Vektorräume funktionieren. Das ist jedoch für diese Arbeit nicht nötig.

[3]In dieser Arbeit wird der Begriff *linearer Operator* synonym zum Begriff *lineare Abbildung* benutzt

Lineare Operatoren lassen sich durch Matrizen darstellen, wobei die Spalten der Matrix die Bilder der Basisvektoren sind.

Für eine Matrix

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} \quad \text{mit Einträgen } a_{ij} \in \mathbb{C} \text{ schreiben wir kurz } A = (a_{ij}) \text{ und definieren } \ldots$$

die *konjugierte* Matrix von A: $\overline{A} := (\overline{a}_{ij})$

die *transponierte* Matrix von A: $A^T := (a_{ji})$

die *adjungierte* Matrix von A: $A^\dagger := \overline{A}^T$

Definition: Ein *unitärer Operator* U ist ein linearer Operator, welcher die folgende Bedingung erfüllt:

$$U^\dagger U = I$$

Aufgrund dieser Bedingung erhalten unitäre Operationen das Skalarprodukt der Vektoren:

$$(U|u\rangle, U|v\rangle) = \langle u| U^\dagger U |v\rangle = \langle u|v\rangle$$

Insbesondere verändert sich auch das Skalarprodukt eines Vektors mit sich selbst nicht, weshalb auch die Norm eines Vektors unter unitären Operatoren erhalten bleibt.

Zustände eines Quantensystems können als Vektoren eines unitären Vektorraumes V beschrieben werden. Zustandsänderungen werden durch unitäre Operationen beschrieben.

2.1.5 Eigenwerte und Eigenvektoren

Ein Eigenvektor $|\varphi\rangle$ zu einem linearen Operator U ist ein Vektor, sodass $U|\varphi\rangle = \lambda|\varphi\rangle$ gilt, wobei λ als der dazugehörige Eigenwert bezeichnet wird und eine komplexe Zahl ist. Bei einem unitären Operator müssen alle Eigenwerte einen Betrag von 1 haben, sonst würde sich die Norm des Vektors verändern. Der grosse Vorteil von Eigenvektoren ist, dass sie sich bis auf einen skalaren Faktor nicht verändern, wenn der dazugehörige Operator auf sie angewendet wird.

2.1.6 Das Tensorprodukt

Das Tensorprodukt von Vektorräumen

Das *Tensorprodukt* von zwei Vektorräumen U und V ist ein Vektorraum $U \otimes V$, welcher durch die Produkte $u \otimes v$ mit $u \in U$ und $v \in V$ erzeugt wird, so dass

$$(\lambda u) \otimes (\mu v) = \lambda\mu(u \otimes v) \,,$$

$$(u_1 + u_2) \otimes v = u_1 \otimes v + u_2 \otimes v$$

$$\text{und } u \otimes (v_1 + v_2) = u \otimes v_1 + u \otimes v_2 \,.$$

(Für alle u, u_1, $u_2 \in U$, v, v_1, $v_2 \in V$ und $\lambda, \mu \in \mathbb{C}$)

Dass es ein solcher Vektorraum $U \otimes V$ wirklich existiert und wie man ihn definiert, kann man in [6] im Kapitel 6.3 nachlesen, ebenso wie die folgende Tatsache:

Sind die Vektoren u_1, u_2, …, u_n eine Basis von U und die Vektoren v_1, v_2, …, v_m eine Basis von V, so bilden die Vektoren $u_i \otimes v_j$ eine Basis von $U \otimes V$.
Dies zeigt, dass $U \otimes V$ ein $n \cdot m$-dimensionaler Vektorraum ist.

Für Tensorprodukte von mehreren Vektorräumen V_1, V_2, …, V_n verwenden wir die folgende Schreibweise:

$$V_1 \otimes V_2 \otimes \ldots \otimes V_n =: \bigotimes_{k=1}^{n} V_k$$

Und für das n-fache Tensorprodukt eines Vektorraumes V mit sich selbst schreiben wir kurz

$$\bigotimes_{k=1}^{n} V =: V^{\otimes n}$$

In dieser Arbeit werden hauptsächlich n-fache Tensorprodukte des Vektorraumes $\mathbb{C}^2$ mit sich selbst benätigt. Wir betrachten deshalb diese Situation noch etwas genauer.
Die Standardbasis des $\mathbb{C}^2$ wird in der Dirac-Schreibweise mit $|0\rangle := \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ und $|1\rangle := \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ bezeichnet.

Entsprechend bilden die folgenden Vektoren eine Basis für das n-fache Tensorprodukt $\mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \ldots \otimes \mathbb{C}^2$:

$$\begin{aligned}
|0\rangle \otimes |0\rangle \otimes \ldots \otimes |0\rangle \otimes |0\rangle &=: |00\ldots00\rangle =: |0\rangle \\
|0\rangle \otimes |0\rangle \otimes \ldots \otimes |0\rangle \otimes |1\rangle &=: |00\ldots01\rangle =: |1\rangle \\
|0\rangle \otimes |0\rangle \otimes \ldots \otimes |1\rangle \otimes |0\rangle &=: |00\ldots10\rangle =: |2\rangle \\
|0\rangle \otimes |0\rangle \otimes \ldots \otimes |1\rangle \otimes |1\rangle &=: |00\ldots11\rangle =: |3\rangle \\
&\vdots \\
|1\rangle \otimes |1\rangle \otimes \ldots \otimes |1\rangle \otimes |1\rangle &=: |11\ldots11\rangle =: |2^n - 1\rangle
\end{aligned}$$

Bei der Abkürzung ganz rechts werden die Sequenzen von Nullen und Einsen als Zahlen im Binärsystem interpertiert.
Aus der obigen Auflistung der Basisvektoren geht auch hervor, dass das n-fache Tensorprodukt $\mathbb{C}^2 \otimes \mathbb{C}^2 \otimes \ldots \otimes \mathbb{C}^2$ zum Vektorraum $\mathbb{C}^{2^n}$ isomorph ist. Je nachdem was gerade zweckdienlich ist, werden wir die eine oder die andere Darstellung wählen.

Das Tensorprodukt von linearen Operatoren

Seien U, U', V und V' endlichdimensionale Vektorräume und $A\colon U \to U'$, $B\colon V \to V'$ lineare Operatoren, so wird durch

$$\begin{array}{rccc} A \otimes B : & U \otimes V & \longrightarrow & U' \otimes V' \\ & u \otimes v & \mapsto & A(u) \otimes B(v) \end{array}$$

ein linearer Operator von $U \otimes V$ nach $U' \otimes V'$ definiert.

Sei $u_1, u_2, \ldots, u_m$ eine Basis von U, $u'_1, u'_2, \ldots, u'_n$ eine Basis von U' sowie $v_1, \ldots, v_s$ und $v'_1, \ldots v'_r$ Basen von V bzw. von V'.

Wenn wir A bezüglich dieser Basen durch eine $n \times m$-Matrix (a_{ij}) darstellen und B durch eine $r \times s$-Matrix (b_{ij}), so kann man die Matrix des Operators $A \otimes B$ bezüglich der Basen

$$u_1 \otimes v_1,\ u_1 \otimes v_2,\ \ldots,\ u_1 \otimes v_s,\ u_2 \otimes v_1,\ u_2 \otimes v_2,\ \ldots,\ u_m \otimes v_s \quad \text{von} \quad U \otimes V$$

und

$$u'_1 \otimes v'_1,\ u'_1 \otimes v'_2,\ \ldots,\ u'_1 \otimes v'_r,\ u'_2 \otimes v'_1,\ u'_2 \otimes v'_2,\ \ldots,\ u'_n \otimes v'_r \quad \text{von} \quad U' \otimes V'$$

auf die folgende Weise bestimmen:

$$\begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nm} \end{bmatrix} \otimes \begin{bmatrix} b_{11} & \cdots & b_{1s} \\ \vdots & \ddots & \vdots \\ b_{r1} & \cdots & b_{rs} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}\begin{bmatrix} b_{11} & \cdots & b_{1s} \\ \vdots & \ddots & \vdots \\ b_{r1} & \cdots & b_{rs} \end{bmatrix} & \cdots & a_{1m}\begin{bmatrix} b_{11} & \cdots & b_{1s} \\ \vdots & \ddots & \vdots \\ b_{r1} & \cdots & b_{rs} \end{bmatrix} \\ \vdots & \ddots & \vdots \\ a_{n1}\begin{bmatrix} b_{11} & \cdots & b_{1s} \\ \vdots & \ddots & \vdots \\ b_{r1} & \cdots & b_{rs} \end{bmatrix} & \cdots & a_{nm}\begin{bmatrix} b_{11} & \cdots & b_{1s} \\ \vdots & \ddots & \vdots \\ b_{r1} & \cdots & b_{rs} \end{bmatrix} \end{bmatrix}$$

$$= \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & \cdots & a_{11}b_{1s} & a_{12}b_{11} & \cdots & a_{1m}b_{1s} \\ a_{11}b_{21} & a_{11}b_{22} & \cdots & a_{11}b_{2s} & a_{12}b_{21} & \cdots & a_{1m}b_{2s} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{11}b_{r1} & a_{11}b_{r2} & \cdots & a_{11}b_{rs} & a_{12}b_{r1} & \cdots & a_{1m}b_{rs} \\ a_{21}b_{11} & a_{21}b_{12} & \cdots & a_{21}b_{1s} & a_{22}b_{11} & \cdots & a_{2m}b_{1s} \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ a_{n1}b_{r1} & a_{n1}b_{r2} & \cdots & a_{n1}b_{rs} & a_{n2}b_{r1} & \cdots & a_{nm}b_{rs} \end{bmatrix}$$

2.2 Quantensysteme

Die Quantensysteme, die wir im Rahmen dieser Arbeit betrachten, sind rein mathematische Systeme, die nicht an eine fixe physikalische Realisierung gebunden sind. In der Tat gibt es verschiedene Möglichkeiten, solche Systeme physikalisch umzusetzen, wobei jede Variante eigene Vor- und Nachteile hat.

2.2.1 Qubits

Ein klassisches Bit hat genau zwei Zustände - 0 und 1 - und befindet sich immer in genau einem dieser beiden Zustände. Kann man zwei solche Zustände in einem quantenmechanischen System gezielt herbeiführen, treten auch die Gesetze der Quantenphysik in Kraft. Ein Beispiel einer physikalischen Realisierung, das bei der Vorstellung helfen kann, wäre ein Atom im Grundzustand und einem angeregten Zustand, wobei der Grundzustand einer klassischen 0 und der angeregte Zustand einer klassischen 1 entsprechen könnte. Gemäss den Gesetzen der Quantenphysik kann

sich das Atom aber auch in einer Überlagerung der beiden Zustände befinden. Dies nennt man eine Superposition. Ausgehend von diesem Beispiel können wir uns nun die genaue Definition eines Qubits betrachten:

Definition: Ein *Qubit* ist ein Quantensystem mit den beiden Basiszuständen $|0\rangle$ und $|1\rangle$. Es kann alle Zustände $\alpha|0\rangle + \beta|1\rangle$ annehmen, sodass $|\alpha|^2 + |\beta|^2 = 1$ gilt. Die komplexen Zahlen α und β nennt man Amplituden.

Ein Zustand des Qubits, welcher durch die Parameter α und β definiert ist, entspricht einem normierten Vektor im Vektorraum $\mathbb{C}^2$.[4]

Wenn wir zu unserem Beispiel von vorhin zurückkehren, würde der Zustand $|0\rangle$ dem Grundzustand und der Zustand $|1\rangle$ dem angeregten Zustand entsprechen.

Qubits können wir messen[5]. Betrachten wir ein Qubit im Zustand $\alpha|0\rangle + \beta|1\rangle$, so beträgt die Wahrscheinlichkeit, dass wir $|0\rangle$ messen, $|\alpha|^2$ und die Wahrscheinlichkeit, $|1\rangle$ zu messen $|\beta|^2$. Die Bedingung, dass $|\alpha|^2+|\beta|^2 = 1$ gelten muss (dass der Vektor normiert sein muss), führt dazu, dass sich die Wahrscheinlichkeiten auf 1 summieren. Nach der Messung kollabiert das Quantensystem in den gemessenen Zustand. Messen wir also $|1\rangle$, befindet sich das Qubit nachher immer im Zustand $|1\rangle$, unabhängig davon, was α und β vorher waren.

Der Zustand eines Multiqubitsystems mit n Qubits wird durch einen normierten Vektor im Raum $\otimes_{k=1}^{n}\mathbb{C}^2$ beschrieben[6]. Wie in 2.1.6 erklärt wurde, hat dieser Raum die Basisvektoren $|0\ldots00\rangle, |0\ldots01\rangle, \ldots, |1\ldots11\rangle$. Diese entsprechen dabei den einzelnen Kombinationen der $|0\rangle$s und $|1\rangle$s der einzelnen Qubits. Zum Beispiel hat der Vektorraum zu einem Quantensystem mit 2 Qubits die vier Basiszustände $|00\rangle, |01\rangle, |10\rangle$ und $|11\rangle$ und der Zustand $|101\rangle$ in einem System mit 3 Qubits entspricht dem Zustand, in welchem das erste Qubit im Zustand $|1\rangle$, das zweite im Zustand $|0\rangle$ und das dritte im Zustand $|1\rangle$ ist.

Ausserdem können wir den Vektorraum $\otimes_{k=1}^{n}\mathbb{C}^2$ auch mit dem Raum $\mathbb{C}^{2^n}$ und die Basisvektoren in der Form $|0\rangle, |1\rangle, \ldots, |2^n-1\rangle$ notieren.

Auch summmieren sich die Wahrscheinlichkeiten wieder auf 1, da der Vektor normiert ist. Man schreibt die Zustände oft als eine Linearkombiantion der Basiszustände: $\alpha_{00}|00\rangle+\alpha_{01}|01\rangle+\alpha_{10}|10\rangle+\alpha_{11}|11\rangle = \alpha_0|0\rangle+\alpha_1|1\rangle+\alpha_2|2\rangle+\alpha_3|3\rangle$.

Um den Zustand des Multiqubitsystems beschreiben zu können, wenn wir im Besitz der Zustände der einzelnen Qubits sind, können wir die Vektoren der Zustände der einzelnen Qubits mit Hilfe des Tensorprodukts zusammenmultiplizieren. Zudem kann man mit dem Tensorprodukt auch Operationen zusammenmultiplizieren: Wenn wir n Qubits haben und auf jedes Qubit eine Operation anwenden (dies kann auch die Identität sein, falls wir das Qubit unverändert lassen wollen) und dann die Matrizen dieser Operationen multiplizieren, bekommen wir die Matrix, welche der Operation entspricht, die alle unsere ausgewählten Operationen ausführt. Wichtig ist aber noch anzumerken, dass das Tensorprodukt nicht kommutativ ist und die Reihenfolge der Faktoren somit wichtig ist.

[4]Dass sich der Zustand eines quantenmechanischen Systems als Vektor beschreiben lässt, garantiert uns das erste Postulat der Quantenmechanik. Da diese Postulate den Rahmen dieser Arbeit übersteigen, verweise ich für eine genauere Betrachtung dieser Postulate auf [13], Kapitel 2.2

[5]Auch die Regeln für die Messungen basieren auf einem Postulat der Quantenmechanik, nämlich auf dem dritten Postulat.

[6]Auch das folgt aus einem Postulat der Quantenmechanik. Für genauere Ausführungen sei auf [13], Kapitel 2.2.8 verwiesen

Falls wir nur einzelne Qubits messen, kollabiert das System in die restlichen noch möglichen Zustände. Nehmen wir als Beispiel ein 2-Qubit-System im Zustand $\frac{1}{\sqrt{6}}|00\rangle+\frac{1}{\sqrt{2}}|01\rangle+\frac{1}{\sqrt{3}}|11\rangle$ und messen das erste Qubit. Die Wahrscheinlichkeit, dass wir dieses Qubit im Zustand $|1\rangle$ messen, liegt bei $|\alpha_{10}|^2+|\alpha_{11}|^2=\frac{1}{3}$. Falls wir diesen Zustand messen, kollabiert unser Quantensystem direkt in den Zustand $\frac{\alpha_{10}|10\rangle+\alpha_{11}|11\rangle}{\sqrt{|\alpha_{10}|^2+|\alpha_{11}|^2}}=|11\rangle$, wobei der Nenner dafür sorgt, dass der neue Quantenzustand wieder normiert ist. Die Wahrscheinlichkeit, bei der Messung des ersten Qubits $|0\rangle$ zu erhalten, liegt bei $|\alpha_{00}|^2+|\alpha_{01}|^2=\frac{2}{3}$. Der Zustand des Systems nach der Messung ist dann $\frac{\alpha_{00}|00\rangle+\alpha_{01}|01\rangle}{\sqrt{|\alpha_{00}|^2+|\alpha_{01}|^2}}=\sqrt{\frac{1}{4}}|00\rangle+\sqrt{\frac{3}{4}}|01\rangle$.

2.2.2 Die Blochkugel

Die Blochkugel dient der graphischen Darstellung des Zustands eines einzelnen Qubits. Zu diesem Zweck betrachten wir noch einmal ein einzelnes Qubit $\alpha|0\rangle+\beta|1\rangle$ mit $|\alpha|^2+|\beta|^2=1$. Diese Bedingung führt dazu, dass wir den Zustand als $e^{i\gamma}\left(\cos\frac{\psi}{2}|0\rangle+e^{i\varphi}\sin\frac{\psi}{2}|1\rangle\right)$ umschreiben können. Den Faktor $e^{i\gamma}$ können wir nicht beobachten, da er auf beide Koeffizienten wirkt[7]. Deshalb ist der Zustand durch die beiden Winkel ψ und φ definiert. Diese beiden Winkel kann man graphisch als einen Punkt auf einer Einheitskugel darstellen. Diese Darstellung wird die Bloch-Kugel genannt.

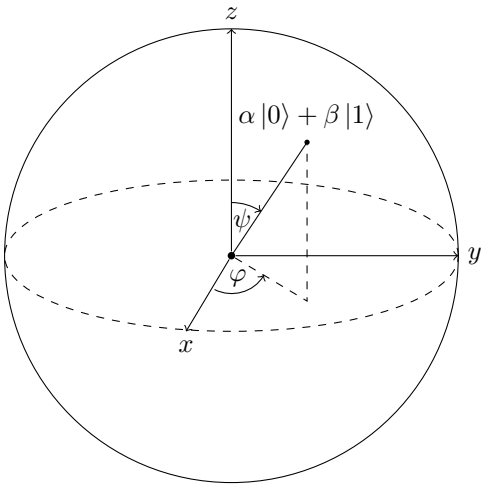

Abbildung 2.1: Beispiel einer Blochkugel

2.2.3 Operationen auf Qubits

Operationen, die man auf Qubits implementiert, sind lineare Operationen und lassen sich somit als Matrizen darstellen. Man kann den Vektor, welcher den Zustand beschreibt, mit der Matrix des Operators multiplizieren, um den Zustand nach der Operation zu erhalten. Dazu müssen die Operatoren die Norm des Quantenzustands bewahren, da die Zustände immer normiert sein müssen, und somit unitär sein. Dies hat zur Folge (für eine unitäre Matrix U gilt $UU^\dagger=I$),

[7] Man nennt dies eine globale Phase: Dieser Faktor hat Betrag 1 und verändert deshalb die Wahrscheinlichkeiten nicht. Da er sich zudem auf alle Zustände auswirkt, kann man ihn bei den Quantenoperatoren aufgrund der Linearität ausklammern. Somit bleibt er immer über alle Qubits bestehen und ist nicht beobachtbar.

dass ein inverser Operator existieren muss und deshalb alle Berechnungen reversibel sein müssen. Beispielsweise kann der Modulo-Operator nicht auf Quantencomputern implementiert werden, da man aus dem Ergebnis $x \equiv 2 \pmod 3$ die Eingabe $x \in \{2, 5, 8 \cdots\}$ nicht eindeutig wiederherstellen kann. Dies hat grosse Konsequenzen für die Berechnungen auf Quantencomputern.

Gleichzeitig stellt sich heraus, dass es verschiedene universelle Kombinationen von Operationen gibt, wobei universell in diesem Zusammenhang bedeutet, dass man jeden unitären Operator nur mit den ausgewählten Operatoren beliebig annähern kann. Die Konstruktion dazu kann man in [13], Seiten 188ff. nachlesen.

2.2.4 Wichtige Quantengatter

An dieser Stelle werden die wichtigsten Quantengatter eingeführt, die wir im Verlauf dieser Arbeit benötigen werden. Zuerst betrachten wir fünf grundlegende Gatter auf Qubits: Die drei Pauli-Matrizen X, Y und Z, das H-Gatter und das $CNOT$-Gatter.

- Das X-Gatter ist das Qubit-Equivalent zum NOT-Gatter eines elektronischen Schaltkreises. In Matrixform sieht der Operator so aus: $X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. Dieses Gatter dreht den Zustand des Qubits um π um die x-Achse in der Blochkugel.
- Das Y-Gatter wird durch die Matrix $Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$ dargestellt. Dieses Gatter entspricht einer Rotation von π um die y-Achse in der Blochkugel.
- Das Z-Gatter, als Matrix $Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$, dreht den Zustand um π um die z-Achse.
- Das Hadamard-Gatter, meist durch den Buchstaben H abgekürzt, ist der einfachste Weg, eine Superposition zu erzeugen. Mit der Matrix $H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ kann man die beiden Zustände $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ und $H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ erzeugen. Diese beiden Zustände kommen so häufig vor, dass man ihnen die Namen
$$|+\rangle = \tfrac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad \text{und} \quad |-\rangle = \tfrac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$$
gegeben hat.
- $CNOT$ steht als Abkürzung für "Controlled NOT". Dieses Gatter wirkt auf zwei Qubits und wendet ein NOT auf das zweite Qubit an, wenn das erste Qubit im Zustand $|1\rangle$ ist. Als Matrix sieht die Operation so aus:
$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$
.

Diese Gatter gehören zu den wichtigsten Gattern im Bereich der Quantencomputer. Wir werden auf unserem Weg jedoch noch weitere Gatter antreffen. Eines davon möchte ich hier kurz definieren. Ich nenne es $\mathrm{Rot}(k)$, und in Matrix-Form sieht es so aus: $\mathrm{Rot}(k) = \begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{2i\pi}{2^k}} \end{bmatrix}$. Dieses

Gatter multipliziert den Koeffiztienten von $|1\rangle$ mit $e^{\frac{2i\pi}{2^k}}$ und wir werden es bei der Quanten-Fouriertransformation[8] und dessen Anwendungen antreffen.

Abschliessend möchte ich noch das SWAP-Gatter erwähnen, welches auf zwei Qubits wirkt. Dieses Gatter vertauscht die Zustände der beiden Qubits, indem es die Amplituden der Zustände $|01\rangle$ und $|10\rangle$ vertauscht. Es wird durch die folgende Matrix beschrieben:

$$SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Wenn wir das erste Qubit als a und das zweite als b bezeichnen, sehen wir auf der rechten Seite, dass wir das $SWAP(a, b)$ als Abfolge dreier $CNOT$-Gatter ($CNOT(a, b)$, dann $CNOT(b, a)$ und dann $CNOT(a, b)$) programmieren können.

Man kann die Wirkung des SWAP-Gatters auch folgendermassen erkennen:

$$\begin{aligned} &(\alpha|0\rangle + \beta|1\rangle) \otimes (\gamma|0\rangle + \delta|1\rangle) \\ &= \alpha\gamma|00\rangle + \alpha\delta|01\rangle + \beta\gamma|10\rangle + \beta\delta|11\rangle = \begin{pmatrix} \alpha\gamma \\ \alpha\delta \\ \beta\gamma \\ \beta\delta \end{pmatrix} \xrightarrow{SWAP} \begin{pmatrix} \alpha\gamma \\ \beta\gamma \\ \alpha\delta \\ \beta\delta \end{pmatrix} \\ &= \alpha\gamma|00\rangle + \beta\gamma|01\rangle + \alpha\delta|10\rangle + \beta\delta|11\rangle \\ &= (\gamma|0\rangle + \delta|1\rangle) \otimes (\alpha|0\rangle + \beta|1\rangle) \end{aligned}$$

2.2.5 Kontrollierte und adjungierte Operatoren

Wie wir bereits erfahren haben, müssen Operatoren, welche man auf Quantencomputern umsetzen kann, unitär und somit invertierbar sein. Dies setzt direkt die Existenz eines adjungierten Operators[9] zu einem Operator U voraus, welcher aus einem Zustand $|\varphi\rangle$ den Zustand $|\psi\rangle$ mit $U|\psi\rangle = |\varphi\rangle$ macht. Ein Schaltkreis zu diesem adjungierten Operator lässt sich generieren, indem man die Gatter des Schaltkreises für U rückwärts durchgeht, und dabei immer den adjungierten Operator des jeweiligen Gatters nimmt. Viele Quantenprogrammiersprachen können so auch die die adjungierte Version eines Quantenoperators aus dem Programmcode des ursprünglichen Operators generieren.

Auch lassen sich *kontrollierte* Versionen eines Operators bilden. Die kontrollierte Version eines Operators hat zusätzlich zu den Qubits, auf welchen die Operation ausgeführt werden soll, eine Menge an sogenannten Kontrollqubits als Argumente. Diese Version der Operation wird nur in denjenigen Zuständen ausgeführt, in welchen alle Kontrollqubits im Zustand $|1\rangle$ sind. Als Beispiel nehmen wir eine Operation, welche die Bitrepräsentation zyklisch um ein Bit nach rechts verschiebt (also aus dem Zustand $|01101\rangle$ den Zustand $|10110\rangle$ macht). Betrachten wir nun den Zustand $|0\rangle|11101\rangle + |1\rangle|10110\rangle + |0\rangle|01000\rangle$. Das erste Qubit habe ich hier getrennt dargestellt, da ich es für diese Operation als Kontrollqubit verwenden möchte. Wenn wir nun

[8] Siehe Kapitel 3

[9] Der inverse Operator eines unitären Operators U ist der adjungierte Operator, da $U^\dagger U = I$.

die Operation mit dem ersten Qubit als Kontrollqubit anwenden, erhalten wir den Zustand $|0\rangle|11101\rangle + |1\rangle|01011\rangle + |0\rangle|01000\rangle$. Sei U ein Operator, welcher auf Quantencomputern programmierbar ist. Dann schreiben wir die durch das Qubit c kontrollierte Version von U als U^c. In den Fällen, in denen wir mit U^k die k mal wiederholte Anwendung des Operators U meinen, werde ich darauf hinweisen.

Ein Qubit darf aber nie gleichzeitig ein Kontrollqubit und eines jener Qubits sein, auf die der Operator angewendet wird. Um dies zu sehen, betrachten wir das $CNOT$, welches die kontrollierte Variante des X-Operators ist. Wenn wir den Operator auf ein Qubit anwenden, welches gleichzeitig das Kontrollqubit ist, wird dieses Qubit nach der Anwendung immer im Zustand $|0\rangle$ sein. Somit ist dieser Operator nicht mehr invertierbar. Wenn die oben genannte Bedingung aber eingehalten wird, sind kontrollierte Operatoren invertierbar, da man aus den Kontrollqubits, welche während der Operation nicht verändert werden, ablesen kann, ob die Operation durchgeführt wurde.

Auch diese Version wird von den meisten Quantenprogrammiersprachen automatisch generiert, da man den Schaltkreis der kontrollierten Version einfach herstellen kann, indem man vom Schaltkreis des ursprünglichen Operators bei jedem Gatter die kontrollierte Version nimmt.

Zudem lassen sich die beiden oben genannten Varianten auch kombinieren, um eine kontrollierte, adjungierte Version eines Operators zu generieren.

Kapitel 3

Arithmetische Operationen auf Qubits ausführen: Die QInteger Library

3.1 Zahlen in Qubits speichern - Der QInt-Typ

Da ich davon ausgehe, dass in absehbarer Zeit die Anzahl Qubits zwar wachsen, aber nicht so schnell ansteigen wird, dass man schon bald mehrere grössere Qubit-Einheiten speichern kann, habe ich mich entschieden, in meiner Implementierung auf eine einheitliche Grösse zu verzichten und für den Moment eine variable Anzahl von Qubits zur Speicherung einer Zahl zu benutzen. Aus diesem Grund besteht der QInt-Typ aus einer klassischen Zahl, der Anzahl Qubits, und einem Array von Qubits, welcher die in den Qubits zu speichernde Zahl speichert.

```
// Definition of the QInt type with variable size. QInts
// are represented in little-endian.
newtype QInt = (Size : Int, Number : Qubit[]);
```

Dies ist die Definition des Typs QInt in meinem Code: Wir definieren den neuen Typ "QInt" als eine Kombination einer Zahl, genannt *Size*, und einem Array von Qubits, genannt *Number*.

Im weiteren Verlauf dieser Arbeit werde ich Zahlen, die auf klassische Weise gespeichert sind und somit keine Superpositionen erlauben, als "klassische Zahlen" bezeichnen und Zahlen, welche in Quantenregistern gespeichert werden, als "Quantenzahlen" oder "QInts".

3.2 Die Quanten-Fouriertransformation und die Fourier-Basis

3.2.1 Die Quantenfouriertransformation

Sei $|x\rangle$ ein Basisvektor. Damit können wie die Quantenfouriertransformation, kurz QFT, wie folgt definieren:

$$QFT\,|x\rangle = \frac{1}{\sqrt{2^n}} \sum_{j=0}^{2^n-1} e^{2i\pi \frac{xj}{2^n}} |j\rangle$$

Diesen Zustand kann man folgendermassen faktorisieren, was ich nachher beweisen werde:

$$\left(|0\rangle + e^{2i\pi\frac{x}{2^1}}|1\rangle\right) \otimes \left(|0\rangle + e^{2i\pi\frac{x}{2^2}}|1\rangle\right) \otimes \cdots \otimes \left(|0\rangle + e^{2i\pi\frac{x}{2^n}}|1\rangle\right) = \bigotimes_{j=0}^{n-1} \left(|0\rangle + e^{2i\pi\frac{x}{2^{1+j}}}|1\rangle\right)$$

Diese Faktorisierung werde ich nun beweisen. Sei die Darstellung von $j \in \{0, 1, \ldots, 2^n - 1\}$ im Binärsystem:

$$j = 2^{n-1} \cdot j_{n-1} + \ldots + 2^1 \cdot j_1 + 2^0 \cdot j_0 = \sum_{s=0}^{n-1} 2^s j_s$$

Wir können nun folgende Umformung durchführen:

$$\bigotimes_{s=0}^{n-1} \left(|0\rangle + e^{2i\pi\frac{x}{2^{(1+s)}}}|1\rangle\right) = \bigotimes_{s=0}^{n-1} \left(e^{2i\pi\frac{0\cdot x}{2^{(1+s)}}}|0\rangle + e^{2i\pi\frac{1\cdot x}{2^{(1+s)}}}|1\rangle\right)$$

$$= \sum_{(j_{n-1},\ldots,j_0)\in\{0,1\}^n} \bigotimes_{s=0}^{n-1} e^{2i\pi\frac{x\cdot j_{n-1-s}}{2^{(1+s)}}}|j_{n-1-s}\rangle = \sum_{(j_{n-1},\ldots,j_0)\in\{0,1\}^n} \left(\prod_{s=0}^{n-1} e^{2i\pi\frac{x\cdot j_{n-1-s}}{2^{(1+s)}}}\right)|j_{n-1}j_{n-2}\cdots j_0\rangle$$

$$= \sum_{j=0}^{2^n-1} \left(\prod_{s=0}^{n-1} e^{2i\pi\frac{x\cdot j_{n-1-s}}{2^{(1+s)}}}\right)|j\rangle = \sum_{j=0}^{2^n-1} \left(e^{2i\pi\frac{x\sum_{s=0}^{n-1} j_{n-1-s}\cdot 2^{n-1-s}}{2^n}}\right)|j\rangle = \sum_{j=0}^{2^n-1} e^{2i\pi\frac{xj}{2^n}}|j\rangle$$

Schauen wir uns nun die Produktdarstellung der Fouriertransformierten von $|x\rangle$ genauer an. Dazu benutzen wir die folgende Notation, um x im Binärsystem darzustellen:
$x = 2^{n-1} \cdot x_{n-1} + \cdots + 2^1 \cdot x_1 + 2^0 \cdot x_0 =: (x_{n-1} \ldots x_1 x_0)$ für $x_s \in \{0, 1\}$
und für $k \in \mathbb{N}, k < n$ schreiben wir:
$\frac{x}{2^k} =: (x_{n-1} \ldots x_k . x_{k-1} \ldots x_1 x_0)$ und $\frac{x}{2^n} =: (0.x_{n-1} \ldots x_1 x_0)$
Diese Schreibweise ist das Äquivalent zu den Dezimalzahlen mit Nachkommastellen im binären Zahlensystem.

Mit dieser Notation erhält man:

$$\left(|0\rangle + e^{2i\pi\frac{x}{2^1}}\right) \otimes \left(|0\rangle + e^{2i\pi\frac{x}{2^2}}\right) \otimes \cdots \otimes \left(|0\rangle + e^{2i\pi\frac{x}{2^{n-1}}}\right) \otimes \left(|0\rangle + e^{2i\pi\frac{x}{2^n}}\right)$$

$$= \left(|0\rangle + e^{2i\pi\frac{(x_{n-1}\ldots x_0)}{2^1}}\right) \otimes \left(|0\rangle + e^{2i\pi\frac{(x_{n-1}\ldots x_0)}{2^2}}\right) \otimes \cdots \otimes \left(|0\rangle + e^{2i\pi\frac{(x_{n-1}\ldots x_0)}{2^n}}\right)$$

$$= \left(|0\rangle + e^{2i\pi\cdot(x_{n-1}\,\ldots\,x_1+0\,.\,x_0)}\right) \otimes \left(|0\rangle + e^{2i\pi\cdot(x_{n-1}\,\ldots\,x_2+0\,.\,x_1x_0)}\right) \otimes \ldots$$

$$\ldots \otimes \left(|0\rangle + e^{2i\pi\cdot(x_{n-1}+0\,.\,x_{n-2}\,\ldots\,x_0)}\right) \otimes \left(|0\rangle + e^{2i\pi\cdot(0\,.\,x_{n-1}\,\ldots\,x_0)}\right)$$

$$= \left(|0\rangle + e^{2i\pi\cdot(0\,.\,x_0)}\right) \otimes \left(|0\rangle + e^{2i\pi\cdot(0\,.\,x_1x_0)}\right) \otimes \ldots$$

$$\ldots \otimes \left(|0\rangle + e^{2i\pi\cdot(0\,.\,x_{n-2}\,\ldots\,x_0)}\right) \otimes \left(|0\rangle + e^{2i\pi\cdot(0\,.\,x_{n-1}\,\ldots\,x_0)}\right)$$

Im letzten Schritt wurde benutzt, dass für $z \in \mathbb{Z}$ und $\alpha \in \mathbb{R}$ gilt:

$$e^{2i\pi\cdot(z+\alpha)} = \underbrace{e^{2i\pi z}}_{1} \cdot e^{2i\pi\alpha} = e^{2i\pi\alpha}$$

Unter der Darstellung der Zahl $x \in \mathbb{N}_0$ in der *binären Basis* verstehen wir die Darstellung von x im Binärsystem, also $x = 2^{n-1} \cdot x_{n-1} + \cdots + 2^1 \cdot x_1 + 2^0 \cdot x_0$, beziehungsweise die Darstellung durch den Zustand $|x_{n-1} \ldots x_0\rangle$ (für $x_k \in \{0, 1\}$).

Unter der Darstellung von x in der *Fourierbasis* verstehen wir die Produktdarstellung der Fouriertransformierten von $|x\rangle$, also

$$\frac{1}{\sqrt{2^n}} \left(|0\rangle + e^{2i\pi(0.x_0)}\right) \otimes \left(|0\rangle + e^{2i\pi(0.x_1x_0)}\right) \otimes \cdots \otimes \left(|0\rangle + e^{2i\pi(0.x_{n-1}\ldots x_0)}\right)$$

Die im obigen Produkt vorkommenden Zustände nennen wir in diesem Zusammenhang die *Fourierbasis*[1]. Hier ist darauf hinzuweisen, dass die Qubits von rechts nach links angeordnet (das Qubit ganz rechts ist das Qubit 0) und auch so in meinem Programm im Array gespeichert sind, da in der binären Basis das Qubit ganz rechts jenes mit dem Wert $2^0 \cdot x_0$ ist.

Dass wir die Darstellung einer Zahl in der Fourierbasis als Tensorprodukt der einzelnen Qubits schreiben können, bedeutet, dass diese Qubits unabhängig voneinander, also nicht verschränkt sind. Dies bedeutet für uns, dass wenn wir mit einer Zahl in der Fourierbasis rechnen wollen, wir die Qubits einzeln bearbeiten können. Zudem sehen wir, wenn wir den Zustand eines einzelnen Qubits $\frac{1}{\sqrt{2}} \left(|0\rangle + e^{2i\pi\frac{x}{2^{(1+s)}}}\right)$ betrachten, dass dieser Zustand einem Zeiger in der Blochkugel entspricht, welcher in der XY-Ebene liegt und dort um $2\pi\frac{x}{2^{s+1}}$ um den Mittelpunkt gedreht ist.

[1] Man beachte, dass die Fourierbasis keine Vektorraumbasis im Sinn von Kapitel 2.1.2 ist.

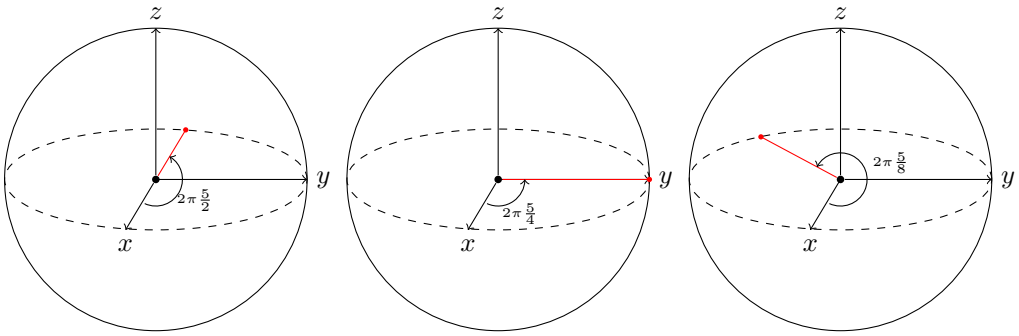

Abbildung 3.1: Die Darstellung der Zahl 5 in der Fourierbasis

Wir können nun feststellen, dass wenn wir die Zahl in der Fourierbasis verändern wollen, wir nur die Qubits um den richtigen Winkel θ drehen müssen, was der Multiplikation des Koeffizienten von $|1\rangle$ mit $e^{i\pi\theta}$ entspricht. Zudem können wir feststellen, dass wenn wir den Koeffizienten von $|1\rangle$ eines Qubits der Fourierbasis mit $e^{2i\pi\theta}$ multiplizieren, wir den Winkel θ mit Hilfe der inversen QFT bestimmen können. Diese Erkenntnis werden wir in Kapitel 4.3 benötigen.

3.2.2 Unitarität

Wir haben noch nicht gezeigt, dass die oben beschriebene Operation unitär ist. Zu diesem Zweck schauen wir uns die Matrix mit Einträgen an, die die Operation der QFT darstellt. Dafür sei $\omega_N^s = e^{2i\pi\frac{s}{N}}$. Dann können wir die QFT wie folgt darstellen: $[QFT] := (\omega_{2^n}^{jk})$ (wobei hier j die Reihe und k die Spalte ist, da i schon eine andere Bedeutung hat). Nun können wir die Bedingung $[QFT]^\dagger [QFT] = I$ überprüfen. Dazu berechnen wir den Eintrag jk von $[QFT]^\dagger [QFT]$:

$$\sum_{s=0}^{2^n-1} \frac{\overline{\omega_{2^n}^{js}}}{\sqrt{2^n}} \cdot \frac{\omega_{2^n}^{sk}}{\sqrt{2^n}} = \frac{1}{2^n} \sum_{s=0}^{2^n-1} \omega_{2^n}^{-js} \cdot \omega_{2^n}^{sk} = \frac{1}{2^n} \sum_{s=0}^{2^n-1} \omega_{2^n}^{s(k-j)}$$

Wir stellen nun fest: Falls $j = k$ gilt, dann ist $\omega_{2^n}^{s\cdot 0} = 1$ und somit

$$\frac{1}{2^n} \sum_{s=0}^{2^n-1} \omega_{2^n}^{s\cdot 0} = 1$$

Falls $j \neq k$, lässt die sich Summe wie folgt umformen, da $\omega_{2^n}^{0(k-j)}$, $\omega_{2^n}^{1(k-j)}$, ..., $\omega_{2^n}^{(2^n-1)(k-j)}$ eine geometrische Reihe ist:

$$\frac{1}{2^n} \sum_{s=0}^{2^n-1} \omega_{2^n}^{s(k-j)} = \frac{1}{2^n} \cdot \frac{\omega_{2^n}^{(k-j)2^n} - 1}{\omega_{2^n}^{(k-j)} - 1}$$

Gleichzeitig gilt $\omega_{2^n}^{(k-j)2^n} = 1$ und somit

$$\frac{1}{2^n} \cdot \frac{\omega_{2^n}^{(k-j)2^n} - 1}{\omega_{2^n}^{(k-j)} - 1} = 0$$

wobei der Nenner des Bruches nicht 0 ist, da $j \neq k$ gilt.

Wir sehen nun, dass in der resultierenden Matrix der Eintrag jk dann 1 ist, wenn $j = k$ gilt, und sonst 0. Dies ist aber genau die Definition von I. Daraus folgt, dass die QFT eine unitäre Operation und somit auf Quantencomputern implementierbar ist.

3.2.3 Implementierung

Gehen wir zurück zur Produktdarstellung der Fouriertransformierten von $|x\rangle$. Diese lässt sich folgendermassen mit Quantengattern realisieren:

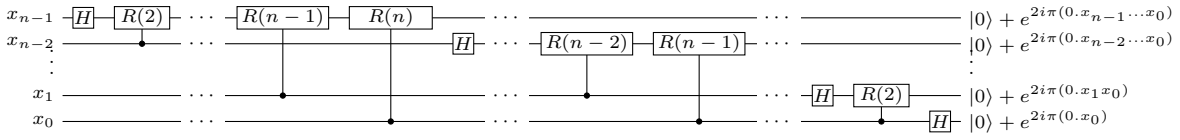

Abbildung 3.2: Schaltkreis der QFT. Die Rot-Gatter werden alle kontrolliert angewendet und die vertikalen Verbindungen zeigen, welches die Kontrollqubtis sind. Die Normalisierungsfaktoren von $\frac{1}{\sqrt{2}}$ wurden auf der rechten Seite weggelassen.

In der obigen Grafik habe ich aus Platzgründen den Namen $R(k)$ für das Rot(k)-Gatter verwendet. Um die Funktionsweise dieses Schaltkreises zu sehen, bemerken wir, dass $H|x_j\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + e^{2i\pi(0.x_j)}|1\rangle\right)$ gilt, da $e^{2i\pi\cdot 0} = 1$ und $e^{2i\pi\cdot\frac{1}{2}} = -1$ gilt. Wenden wir nun Rot$(2)^{x_{j-1}}$ an, bekommen wir:

$$\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2i\pi(0.x_j)}|1\rangle\right) \xrightarrow{\text{Rot}(2)^{x_{j-1}}} \frac{1}{\sqrt{2}}\left(|0\rangle + e^{2i\pi\frac{x_j}{2}+2i\pi\frac{x_{j-1}}{4}}|1\rangle\right) = \frac{1}{\sqrt{2}}\left(|0\rangle + e^{2i\pi(0.x_jx_{j-1})}|1\rangle\right)$$

Dies können wir wie folgt fortsetzen:

$$\begin{aligned}\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2i\pi(0.x_jx_{j-1})}|1\rangle\right) &\xrightarrow{\text{Rot}(3)^{x_{j-2}}} \frac{1}{\sqrt{2}}\left(|0\rangle + e^{2i\pi\left(\frac{x_j}{2}+\frac{x_{j-1}}{4}+\frac{x_{j-2}}{8}\right)}|1\rangle\right)\\ &= \frac{1}{\sqrt{2}}\left(|0\rangle + e^{2i\pi(0.x_jx_{j-1}x_{j-2})}|1\rangle\right)\end{aligned}$$

$$\dots$$

$$\frac{1}{\sqrt{2}}\left(|0\rangle + e^{2i\pi(0.x_jx_{j-1}\dots x_{j-k+1})}|1\rangle\right) \xrightarrow{\text{Rot}(1+k)^{x_{j-k}}} \frac{1}{\sqrt{2}}\left(|0\rangle + e^{2i\pi(0.x_jx_{j-1}x_{j-2}\dots x_{j-k+1}x_{j-k})}|1\rangle\right)$$

Auf diese Weise können wir mit Hilfe der Rot(k)-Gatter die Nachkommastellen der Binärzahl eine Stelle nach der anderen aufbauen. Zum Schluss müssen wir noch die Reihenfolge der Qubits umkehren, was mit $SWAP$-Gattern erreicht werden kann.

Wenn wir uns den Schaltkreis anschauen, stellen wir fest, dass wir $\mathcal{O}(n^2)$ Gatter benötigen, um diese Operation zu implementieren, wobei n die Anzahl Qubits des Registers ist. Dabei werden auch keine zusätzlichen temporäre Qubits benötigt.

3.3 Addition

Die wohl grundlegendste arithmetische Operation ist die Addition. Die Subtraktion kann als Addition ausgedrückt werden und auch die Multiplikation (und somit die Division) sind abhängig von der Addition. Aus diesem Grund ist die Addition die erste arithmetische Operation, die wir betrachten.

Wir wollen die Operation implementieren, welche zwei QInts mit Quantenregistern der Grösse n in den Zuständen $|x\rangle$ und $|y\rangle$ nimmt und den Zustand des zweiten QInts auf $|x+y\rangle$ setzt. Die Implementierung anderer Additionsmethoden (Addition einer klassischen Zahl zu einem QInt, Addition zweier QInts in ein drittes QInt) funktioniert analog. Zusätzlich kann man feststellen, dass die Subtraktion nichts anderes als die inverse Operation zur Addition ist, somit hat man zur Addition die Subtraktion mit-implementiert.

Für die Addition gibt es zwei verschiedene Techniken, die häufig benutzt werden. Die eine erreicht eine Gatterzahl von $\mathcal{O}(n)$, benötigt aber $\mathcal{O}(n)$ zusätzliche Qubits, während die andere ohne zusätzliche Qubits auskommt, dafür aber $\mathcal{O}(n^2)$ Gatteroperationen benötigt. Ich habe mich entschieden, in meiner QInteger-Library die zweite Version zu implementieren. Gründe dafür sind, dass in heutigen Systemen die Anzahl verfügbarer Qubits stark begrenzt ist und in Simulationen einzelne Qubits sehr viel zusätzliche Leistung benötigen, während eine Laufzeit von $\mathcal{O}(n^2)$ in diesem Fall weniger ausmacht. Sobald mehr Qubits zur Verfügung stehen, wird es lohnenswerter sein, auf die andere Version zu wechseln, denn da die Addition eine Operation auf einem sehr tiefen Level ist, kann die Zeit, welche sie benötigt, beträchtliche Auswirkungen auf die gesamte Laufzeit eines komplexeren Algorithmus haben.

Schauen wir uns nun den in der QInteger-Library verwendeten Additionsalgorithmus an. Der Algorithmus basiert auf der Fourierbasis (und somit auf der Faktorisierung der Fouriertransformation). Bei der Addition in der binären Basis sind die einzelnen Bits voneinander abhängig. Deshalb werden sogenannte Carry-Bits verwendet, welche für jedes Bit abspeichern, ob wir beim nächsten Bit noch eine zusätzliche 1 addieren müssen. Dies ist bei der Fourierbasis nicht der Fall: Die Bits sind voneinander unabhängig. Das heisst, wir können die einzelnen Bits voneinander unabhängig modifizieren, ohne dabei auf die anderen Bits achten zu müssen. Dies ist der grosse Vorteil der Fourierbasis, der es uns erlaubt, auf zusätzliche Qubits zu verzichten. Betrachten wir nun noch einmal die Faktorisierung: Das j-te Qubit der Zahl y in der Fourierbasis ist im Zustand $\frac{1}{\sqrt{2}}(|0\rangle+e^{2i\pi\frac{y}{2^{1+j}}}|1\rangle)$. Wir wollen es aber in den Zustand $\frac{1}{\sqrt{2}}(|0\rangle+e^{2i\pi\frac{x+y}{2^{1+j}}}|1\rangle)$ bringen, denn wenn wir alle Qubits in den entsprechenden Zustand bringen können, könnten wir mit der inversen QFT den Zustand $|x+y\rangle$ wiederherstellen. Nehmen wir wieder das aus der Fouriertransformation bereits bekannte Gatter $\mathrm{Rot}(k)=\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 0\\ 0 & e^{\frac{2i\pi}{2^k}}\end{bmatrix}$. Mit diesem können wir den Wert $2i\pi\frac{1}{2^k}$ zum Exponenten des Koeffizienten von $|1\rangle$ addieren. Das heisst, wenn wir das Gatter auf ein Qubit im Zustand $\frac{1}{\sqrt{2}}(|0\rangle+e^{2i\pi\frac{y}{2^{1+j}}}|1\rangle)$ anwenden, wird es in den Zustand $\frac{1}{\sqrt{2}}(|0\rangle+e^{2i\pi\frac{y+2^{1+j-k}}{2^{1+j}}}|1\rangle)$ versetzt. Wir können also mit Hilfe dieses Gatters Zweierpotenzen zum Qubit in der Fourierbasis addieren. Wenn wir das Qubit im Zustand $|x\rangle$ in der binären Basis belassen, können wir die Addition wie folgt mit $\mathcal{O}(n^2)$ Gatteroperationen implementieren:

1. Wende die QFT auf den zweiten Summanden im Zustand $|y\rangle$ an. Das Register befindet sich nun im Zustand $\frac{1}{\sqrt{2^n}}\bigotimes_{j=0}^{n-1}\left(|0\rangle+e^{2i\pi\frac{y}{2^{1+j}}}\right)$.

2. Für jedes $j=0,...,n-1$, wende für jedes Bit x_k mit $k\leq j$ ein kontrolliertes $\mathrm{Rot}^{x_k}(j+1-k)$ auf das Qubit im Zustand $\frac{1}{\sqrt{2}}\left(|0\rangle+e^{2i\pi\frac{y}{2^{1+j}}}\right)$ im y-Register an. Dieses befindet sich danach

im Zustand:

$$\frac{1}{\sqrt{2}}(|0\rangle + e^{2i\pi\frac{y}{2^{1+j}}} \cdot e^{2i\pi\sum_{s=0}^{j}\frac{x_{j-s}\cdot 2^{j-s}}{2^{1+s}\cdot 2^{j-s}}}|1\rangle) = \frac{1}{\sqrt{2}}(|0\rangle + e^{2i\pi\frac{y+\sum_{s=0}^{j}x_{j-s}\cdot 2^{j-s}}{2^{1+j}}}|1\rangle)$$
$$= \frac{1}{\sqrt{2}}(|0\rangle + e^{2i\pi\frac{y+x}{2^{1+j}}}|1\rangle)$$

Wobei uns alle Bits mit Indizes grösser als j nicht interessieren, wie man analog zu den Überlegungen aus Abschnitt 3.2.1 feststellen kann.

3. Die Qubits im zweiten Register befinden sich nun in folgendem Zustand: $\frac{1}{\sqrt{2^n}}\bigotimes_{j=0}^{n-1}\left(|0\rangle + e^{2i\pi\frac{x+y}{2^{1+j}}}\right)$. Mit der inversen QFT kann man aus diesem Zustand den Zustand $|x+y\rangle$ wiederherstellen.

3.4 Modulare Addition

Den uns bekannten Modulo-Operator kann man auf Qubits nicht implementieren, da er nicht invertierbar ist (a und $a+m$ haben dasselbe Resultat modulo m). Die modulare Addition ist jedoch invertierbar, wenn beide Summanden kleiner als m sind. Wir haben wieder zwei QInts in den Zuständen $|x\rangle$ und $|y\rangle$ und eine klassische Zahl m und möchten den Zustand des zweiten QInts auf $|x+y \pmod m\rangle$ setzen[2].

Zuerst addieren wir $|x\rangle$ zum Register $|y\rangle$, um das Register in den Zustand $|x+y\rangle$ zu versetzen. Nun überprüfen wir, ob diese Summe grösser oder gleich m ist.

Wie überprüfen wir, ob ein QInt x grösser oder gleich einer anderen Zahl y ist? In der QInteger-Bibliothek ist die Funktion *GreaterOrEqual* als $\neg$*LessThan* implementiert[3]. Das heisst, wir versuchen herausezufinden, ob $x < y$ gilt, und negieren dann das Resultat. Wir können dies wie folgt umformen: $x < y \Leftrightarrow x - y < 0$. Subtrahieren wir also y von x, müssen wir herausfinden, ob das Resultat negativ ist. Ist $x-y$ nun tatsächlich negativ, passiert ein sogenannter *Underflow*. Dies bedeutet, dass $x-y$ zu $2^n+(x-y)$ wird, weil das nächstgrössere Bit fehlt. Um zu erkennen, dass dieses von klassischen Bits bekannte Phänomen auch bei Qubits auftritt, beobachten wir, was passiert, wenn wir a und b auf Qubits addieren, mit $0 \leq a, b \leq 2^n - 1$ und $a + b \geq 2^n$. Der Zustand in der Addition vor der inversen QFT ist:

$$\bigotimes_{s=0}^{n-1}\left(|0\rangle + e^{2i\pi\frac{a+b}{2^{(1+s)}}}|1\rangle\right) = \bigotimes_{s=0}^{n-1}\left(|0\rangle + e^{2i\pi\left(\frac{a+b}{2^{(1+s)}}-2^{n-1-s}\right)}|1\rangle\right) = \bigotimes_{s=0}^{n-1}\left(|0\rangle + e^{2i\pi\frac{a+b-2^n}{2^{(1+s)}}}|1\rangle\right)$$

Die Resultate $a+b$ und $a+b-2^n$ sind somit nicht unterscheidbar. Da aber nach der inversen QFT kein Bit vorhanden ist, um das Bit mit dem Wert 2^n zu speichern, wird das Resultat dieser Addition zu $a+b-2^n$. Gleichzeitig ist die Subtraktion dazu die inverse Operation. Wenn wir also b von $a+b-2^n$ subtrahieren, erhält man $a - 2^n \leq 2^n - 1 - 2^n < 0$, das Resultat wird aber zu a. Dies bedeutet, dass wenn das Resultat einer Subtraktion $x-y$ kleiner als 0 ist, es zu $2^n+(x-y)$ wird. Somit können wir messen, ob $x-y<0$ gilt, indem wir uns einfach das vorderste

[2] Hier lässt sich m durch einen QInt ersetzen (oder der erste Summand durch eine klassische Zahl). Für den Shor-Algorithmus benötigen wir die Operation nur mit einem klassischen m, die Implementierungen für m als QInt folgen analog und sind auch in der QInt-Library enthalten.

[3] Hier unterscheiden sich die Implementierungen für den Fall, dass die andere Zahl auch ein QInt oder eine klassische Zahl ist. Die zugrundeliegende Idee ist jedoch bei beiden Implementierungen die gleiche.

Qubit anschauen, welches dem Wert 2^{n-1} entspricht. Ist dieses 1, gab es möglicherweise einen Underflow. Um sicherzugehen, dass das vorderste Bit nur dann 1 ist, und nicht wenn $x-y \geq 2^{n-1}$ gilt, verlängere ich in meiner Implementierung vor der Subtraktion noch die Länge der beiden Register um 1, so dass $0 \leq a, b \leq 2^{n-1} - 1$ gilt, wobei n die neue grösse des Registers ist. Dann muss auch $a - b \leq 2^{n-1} - 1 - 0 < 2^{n-1}$ gelten, und so kann das erste Bit nur dann 1 sein, wenn ein Underflow auftritt.

Wir können nun herausfinden, ob $x + y \geq m$ gilt, und diese Information in einem zusätzlichen Qubit speichern. Falls dies zutrifft, subtrahieren wir m von der Zahl und erhalten somit die Zahl $x + y - m$ im zweiten QInt. Die Information, ob $x + y \geq m$ gilt, ist aber noch in einem Qubit gespeichert, welches wir zurücksetzen müssen, da wir Qubits nur im Zustand $|0\rangle$ freigeben können[4]. Wir werden nun beweisen, dass $x + y \geq m$ genau dann gilt, wenn das Endresultat $x + y \pmod m$ kleiner als der Summand x ist.

Dass die Richtung $(x + y \pmod m) < x \Rightarrow x + y \geq m$ stimmt, erkennt man daran, dass $x + y \geq x$ gilt. Damit das Resultat $(x + y \pmod m)$ kleiner als x ist, muss etwas von $x + y$ abgezogen worden sein. Dies kann aber nur der Fall sein, wenn $x + y \geq m$ gilt, denn dann subtrahiert man m von $x + y$.
Die andere Richtung, $(x + y \pmod m) \geq x \Rightarrow x + y < m$, zeigen wir, indem wir annehmen, dass $x + y \geq m$ gilt und wir m subtrahiert haben. Daraus würde folgen, dass $x + y - m \geq x$. Dies kann man umformen zu $y \geq m$, was ein Widerspruch zur Annahme $x, y < m$ ist. Somit kann in diesem Fall $x + y \geq m$ nicht stimmen.

So können wir mit diesem Vergleich die Information in unserem Aushilfsqubit wieder löschen.

3.5 Modulare Multiplikation

Mit Hilfe der modularen Addition können wir nun die modulare Multiplikation implementieren. Zuerst stellen wir fest, dass wir die modulare Multiplikation $|x\rangle \to |(ax) \pmod m\rangle$ nur dann implementieren können, wenn $\text{ggT}(a, m) = 1$ gilt, ansonsten wäre sie nicht invertierbar (zum Beispiel ist $2 \cdot 3 \equiv 2 \equiv 2 \cdot 1 \pmod 4$). Gilt hingegen $\text{ggt}(a, m) = 1$, so lässt sich ein Inverses a^{-1} zu a modulo m finden, sodass $a \cdot a^{-1} \equiv 1 \pmod m$ gilt. Somit lässt sich die Operation durch die Multiplikation mit a^{-1} invertieren.

Wir schauen uns die modulare Multiplikation auf QInts in zwei Schritten an.
Zuerst implementieren wir die Quantenoperation auf zwei Registern, welche für gegebenes a und m folgende Operation bewirkt:

$$U'_{a,m} |x\rangle |y\rangle = |x\rangle |(y + ax) \pmod m\rangle$$

Wir sehen, dass wenn wir $x = 2^0 x_0 + 2^1 x_1 ...$ in seine Zweierpotenzen aufteilen, wir $ax \pmod m = x_0(2^0 a \pmod m) + x_1(2^1 a \pmod m) + ...$ schreiben können. Dieses $+ x_0(...)$ ist nichts anderes als eine Addition, kontrolliert durch das Qubit x_0. Das bedeuet, wir können diese Operation relativ einfach durchführen:
Führe für jedes j eine modulare Addition durch, kontrolliert durch das Qubit x_j, auf das Ausgaberegister mit dem Summanden $2^j a \pmod m$, den wir klassisch berechnen können.

[4] Wäre dieses temporäre Qubit immer noch verschränkt mit den restlichen Qubits, könnte dies deren Zustand beeinflussen oder sogar zerstören.

Diese Unteroperation ruft den modularen Addierer $\mathcal{O}(n)$ mal auf und jede dieser Additionen braucht $\mathcal{O}(n^2)$ Gatteroperationen. Damit kommen wir auf $\mathcal{O}(n^3)$ Gatteroperationen.
Mit Hilfe dieser Unteroperation können wir nun die Operation, welche

$$U_{a,m} |x\rangle = |(ax) \pmod m\rangle$$

bewirkt, implementieren:

1. Führe ein temporäres Register im Zustand $|0\rangle$ ein und bringe es mit Hilfe der obigen Unteroperation in den Zustand $|ax \pmod m\rangle$.

2. Berechne klassisch das Inverse von a modulo m. Dieses lässt sich effizient mit Hilfe des erweiterten euklidischen Algorithmus[5] berechnen. Das Inverse existiert, da a und m teilerfremd sind.

3. Die inverse Operation zur oben definierten Unteroperation $U'_{a,m}$ ist

   $$[U'_{a,m}]^{-1} |u\rangle |v\rangle = |u\rangle |v - au \pmod m\rangle$$

   Wende diese Operation mit a^{-1} mit dem temporären Register als erstem Register und dem Register im Zustand $|x\rangle$ als zweitem Register an. Dies ergibt den Zustand:

   $$\begin{aligned} &[U'_{a^{-1},m}]^{-1} |ax \pmod m\rangle |x\rangle \\ =&|ax \pmod m\rangle \left|x - a^{-1}(ax) \pmod m\right\rangle \\ =&|ax \pmod m\rangle |0\rangle \end{aligned}$$

4. Wechsle den Zustand der beiden Register mit Hilfe der Swap-Operation, sodass $|ax \pmod m\rangle$ im ersten Register und das temporäre Register wieder im Zustand $|0\rangle$ ist. Das temporäre Register kann wieder wieder freigeben werden und das erste Register befindet sich nun im Zustand $|ax \pmod m\rangle$.

Diese Multiplikation benötigt n extra Qubits für das temporäre Register. Sie ruft die Unteroperation $2 \in \mathcal{O}(1)$ mal auf und benötigt somit $\mathcal{O}(n^3)$ Gatteroperationen. Wir erinnern uns, dass man die Addition mit $\mathcal{O}(n)$ Gatteroperationen und dafür n zusätzlichen Qubits implementieren könnte, was dazu führt, dass wir nur noch $\mathcal{O}(n^2)$ Gatteroperationen, dafür aber $\mathcal{O}(n)$ zusätzliche Qubits benötigen würden.

[5] Siehe auch [4], Kapitel 31

Kapitel 4

Der Weg zum Shor-Algorithmus

4.1 Überblick

Im folgenden Kapitel werden wir uns die relevanten Konzepte und Ideen hinter dem quantenbasierten Teil des Shor-Algorithmus anschauen. Dabei beginnen wir beim simplen Konzept des "Phase Kickback" , schauen uns dann die darauf basierende Phasenabschätzung an, bevor wir mit deren Hilfe die Ordnung einer Zahl finden. Zum Schluss werden wir uns den kompletten quantenbasierten Teil des Algorithmus zusammenfassend anschauen.

4.2 Phase Kickback

Ich beginne den Abschnitt mit einer Frage: Wenn wir eine kontrollierte Operation ausführen, sollte sich das Kontrollqubit eigentlich nicht ändern, oder? Im Folgenden werden wir sehen, dass dem überraschenderweise doch so sein kann. Dazu schauen wir uns das CNOT-Gatter an. Was geschieht, wenn wir CNOT auf zwei Qubits im Zustand $|+-\rangle$ anwenden[1], mit dem ersten Qubit als Kontrollqubit? Unser Ausgangszustand ist $|+-\rangle = \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)$. Nachdem wir das CNOT angewendet haben, erhalten wir den Zustand $\frac{1}{2}(|00\rangle - |01\rangle - |10\rangle + |11\rangle) = |--\rangle$. Überraschenderweise stellen wir fest, dass sich das Kontrollqubit verändert hat, während das Ziel-Qubit gleich geblieben ist. Was ist geschehen? Betrachten wir das CNOT-Gatter genauer: Es ist nichts anderes als eine kontrollierte Version des X-Gatters. Was geschieht, wenn wir das X-Gatter auf den $|-\rangle$-Zustand anwenden? $X|-\rangle = -\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle = -|-\rangle = (-1) * |-\rangle$. Hier können wir sehen, dass $|-\rangle$ ein Eigenvektor des X-Gatters mit Eigenwert -1 ist. Der Zustand ändert sich somit bis auf einen globalen Faktor nicht, was eine Unterscheidung zum ursprünglichen Zustand durch eine Messung verunmöglicht, wie in 2.2.2 ausgeführt wurde.

Wenn wir hingegen die Operation kontrolliert durchführen, wird diese Phase nur in den Zuständen sichtbar, in welchen die Operation durchgeführt wurde, sprich in den Zuständen, in denen das Kontrollqubit im Zustand $|1\rangle$ ist. Dies konnten wir zuvor beim CNOT-Gatter beobachten. Betrachten wir nun eine allgemeinere Operation U mit einem Eigenvektor $|\psi\rangle$ und dem Eigenwert λ. Nehmen wir jetzt ein Kontrollqubit im Zustand $\alpha|0\rangle + \beta|1\rangle$, ein Qubit-Register im Zustand $|\psi\rangle$ und wenden U auf das Register $|\psi\rangle$ an, kontrolliert durch jenes Kontrollqubit:

$$(\alpha|0\rangle + \beta|1\rangle)|\psi\rangle \xrightarrow{U^c} \alpha|0\psi\rangle + \beta|1\rangle \otimes U|\psi\rangle = (\alpha|0\rangle + \lambda\beta|1\rangle)|\psi\rangle$$

[1] $|+\rangle$ und $|-\rangle$ sind im Kapitel 2.2.4 definiert.

Das Ziel-Qubit verändert sich nicht – es ist ja ein Eigenvektor – stattdessen sehen wir, dass der Eigenwert zurück in die Phase des Kontrollqubit "gekickt" wird, daher der Name "Phase Kickback". Im nächsten Abschnitt werden wir diesen Effekt anwenden, um den Eigenwert eines Operators abzuschätzen.

4.3 Phasenabschätzung

Verschiedene Quanten-Algorithmen basieren darauf, den Eigenwert eines Operators U zu einem Eigenvektor $|\psi\rangle$ abzuschätzen. Dazu werden Phase Kickbacks benutzt, um den Eigenwert in ein Zähler-register in der Fourier-Basis zu schreiben, welches wir dann mit der inversen Quantenfourier-transformation in die binäre Basis zurückrechnen. Dabei können wir die Anzahl Qubits variieren, um die Präzision der Approximation festzulegen. Genauer gibt der Algorithmus zum Eigenwert $\lambda = e^{2i\pi\theta}$ die Zahl $2^n\theta$ zurück, wobei n die Anzahl Qubits des Zählerregisters ist, die für eine bessere Präzision erhöht werden kann.

Um zu verstehen, wie dieser Algorithmus funktioniert, erinnern wir uns zuerst daran, wie eine Zahl in der Fourierbasis aussieht. Dazu benutzen wir die Bloch-Kugel. Wir erinnern uns, dass für die Zahl x in der Fourierbasis mit n Qubits das k-te Qubit (hier muss man aufpassen, denn wenn man die Faktorisierung anschaut, ist das k-te Qubit das k-te Qubit von rechts für $k = 0, \ldots, n-1$) um $2\pi\frac{2^k x}{2^n}$ um die Z-Achse gedreht wird. Das heisst, es befindet sich im Zustand $\frac{1}{\sqrt{2}}(|0\rangle + e^{2i\pi\frac{2^k x}{2^n}}|1\rangle)$. Wir machen jetzt die Beobachtung, dass wir mit Hilfe des Phase Kickbacks das gesuchte θ in der Fourierbasis in die Kontrollqubits schreiben können, da der Phase Kickback nichts anderes macht, als das Kontrollqubit auf diese Art und Weise zu rotieren. Beobachten wir nun, was geschieht, wenn wir 2^k-mal kontrolliert den Operator U anwenden (wobei hier $(U^c)^{2^k}$ die kontrollierte Version von U bedeutet, 2^k-mal angewendet):

$$\begin{aligned}
\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|\psi\rangle \xrightarrow{(U^c)^{2^k}} &\frac{1}{\sqrt{2}}(|0\psi\rangle + |1\rangle \otimes U^{2^k}|\psi\rangle) \\
=&\frac{1}{\sqrt{2}}(|0\rangle + (e^{2i\pi\theta})^{2^k}|1\rangle)|\psi\rangle \\
=&\frac{1}{\sqrt{2}}(|0\rangle + e^{2i\pi 2^k\theta}|1\rangle)|\psi\rangle \\
=&\frac{1}{\sqrt{2}}(|0\rangle + e^{2i\pi\frac{2^k(2^n\theta)}{2^n}}|1\rangle)|\psi\rangle
\end{aligned}$$

Dies entspricht genau dem k-ten Qubit der Repräsentation von $2^n\theta$ in der Fourierbasis. Das heisst, wenn wir ein Zählerregister im Zustand $|0\rangle$ nehmen und dann für $k = 0, ..., n-1$ zuerst das H-Gatter auf das k-te Qubit und dann 2^k-mal ein kontrolliertes U mit dem k-ten Qubit als Kontrollqubit auf $|\psi\rangle$ anwenden, erhalten wir das k-te Qubit der Darstellung von $2^n\theta$ in der Fourierbasis. Wenden wir anschliessend die inverse Fouriertransformation an, können wir die Zahl $2^n\theta$ im Zählerregister ablesen. Falls $2^n\theta$ keine ganze Zahl ist, erhalten wir im Zählerregister eine Superposition, wobei eine Zahl wahrscheinlicher ist, je näher sie beim echten Eigenwert liegt.

Um die Phase abzuschätzen, müssen wir also den Operator U mehrmals hintereinander anwenden, zuerst nur einmal, dann zweimal, im i-ten Mal 2^i mal. Dies führt dazu, dass wir die Operation $\mathcal{O}(2^n)$ mal anwenden müssen. Allerdings ist es oft möglich, dass wir die Operation U^{2^m} für einen beliebigen Parameter m implementieren können. Wenn dies möglich ist, brauchen wir nur n Anwendungen jener Operation.

Algorithmus

1. Initialisiere zwei Quantenregister, das Zählerregister und das Eigenvektor-Register, und setze das Eigenvektor-Register auf den gewünschten Eigenvektor $|\psi\rangle$.
2. Wende $H^{\otimes n}$ auf das Zähler-Register an, um es auf $|+\rangle^{\otimes n}$ zu setzen.
3. Für $k = 0, \ldots, n-1$: Wende auf den Eigenvektor $|\psi\rangle$ die Operation $(U^{c_k})^{2^k}$, also die zur 2^k-fachen Anwendung von U äquivalente Operation, kontrolliert durch das Qubit c_k, an.
4. Wende die inverse Quantenfouriertransformation auf das Zählerregister an, um die Approximation in die binäre Basis umzurechnen.
5. Miss das Zählerregister, um die Abschätzung abzulesen.

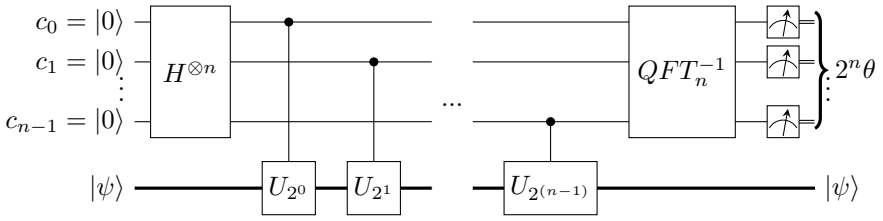

Abbildung 4.1: Schaltkreis der Phasenabschätzung

4.4 Periodenabschätzung

Gegeben sei eine Funktion $f : S \to S$ für eine endlichen Menge $S \subset \mathbb{Z}$, welche sich auf einem Quantencomputer implementieren lässt, und ein Wert $x \in S$. Wir versuchen nun, die kleinste Zahl $r \in \mathbb{N}$ zu berechnen, sodass $f^r(x) = x$ gilt.

Da die auf Quantencomputern berechneten Operationen unitär sind, müssen die Funktionswerte von f anhand von invertierbaren Operationen berechnet werden. Deshalb muss f injektiv sein und somit müssen $|S|$ verschiedene Funktionswerte von f existieren, woraus die Surjektivität folgt. Somit ist f bijektiv und man kann f als Permutation der Elemente von S interpretieren. Diese Permutation kann man als Komposition disjunkter Zykeln darstellen[2].

Ein *Zykel* ist eine Permutation $\sigma : S \to S$ mit den folgenden Eigenschaften:

Es gibt eine Teilmenge $\{x_1, x_2, \ldots, x_r\} =: S_\sigma \subset S$, so dass

1. $\sigma(x_r) = x_1$
2. $\sigma(x_i) = x_{i+1}$ für $i < r$
3. $\sigma(x) = x$ für $x \in S \setminus S_\sigma$

Zwei Zykeln σ und δ heissen *disjunkt*, wenn die Mengen S_σ und S_δ disjunkt sind.

Seien nun $\sigma_0, ..., \sigma_{k-1}$ disjunkte Zykeln mit $f = \sigma_0 \circ \sigma_1 \circ \ldots \circ \sigma_{k-1}$. Dazu sei S_i die Menge der Zahlen $j \in S$, so dass $\sigma_i(j) \neq j$ gilt.

[2]Für eine genauere Beschreibung und einen Beweis verweise ich auf Seiten 23 ff. von [12]

Sei nun $x \in S_i$. Da σ_i einen Zykel bildet, gilt $f^{|S_i|}(x) = x$. Gleichzeitig kann kein $r \in \mathbb{N}$ mit $r < |S_i|$ existieren, sodass $f^r(x) = x$ gilt, denn sonst hätte unser Zykel nur $r < |S_i|$ Elemente. Wir wollen nun also für ein $x \in S_i$ die Grösse $|S_i|$ finden.

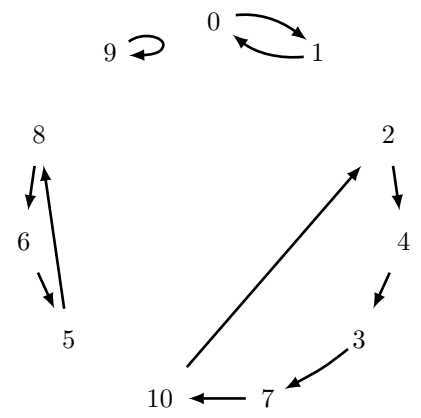

Abbildung 4.2: Der Funktionsgraph der Funktion $g(x) = -x^3 + 1 \pmod{11}$

Als Beispiel nehmen wir $g : S \to S$ mit $S = \mathbb{Z}/11\mathbb{Z}$, $g(x) = -x^3 + 1$. Wenn wir die Funktionswerte von g betrachten, sehen wir, dass diese Funktion bijektiv ist. Wenn wir dann den Graphen betrachten, sehen wir die einzelnen Zykeln und ihre dazugehörigen Mengen S_i: $S_0 = \{0, 1\}$, $S_1 = \{2, 4, 3, 7, 10\}$, $S_2 = \{5, 8, 6\}$ und $S_3 = \{9\}$. Wir sehen nun, dass $g^1(9) = 9$, $g^3(8) = 8$, $g^5(2) = 2$ etc.

Es fragt sich nun, wie wir effizient die Grösse der Teilmenge finden können, in der x sich befindet. Sei dafür U_f der Operator, welcher f auf Quantencomputern implementiert. Wir haben seine Existenz vorausgesetzt. Was geschieht, wenn wir diesem Operator eine Superposition der Zahlen in S_i übergeben? Seien $r = |S_i|$, $x_0, x_1, ..., x_{r-1}$ die Zahlen in S_i, sodass $f(x_j) = x_{(j+1) \pmod r}$ und U_f die Quantenoperation, die f implementiert. Betrachten wir, was geschieht, wenn wir U_f auf den Zustand $\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} |x_j\rangle$ anwenden. Wir erhalten:

$$U_f(\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} |x_j\rangle) = \frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} |f(x_j)\rangle = \frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} \left|x_{(j+1) \pmod r}\right\rangle = \frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} |x_j\rangle)$$

Daraus schliessen wir, dass $\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} |x_j\rangle$ ein Eigenvektor von U_f mit Eigenwert 1 ist. Dieser Eigenwert ist nicht wirklich interessant. Wir können ihn aber interessanter machen, indem wir den einzelnen Summanden eine Phase mitgeben. Dazu konstruieren wir die Superposition $\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} (e^{-2i\pi \frac{kj}{r}} |x_j\rangle)$ für ein $k < r$. Was geschieht, wenn wie U_f darauf anwenden?

$$\begin{aligned} U_f(\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} (e^{-2i\pi \frac{kj}{r}} |x_j\rangle)) &= \frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} (e^{-2i\pi \frac{kj}{r}} \left|x_{(j+1) \pmod r}\right\rangle) \\ = \frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} (e^{-2i\pi \frac{k(j-1)}{r}} |x_j\rangle) &= e^{2i\pi \frac{k}{r}} (\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} (e^{-2i\pi \frac{kj}{r}} |x_j\rangle))) \end{aligned}$$

Hier haben wir es mit einem Eigenvektor zum Eigenwert $e^{2i\pi \frac{k}{r}}$ zu tun. Dieser ist für uns interessant, weil r darin vorkommt. Wir machen die Beobachtung, dass unser Eigenvektor von vorher $(\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} |x_j\rangle)$ ebenfalls von der Form ist, die wir gerade analysiert haben, einfach mit $k = 0$. Wenn es uns nun gelingt, einen Zustand in der Form $\frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} (e^{-2i\pi \frac{kj}{r}} |x_j\rangle)$ zu erzeugen, können wir mit Hilfe der Phasenabschätzung den Quotienten $\frac{k}{r}$ abschätzen. Es fragt sich jedoch, wie wir einen solchen Zustand generieren können. Sei nun $|\psi_k\rangle := \frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} (e^{-2i\pi \frac{kj}{r}} |x_j\rangle)$. Was geschieht,

wenn wir die Vektoren $|\psi_0\rangle, |\psi_1\rangle, ..., |\psi_{r-1}\rangle$ aufsummieren?

$$\frac{1}{\sqrt{r}}\sum_{k=0}^{r-1}\frac{1}{\sqrt{r}}(\sum_{j=0}^{r-1}(e^{-2i\pi\frac{kj}{r}}|x_j\rangle)) = \frac{1}{r}\sum_{j=0}^{r-1}\sum_{k=0}^{r-1}(e^{-2i\pi\frac{kj}{r}}|x_j\rangle) = |x_0\rangle$$

Um dieses überraschende Resultat zu beweisen, betrachten wird die Summe $\sum_{k=0}^{r-1}(e^{-2i\pi\frac{kj}{r}}|x_j\rangle)$ für $j = 0$. Da $j = 0$ gilt, gilt $e^{-2i\pi\frac{kj}{r}} = e^0 = 1$ und somit $\sum_{k=0}^{r-1}(e^{-2i\pi\frac{k\cdot 0}{r}}|x_0\rangle) = r|x_0\rangle$. Da $\frac{1}{r}(r|x_0\rangle)$ bereits einen Betrag von 1 hat, kann kein anderer Zustand mit positivem Betrag existieren, da die Beträge sich sonst zu etwas Grösserem als 1 aufsummieren würden.

Nehmen wir nun die kürzere Schreibweise $|\psi_k\rangle = \frac{1}{\sqrt{r}}\sum_{j=0}^{r-1}(e^{-2i\pi\frac{kj}{r}}|x_j\rangle)$ für die oben genannten Eigenvektoren wieder auf. Somit ist $|x_0\rangle$ einfach eine Superposition jener Eigenvektoren. Wir zeigen nun, wie man aus dieser Superposition mit Hilfe der Phasenabschätzung den Eigenwert von einem der Zustände $|\psi_k\rangle$ abschätzen kann und wie man aus diesem Eigenwert eine Abschätzung der Periode r erhält.

Betrachten wir dazu den Phasenabschätzungs-Algorithmus nochmals bis zum Zeitpunkt vor der Messung des Zählerregisters. Dies können wir als eine Operation P mit $P|0\rangle|\psi\rangle \rightarrow |2^m\theta\rangle|\psi\rangle$ zu einem Eigenvektor $|\psi\rangle$ schreiben, wobei m die gewählte Genauigkeit der Phasenabschätzung ist. Da die auf Quantencomputern implementierten Operationen linear sind, gilt:

$$P|0\rangle|x_0\rangle = P\left(|0\rangle\left(\sum_{k=0}^{r-1}|\psi_k\rangle\right)\right) = \sum_{k=0}^{r-1}(P|0\rangle|\psi_k\rangle) = \sum_{k=0}^{r-1}(|2^m\theta_k\rangle|\psi_k\rangle) = \sum_{k=0}^{r-1}\left(\left|2^m\frac{k}{r}\right\rangle|\psi_k\rangle\right)$$

Messen wir dann das Resultat, so kollabiert diese Superposition und wir erhalten den Wert $2^m\frac{k}{r}$ für ein $k \in \{0, \dots r-1\}$, wobei alle Werte die gleiche Wahrscheinlichkeit haben. Mit Hilfe von Kettenbrüchen[3] können wir nun den Bruch $\frac{k}{r}$ vom Bruch $\frac{2^m\frac{k}{r}}{2^m}$ abschätzen. Nun stellt sich heraus: Wenn $m \geq 2n+1$ gilt, so ist der Bruch $\frac{2^m\frac{k}{r}}{2^m}$ exakt genug, sodass der Bruch mit Nenner $\leq |S|$, welcher die Differenz zu $\frac{2^m\frac{k}{r}}{2^m}$ minimiert, der Bruch $\frac{k}{r}$ selbst ist[4]. Wenn wir also mit Hilfe von Kettenbrüchen den Näherungsbruch mit dem grössten Nenner $\leq |S|$ nehmen, muss dieser Näherungsbruch $\frac{k}{r}$ sein.

Nun kann es sein, dass $\text{ggT}(k, r) = g \neq 1$ gilt, somit der Bruch mit g gekürzt wird, und wir so an Stelle von r die Zahl $\frac{r}{g}$ zurückbekommen. Wir können aber durch Anwendung von f^r überprüfen, ob r tatsächlich die Periode ist, da für $r' < r$ die Gleichung $f^{r'}(x) = x$ nicht gelten kann. Um den richtigen Wert r zu erhalten, können wir die Prozedur wiederholen. Wiederholen wir sie $2\log(|S|)$ mal, erhalten wir mit hoher Wahrscheinlichkeit mindestens einmal die korrekte Periode. Den Beweis dazu kann man in [13], Seiten 229ff., nachlesen.

4.5 Die Ordnung von Zahlen bestimmen

Der Algorithmus von Shor ist deshalb so schnell, da mit Hilfe des quantenbasierten Teils des Algorithmus die Ordnung einer Zahl effizient bestimmt werden kann. Die Ordnung einer Zahl

[3] Auf eine geneauere Betrachtung von Kettenbrüchen verzichte ich hier, sollte der Leser damit nicht vertraut sein, verweise ich auf [16]

[4] Für dieses Resultat verweise ich auf [13]. Die Beschreibung befindet sich auf Seite 229 und der Beweis des dazu verwendeten Theorems im Appendix 4

a modulo einer Zahl n, geschrieben als $\text{ord}_n(a)$, ist die kleinste positive Zahl r, sodass $a^r \equiv 1 \pmod{n}$ gilt. Dabei muss $\text{ggT}(a, n) = 1$ gelten, sonst kann dieses r nicht existieren (zum Beispiel gibt es kein r mit $2^r \equiv 1 \pmod{8}$). Wenn diese Bedingung jedoch erfüllt ist, besagt der Satz von Euler-Fermat, dass $a^{\phi(n)} \equiv 1 \pmod{n}$ gilt, wobei $\phi(n)$ die eulersche Phi-Funktion ist ($\phi(n)$ ist aber nicht zwingend der kleinste mögliche Exponent).

Wenn wir $\text{ggT}(a, n) = 1$ voraussetzen, ist die Restklasse kongruent zu a modulo n ein Element der primen Restklassengruppe $(\mathbb{Z}/n\mathbb{Z})^\times$. Insbesondere existiert ein inverses Element a^{-1} modulo n. Wir rechnen nun in $(\mathbb{Z}/n\mathbb{Z})^\times$.

Sei f definiert als $f(x) := ax$. Da diese Funktion invertierbar ist (die inverse Funktion ist $f^{-1}(x) = a^{-1}x$), ist sie bijektiv. Wir stellen nun fest, dass die Zahl $r = \text{ord}_n(a)$ die kleinste Zahl ist, für welche $f^r(1) = a^r \cdot 1 = 1$ gilt. Da f bijektiv ist, kann man sie auf Quantencomputern implementieren. Tatsächlich haben wir diese Funktion bereits implementiert: Sie ist nichts anderes als die in 3.5 beschriebene modulare Multiplikation.

Sei U_f die Quantenoperation, die f auf Quantencomputern implementiert. Nun können wir der Periodenabschätzungsfunktion diesen Quantenoperator und als Startwert die Zahl 1, bzw. den Zustand $|1\rangle$ übergeben, und sie berechnet für uns die kleinste Zahl r, sodass $f^r(1) = 1$ gilt, was tatsächlich die gesuchte Ordnung ist. Um klassisch effizient zu überprüfen, ob das gefundene r wirklich die Ordnung von a ist, können wir die Binäre Exponentiation[5] verwenden.

Es stellt sich nun die Frage, ob sich $U_f^{2^k}$ effizient implementieren lässt. Dazu stellen wir fest, dass $U_f^{2^k}$ nichts anderes als die Quantenoperation ist, welche f^{2^k} implementiert. Gleichzeitig gilt $f^{2^k}(x) = a^{2^k}x$. Dies wiederum ist nichts anderes als eine Modulare Multiplikation mit der Zahl a^{2^k}, wobei sich die Zahl $a^{2^k} \pmod{n}$ klassisch effizient berechnen lässt (auch dazu können wir wieder die Binäre Exponentiation verwenden). Somit können wir für jedes k die Quantenoperation $U_f^{2^k}$ mit nur einer modularen Multiplikation implementieren.

Fassen wir das Gesagte zusammen: Wir können die im vorangehenden Abschnitt erklärte Periodenabschätzungsfunktion verwenden, um die Ordnung einer Zahl a modulo einer Zahl n effizient abzuschätzen, falls $\text{ggT}(a, n) = 1$ gilt.

4.6 Das Ziel - Der Shor-Algorithmus

4.6.1 Der Algorithmus

Im Grunde genommen erlaubt uns der Algorithmus von Shor nur, einen nicht-trivialen Teiler d von n zu finden. Dies lässt sich jedoch leicht zu einem Algorithmus erweitern, welcher Zahlen faktorisiert: Solange n keine Primpotenz ist, finde einen nicht-trivialen Teiler d von n und faktorisiere rekursiv die beiden Zahlen d und $\frac{n}{d}$. Versuchen wir nun, einen nicht-trivialen Teiler von n zu finden.

In einem ersten Schritt überprüfen wir, ob n durch 2 teilbar und / oder eine Primpotenz ist. Letzteres lässt sich überprüfen, indem man für jede natürliche Zahl $s \leq \log(n)$ überprüft, ob eine

[5] Siehe auch [4], Abschnitt 31.6, "Raising to powers with repeated squaring"

Zahl p mit $p^s = n$ existiert. Diese Zahl p, falls sie existiert, kann man mit einer binären Suche finden (auch wenn p keine Primzahl ist, haben wir trotzdem bereits einen Faktor gefunden und können terminieren).

Als nächstes wählen wir eine zufällige Zahl a mit $1 < a < n-1$. Falls $\text{ggT}(a,n) \neq 1$ gilt, hat man bereits einen nicht-trivialen Teiler gefunden, den man zurückgeben kann. Andernfalls bestimmen wir mit Hilfe der Periodenabschätzung wie in Kapitel 4.5 beschrieben die Ordnung der Zahl a modulo n. Dies ist der einzige Schritt des Algorithmus, bei dem wir Quantencomputer benötigen. Sei diese Ordnung r. Falls r ungerade ist oder $a^{\frac{r}{2}} \equiv -1 \pmod{n}$ gilt, beginnen wir wieder von vorne. Die Wahrscheinlichkeit, dass wir wieder von vorne beginnen müssen, ist dabei aber kleiner als $\frac{1}{2^m}$, wobei m die Anzahl verschiedener Primteiler von n ist. Den Beweis dazu kann man in [13] Seiten 233ff. nachlesen.

Wir sind nun im Besitz einer Zahl a mit Ordnung r, wobei $2 \mid r$ und $a^{\frac{r}{2}} \not\equiv -1 \pmod{n}$ gilt. Da r die Ordnung ist, gilt auch $a^{\frac{r}{2}} \not\equiv 1 \pmod{n}$. Dies bedeutet, wir haben eine nicht-triviale Wurzel von 1 modulo n gefunden. Nun gilt $n \mid (a^{\frac{r}{2}}+1)(a^{\frac{r}{2}}-1)$, aber $n \nmid a^{\frac{r}{2}}+1$ und $n \nmid a^{\frac{r}{2}}-1$. Somit haben wir zwei Zahlen $b = a^{\frac{r}{2}}+1$ und $c = a^{\frac{r}{2}}-1$ mit:

$$n \mid bc \tag{1}$$
$$n \nmid b \tag{2}$$
$$n \nmid c \tag{3}$$

Sei nun $p_0^{\alpha_0} p_1^{\alpha_1} ... = n$ die Primfaktorzerlegung von n, $p_0^{\beta_0} p_1^{\beta_1} ... = b$ diejenige von b und $p_0^{\gamma_0} p_1^{\gamma_1} ... = c$ diejenige von c. Aus (1) folgt, dass für jedes k die Ungleichung $\beta_k + \gamma_k \geq \alpha_k$ stimmen muss. Gleichzeitig folgt aus (2) und (3), dass ein i und ein j existieren müssen, sodass $\beta_i < \alpha_i$ und $\gamma_j < \alpha_j$ gelten. Daraus folgt, dass $\text{ggT}(b,n)$ und $\text{ggT}(c,n)$ nicht n sein können. Gleichzeitig müssen wegen $\beta_k + \gamma_k \geq \alpha_k$ auch $\gamma_i > 0$ und $\beta_j > 0$ gelten, woraus $\text{ggT}(b,n) > 1$ und $\text{ggT}(c,n) > 1$ folgt.

Daraus schliessen wir, dass beide Zahlen b und c Faktoren von n enthalten. Diese können durch die Berechnung von $\text{ggT}(b,n)$ und $\text{ggT}(c,n)$ mit Hilfe des euklidischen Algorithmus extrahiert werden. Fassen wir all dies zusammen, sind wir im Besitz eines Algorithmus, der einen nicht-trivialen Teiler von n findet:

1. Falls n durch zwei teilbar ist, gib 2 zurück und terminiere.
2. Falls $n = p^a$ eine Primpotenz ist, gib die Primzahl p zurück und terminiere.
3. Bestimme eine zufällige Zahl $1 < a < n-1$.
4. Finde $g = \text{ggT}(a,n)$. Falls $g \neq 1$ ist, gib g zurück und terminiere.
5. Bestimme die Ordnung von a modulo n mit Hilfe des Quantenteils des Algorithmus:
   (a) Schätze die Phase des Operators U_f, der $f(x) = ax$ implementiert, mit einer Präzision von $m = 2\lceil \log_2(n) \rceil$ ab. Benutze dazu den in Kapitel 4.3 vorgestellten Algorithmus. Sei das Resultat $2^m \lambda$.
   (b) Schätze den Quotienten $\frac{k}{r}$ von $\frac{2^m \lambda}{2^m}$ ab. Falls r nicht die gesuchte Periode ist, gehe zurück zu (a), andernfalls gib die Periode r zurück.

6. Falls r ungerade ist, gehe zurück zu 2. Andernfalls berechne $a^{\frac{r}{2}} \pmod n$. Falls dies kongruent zu $-1 \pmod n$ ist, gehe zurück zu 2.

7. Berechne $b = (a^{\frac{r}{2}}+1)$ oder $c = (a^{\frac{r}{2}}-1)$. Gib $\text{ggT}(b, n)$ oder $\text{ggT}(c, n)$ zurück und terminiere.

4.6.2 Analyse des Algorithmus

Keiner der heute bekannten klassischen Faktorisierungsalgorithmen ist polynomiell[6] in der Länge $L = \lceil \log_2(n) \rceil$ der Zahl n, wobei die Länge der Zahl die Anzahl Bits beschreibt, welche man benötigt, um die Zahl zu speichern. Die meisten[7] bekannten klassischen Algorithmen sind polynomiell in $n \geq 2^{L-1}$ und werden somit als exponentielle Algorithmen bezeichnet. Der Algorithmus von Shor hingegen ist polynomiell in L und deshalb viel effizienter als alle bekannten Algorithmen. Ich werde nun die Laufzeit[8] meiner Implementierung analysieren.

Ob n durch zwei teilbar ist, lässt sich ganz einfach in $\mathcal{O}(1)$ überprüfen. Um zu überprüfen, ob n eine Potenz einer anderen Zahl ist, überprüfen wir für jedes $k \in \{1, \ldots, L\}$, ob eine Zahl s mit $s^k = n$ existiert. Ob diese Zahl existiert, können wir herausfinden, indem wir zuerst die grösste Zahl s' mit $(s')^k \leq n$ mit Hilfe einer binären Suche finden, und dann überprüfen, ob $(s')^k = n$ gilt. $(s')^k$ lässt sich in $\mathcal{O}(\log(k))$ mit der binären Exponentiation berechnen. Somit brauchen wir $\mathcal{O}(L^2 \log L)$ Schritte, um überprüfen zu können, ob n eine Potenz einer anderen Zahl ist, wobei ein Faktor L für die L Repetitionen und ein Faktor L von der binären Suche kommt.

Wenn n eine der beiden Kriterien erfüllt, terminieren wir. Deshalb können wir an dieser Stelle annehmen, dass n mindestens zwei verschiedene Primteiler besitzt. Dies führt dazu, dass die Chance, bei einem zufälligen a eine Periode r zu erhalten, die gerade ist und die $a^{\frac{r}{2}} \not\equiv -1 \pmod n$ erfüllt, grösser oder gleich als $\frac{1}{2}$ ist. Somit wird im Erwartungswert $\mathcal{O}(1)$ mal eine zufällige Zahl generiert und deren Periode berechnet. Genau so oft wird auch die Periodenabschätzungsfunktion aufgerufen.

Die Periodenabschätzungsfunktion hat ein Zählerregister der Grösse $2L$ und ruft somit die modulare Multiplikation $2L \in \mathcal{O}(L)$ auf. Auch wendet sie die QFT einmal an, diese ist jedoch im Vergleich mit der Laufzeit der aufgerufenen Multiplikationen vernachlässigbar. Jede davon ruft $2L \in \mathcal{O}(L)$ mal die Addition auf, welche zwei Fouriertransformationen zu $\mathcal{O}(L^2)$ Gatteroperationen anwendet. Rechnen wir alle diese Aufrufe zusammen, ergibt sich eine gesamte Gatterzahl von $\mathcal{O}(L^4)$ für einen Aufruf der Periodenabschätzungsfunktion. Somit ist die Gesamtlaufzeit des Algorithmus in $\mathcal{O}(L^4)$. Würde man die Addition mit Carry-Qubits implementieren, könnte man diese Zahl auf $\mathcal{O}(L^3)$ reduzieren.

Ich möchte nun einen Vergleich aufzeigen: Nehmen wir eine Zahl der Länge $L = 200$. Diese Zahl hat ungefähr 60 Stellen im Dezimalsystem. Rechnen wir nun dieses Beispiel durch, so braucht der naive klassische Algorithmus, der alle Zahlen bis zu $\sqrt{n}$ als Teiler ausprobiert, etwa $2^{100} \approx 10^{30}$ Operationen, wofür ein moderner Computer ungefähr 10^{13} Jahre benötigt, unter der Annahme,

[6] Ein Algorithmus ist *polynomiell* in L, wenn die Anzahl Operationen, die er benötigt, in $\mathcal{O}(L^c)$ für eine Konstante $c > 0$ ist.

[7] Es gibt auch einige Ausnahmen mit subexponentiellen Laufzeiten, wie z. B. der General Number Field Sieve Algorithmus, dessen Laufzeit man heuristisch auf etwa $\exp\left(c(\log n)^{\frac{1}{3}}(\log\log n)^{\frac{2}{3}}\right)$ abschätzen kann. Für eine genauere Beschreibung verweise ich auf [11]

[8] Unter der Laufzeit eines Algorithmus versteht man eine Abschätzung der Anzahl Operationen, die ein Computer durchführen muss, um den Algorithmus auszuführen.

dass ein moderner Computer etwa 10^9 Operationen pro Sekunde ausführen kann. Nimmt man auch an, dass das Universum etwa $1.4 \cdot 10^{10}$ Jahre alt ist, sieht man, dass der Computer somit ungefähr 1000-mal so lange benötigt, wie das Universum alt ist. Die Laufzeit des General Number Field Sieve Algorithmus kann man ähnlich auf etwa 10^{10} Sekunden abschätzen, was ungefähr einigen Jahrhunderten entspricht. Betrachten wir nun meine Implementierung des Algorithmus, benötigt sie ungefähr 10^9 Gatteroperationen. Auch wenn ein Quantencomputer nur 10^5 Gatteroperationen pro Sekunde machen könnte, was meiner Meinung nach eine sehr tiefe Schätzung ist, würde er trotzdem nur einige Stunden benötigen. Würde man die Implementatierung in $\mathcal{O}(L^3)$ nehmen, kommt man mit den gleichen Annahmen auf einige Minuten.

Kapitel 5

Ausblick

5.1 Man kann Zahlen effizient faktorisieren - was nun?

Wir haben nun gesehen, dass man Zahlen mit Hilfe von Quantencomputern effizient faktorisieren kann. Einerseits ist dies eine grosse Errungenschaft: Shors Algorithmus demonstriert uns, wie man mit Quantencomputern exponentielle Verschnellerungen erreichen kann. Auf der anderen Seite birgt diese Errungenschaft auch Gefahren. Zum Beispiel können Verschlüsselungsverfahren geknackt werden, wenn man Zahlen faktorisieren kann. Ich möchte hier kurz vorzeigen, wie man das RSA-Verschlüsselungsverfahren knacken kann, wenn man Zahlen effizient faktorisieren kann. Für eine Beschreibung des RSA-Verfahrens verweise ich auf [15].

Sei $n = pq$ der Modulus der Verschlüsselung und e der öffentliche Schlüssel. Da wir nun $n = pq$ faktorisieren können, können wir $\phi(n) = \phi(p) \cdot \phi(q) = (p-1)(q-1)$ berechnen. Damit können wir den privaten Schlüssel $d \equiv e^{-1} \pmod{\phi(n)}$ berechnen, denn wir wissen, dass für eine Nachricht m die Äquivalenzen $m^{e \cdot d} \equiv m^{e \cdot e^{-1} \pmod{\phi(n)}} \equiv m^1 \equiv m \pmod{n}$ gelten, wobei die erste davon wegen des Satzes von Euler-Fermat gilt.

Da nun Verschlüsselungen, die auf der Schwierigkeit der Faktorisierung beruhen, geknackt werden können, steht das Forschungsgebiet der Kryptographie vor neuen Herausforderungen. Das neue Gebiet, welches sich mit der Kryptographie im Zeitalter der Quantencomputer befasst, nennt sich Post-Quantum-Kryptographie.

5.2 Quantencomputer - wie bald?

Wie bereits angesprochen, gibt es verschiedene Möglichkeiten, Quantencomputer physikalisch zu realisieren. Technik-Firmen und Forschungsinstitute versuchen schon länger, Quantencomputer zu bauen. Bereits im Jahr 1998 gelang es zwei Forschern aus Oxford, einen Quantenalgorithmus für Deutschs Problem [8], und drei amerikanischen Forschern, Grovers Algorithmus experimentell zu realisieren [3]. Bereits im Jahr 2001 folgte darauf die erste experimentelle Realisierung des Shor-Algorithmus [17], mit welcher die Zahl 15 faktorisiert werden konnte.

Über die Jahre entwickelten sich Quantencomputer immer weiter. Zum Zeitpunkt dieser Arbeit haben Tech-Firmen wie Google, IBM, Intel etc. Quantenprozessoren mit bis zu 72 Qubits (Googles Bristlecone [9]). Im Jahr 2019 legte Google einen Artikel [1] vor, laut welchem sie die Quantenüberlegenheit erreicht hätten. Dies bedeutet, dass sie auf einem Quantencomputer etwas effizient berechnet hätten, was auf einem klassischen Computer nicht effizient berechenbar wäre. Dies löste prompt einen Disput aus und die IBM zweifelte in einem Blog die Quantenüberlegenheit

an [14]. Während im Bericht von Google behauptet wurde, ein klassischer Computer würde 10000 Jahre benötigen, wurde in IBMs Blog behauptet, dass ein klassischer Computer dies in 2.5 Tage tun könne. Der Artikel von Google blieb unpubliziert. Trotzdem ist es meiner Meinung nach beeindruckend, dass Googles Quantencomputer für diese Aufgabe nur etwa 200 Sekunden benötigt, während ein Supercomputer 2.5 Tage benötigt.

Die IBM selbst jedoch hat auch grosse Pläne. Im September 2020 hat sie in ihrer Quantum Roadmap [7] angekündigt, dass sie bis im Jahr 2023 einen Quantencomputer mit 1121 Qubits bauen und somit die Qubit-Zahlen sprengen möchte.

Wenn die IBM diesen Plan durchziehen kann und Google sowie die anderen Tech-Firmen auch in naher Zukunft so grosse Quantencomputer bauen, denke ich, dass die Zeit, in der quantenbasierte Supercomputer schwierige Berechnungen übernehmen werden, nicht mehr weit entfernt ist.

Kapitel 6

Nachwort

Sind Quantencomputer die Computer der Zukunft? Ich habe dieses nicht ganz einfache Thema gewählt, weil es mich seit längerem fasziniert. Da sie den komplexen, für uns nur schwer zu verstehenden Gesetzen der Quantenmechanik ausgesetzt sind, erlauben Quantencomputer es uns, manches effizienter zu berechnen, was auf klassischen Computern nicht effizient berechenbar ist. Wenn wir diese Gesetze verstehen und uns zunutze machen können, können wir sehr leistungsfähige Maschinen bauen.

Das Gebiet der Quantencomputer ist ein sehr junges Forschungsgebiet. Obwohl es sich aktuell rasant weiterentwickelt, ist dieses neue Gebiet uns noch weitgehend unbekannt und es liegt noch sehr viel Potenzial für zukünftige Entwicklungen darin verborgen.

Ich habe in dieser Arbeit versucht, mich der Thematik anzunähern und zu verstehen, was hinter diesen als so leistungsfähig gepriesenen Maschinen steckt. Der Weg, den ich in diesem Projekt zurückgelegt habe, war anspruchsvoll. Es dauerte einige Zeit, bis ich verstand, nach welchen Gesetzen die Quanten sich verhalten und welche mathematischen Strukturen dahinterstecken, da diese so fundamental anders funktionieren als in der klassischen Welt.

Auch das Programmieren meiner Bibliotheken gestaltete sich ungewohnt. Wenn man auf einem klassischen Computer beispielsweise eine Funktion einer Zahl programmiert, kann diese Zahl auf einem Quantencomputer verschiedene Werte gleichzeitig haben. Dadurch war es schwierig den Überblick über die Quantenzustände zu wahren und der Prozess des Debuggings wurde sehr zeitintensiv. Auch die Tatsache, dass alle auf Quantencomputern implementierbaren Funktionen invertierbar sein müssen, war für mich ungewohnt.

Doch die Mühe hat sich für mich gelohnt. Ich habe auf diesem Weg sehr viel gelernt und ich bin zuversichtlich, dass mich dieses Thema auch in Zukunft begleiten wird.

Kapitel 7

Anhang

7.1 Mathematische Symbole

$|v\rangle$ Schreibweise des Vektors v in der Bra-Ket-Notation.

$(|u\rangle, |v\rangle)$ **oder** $\langle u|v\rangle$ Skalarprodukt der beiden Vektoren $|u\rangle$ und $|v\rangle$.

$\| |v\rangle \|$ Norm des Vektors $|v\rangle$

$\mathbb{R}, \mathbb{C}$ Körper der reellen beziehungsweise der komplexen Zahlen.

$\mathbb{R}^n, \mathbb{C}^n$ n-dimensionaler Vektorraum über den reellen beziehungsweise den komplexen Zahlen

$\overline{A}, A^T, A^\dagger$ Die komplex konjugierte, die transponierte und die adjungierte Matrix zu der Matrix A. Siehe 2.1.4

I Die Identitätsmatrix mit der Eigenschaft $I|v\rangle = |v\rangle$ für alle $|v\rangle$.

H, X, Y, Z Das Hadamard- und die drei Pauli-Gatter. Siehe 2.2.4

QFT Die Quantenfouriertransformation. Siehe 3.2

f^k k-fache Anwendung der Funktion f.

U^c Kontrollierte Anwendung der Operation U. Siehe 2.2.5.

$\mathbb{Z}/n\mathbb{Z}$ Restklassenring der Restklassen bei der Division durch n.

$(\mathbb{Z}/n\mathbb{Z})^\times$ Prime Restklassengruppe der Restklassen bei der Division durch n, welche mit n teilerfremd sind.

$|S|$ Mächtigkeit einer Menge S.

$a \mid b$ a ist ein Teiler von b.

$\log(n)$ Logarithmus einer Zahl n. Meistens wird dabei der binäre Logarithmus gemeint, Logarithmen mit verschiedenen Basen unterscheiden sich jedoch nur durch einen Konstanten Faktor.

$f(n) \in \mathcal{O}(g(n))$ Es existieren c und n_0, sodass $0 \leq f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$.

7.2 Literatur

[1] F. Arute, K. Arya, R. Babbush, D. Bacon, J. C. Bardin, R. Barends, R. Biswas, S. Boixo, F. G. S. L. Brandao, D. A. Buell, B. Burkett, Y. Chen, Z. Chen, B. Chiaro, R. Collins, W. Courtney, A. Dunsworth, E. Farhi, B. Foxen, A. Fowler, C. Gidney, M. Giustina, R. Graff, K. Guerin, S. Habegger, M. P. Harrigan, M. J. Hartmann, A. Ho, M. Hoffmann, T. Huang, T. S. Humble, S. V. Isakov, E. Jeffrey, Z. Jiang, D. Kafri, K. Kechedzhi, J. Kelly, P. V. Klimov, S. Knysh, A. Korotkov, F. Kostritsa, D. Landhuis, M. Lindmark, E. Lucero, D. Lyakh, S. Mandrà, J. R. McClean, M. McEwen, A. Megrant, X. Mi, K. Michielsen, M. Mohseni, J. Mutus, O. Naaman, M. Neeley, C. Neill, M. Y. Niu, E. Ostby, A. Petukhov, J. C. Platt, C. Quintana, E. G. Rieffel, P. Roushan, N. C. Rubin, D. Sank, K. J. Satzinger, V. Smelyanskiy, K. J. Sung, M. D. Trevithick, A. Vainsencher, B. Villalonga, T. White, Z. J. Yao, P. Yeh, A. Zalcman, H. Neven und J. M. Martinis, "Quantum supremacy using a programmable superconducting processor," *Nature*, Jg. 574, Nr. 7779, S. 505–510, Okt. 2019, ISSN: 1476-4687. DOI: 10.1038/s41586-019-1666-5. Adresse: https://doi.org/10.1038/s41586-019-1666-5.

[2] A. Asfaw, L. Bello, Y. Ben-Haim, S. Bravyi, N. Bronn, L. Capelluto, A. C. Vazquez, J. Ceroni, R. Chen, A. Frisch, J. Gambetta, S. Garion, L. Gil, S. D. L. P. Gonzalez, F. Harkins, T. Imamichi, D. McKay, A. Mezzacapo, Z. Minev, R. Movassagh, G. Nannicni, P. Nation, A. Phan, M. Pistoia, A. Rattew, J. Schaefer, J. Shabani, J. Smolin, K. Temme, M. Tod, S. Wood und J. Wootton. (2020). "Learn Quantum Computation Using Qiskit," Adresse: http://community.qiskit.org/textbook.

[3] I. L. Chuang, N. Gershenfeld und M. Kubinec, "Experimental Implementation of Fast Quantum Searching," *Phys. Rev. Lett.*, Jg. 80, S. 3408–3411, 15 Apr. 1998. DOI: 10.1103/PhysRevLett.80.3408. Adresse: https://link.aps.org/doi/10.1103/PhysRevLett.80.3408.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest und C. Stein, *Introduction to Algorithms, Third Edition*, 3rd. The MIT Press, 2009, ISBN: 0262033844.

[5] T. G. Draper, "Addition on a Quantum Computer," Sep. 1998. Adresse: https://cds.cern.ch/record/450434/files/0008033.pdf.

[6] G. Fischer, *Lineare Algebra*, 18 A. Springer Spektrum, 2013, ISBN: 978-3-658-03944-8.

[7] J. Gambetta, *IBM's Roadmap For Scaling Quantum Technology*, Sep. 2020. Adresse: https://www.ibm.com/blogs/research/2020/09/ibm-quantum-roadmap/ (besucht am 28. 10. 2020).

[8] J. A. Jones und M. Mosca, "Implementation of a quantum algorithm on a nuclear magnetic resonance quantum computer," *The Journal of Chemical Physics*, Jg. 109, Nr. 5, S. 1648–1653, Aug. 1998, ISSN: 1089-7690. DOI: 10.1063/1.476739. Adresse: http://dx.doi.org/10.1063/1.476739.

[9] J. Kelly, *A Preview of Bristlecone, Google's New Quantum Processor*, März 2018. Adresse: https://ai.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html (besucht am 28. 10. 2020).

[10] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. USA: Addison-Wesley Longman Publishing Co., Inc., 1997, ISBN: 0201896842.

[11] A. K. Lenstra, H. W. Lenstra, M. S. Manasse und J. M. Pollard, "The Number Field Sieve," in *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, Ser. STOC '90, Baltimore, Maryland, USA: Association for Computing Machinery, 1990, S. 564–572, ISBN: 0897913612. DOI: 10.1145/100216.100295. Adresse: https://doi.org/10.1145/100216.100295.

[12] G. Mislin, *Algebra I.* vdf Hochschulverlag, 1998, ISBN: 978-3-7281-2408-1.

[13] M. A. Nielsen und I. L. Chuang, *Quantum Computation and Quantum Information*, 10. Aufl. Cambridge University Press, 2010, ISBN: 978-1-107-00217-3.

[14] E. Pednault, J. Gunnels, D. Maslov und J. Gambetta. (Okt. 2019). "On "Quantum Supremacy"," Adresse: https://www.ibm.com/blogs/research/2019/10/on-quantum-supremacy/ (besucht am 25. 10. 2020).

[15] R. L. Rivest, A. Shamir und L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems," *Commun. ACM*, Jg. 21, Nr. 2, S. 120–126, Feb. 1978, ISSN: 0001-0782. DOI: 10.1145/359340.359342. Adresse: https://doi.org/10.1145/359340.359342.

[16] P. Stopp, *Zur Arithmetik von Kettenbrüchen*, Sep. 2009. Adresse: https://www.math.uni-sb.de/ag/gekeler/PERSONEN/Bachelorarbeiten/Stopp.pdf.

[17] L. M. K. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood und I. L. Chuang, "Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance," *Nature*, Jg. 414, Nr. 6866, S. 883–887, Dez. 2001, ISSN: 1476-4687. DOI: 10.1038/414883a. Adresse: https://doi.org/10.1038/414883a.

[18] V. Vedral, A. Barenco und A. Ekert, "Quantum networks for elementary arithmetic operations," *Physical Review A*, Jg. 54, Nr. 1, S. 147–153, Juli 1996, ISSN: 1094-1622. DOI: 10.1103/physreva.54.147. Adresse: http://dx.doi.org/10.1103/PhysRevA.54.147.

7.3 Code

Hier eine Zusammenfassung der wichtigsten Code-Ausschnitten zu den einzelnen Kapitel.

Quantenfouriertransformation (zu Kapitel 3.2):

```
internal operation PerformQFT(qn : QInt) : Unit is Adj+Ctl
{
    let n = qn::Size;
    for (i in n-1..-1..0)
    {
        H(qn::Number[i]);
        for (j in i - 1..-1..0)
        {
            (Controlled RotQ)([qn::Number[j]],
                              (i - j + 1, qn::Number[i]));
        }
    }
    ReverseBits(qn);
}
```

Addition auf Qubits (zu Kapitel 3.3):

```
internal operation Add2nQFT(Summand : QInt,
                            Target : QInt) : Unit is Adj+Ctl
{
    let n = Target::Size;
    within
    {
        QFTQInt(Target);
    }
    apply
    {
        for (i in 0..n-1)
        {
            for (j in 0..i)
            {
                (Controlled RotQ)([Summand::Number[i]],
                                  (n - i - j, Target::Number[j]));
            }
        }
    }
}
```

Modulare Addition auf Qubits (zu Kapitel 3.4):

```
operation AddMod(Summand : QInt, Target : QInt,
                            Mod : QInt) : Unit is Adj+Ctl
{
```

```
    using (an = Qubit[4])
    {
        let c = an[3];
        let _Summand = QInt(Summand::Size + 1,
                            Summand::Number + [an[0]]);

        let _Target = QInt(Target::Size + 1,
                           Target::Number + [an[1]]);

        let _Mod = QInt(Mod::Size + 1, Mod::Number + [an[2]]);
        Add(_Summand, _Target);
        GreaterOrEq(_Target, _Mod, c);
        (Controlled (Adjoint Add))([c], (_Mod, _Target));
        GreaterThan(_Summand, _Target, c);
    }
}
```

Modulare Multiplikation auf Qubits (zu Kapitel 3.5):

```
internal operation _MulMod(a : Int, Target : QInt,
                                    Mod : Int) : Unit is Adj+Ctl
{
    if (GCD(a, Mod) == 1)
    {
        let inv = Inverse(a, Mod);
        let n = Target::Size;
        using (qr = Qubit[n])
        {
            let tmp = QInt(n, qr);
            MulModAdd(a, Target, tmp, Mod);
            (Adjoint MulModAdd)(inv, tmp, Target, Mod);
            for (i in 0..n - 1)
            {
                SWAP(tmp::Number[i], Target::Number[i]);
            }
        }
    }
    else
    {
        fail "GCD of a and Mod not equal to 1.";
    }
}
```

Phasenabschätzung (zu Kapitel 4.3):

```
operation EstimatePhase(U : ((Qubit[], Int) => Unit is Ctl),
                        Eigenstate : Qubit[], Precision : Int) : Int
{
    using (anc = Qubit[Precision])
```

```
    {
        for (i in 0..Precision - 1)
        {
            H(anc[i]);
        }
        mutable rep = 1;
        for (i in 0..Precision-1)
        {
            (Controlled U)([anc[i]], (Eigenstate, rep));
            set rep = rep * 2;
        }
        let qr = QIntR(anc);
        (Adjoint QFTQInt)(qr);
        let res = MeasureQInt(qr);
        ResetQInt(qr);
        return res;
    }
}
```

Periodenabschätzung (zu Kapitel 4.4):

```
operation FindPeriod(Start : Int,
            kth_next : ((Qubit[], Int) => Unit is Ctl),
            cls : ((Int, Int) -> Int), Size : Int,
            MAXREP : Int) : Int
{
    mutable res = 0;
    mutable rep = 0;
    let n = SmallestPow2WithPowBiggerThan(Size);
    let q = 2*n;
    using (qs = Qubit[n])
    {
        repeat
        {
            set rep = rep + 1;
            if (rep > MAXREP)
            {
                fail "Too many repetitions."
            }

            CopyToQInt(Start, QIntR(qs));
            let tp = EstimatePhase(kth_next, qs, q);
            mutable tr = 0;
            set (tr, res) = ApproximateFraction(tp,
                                                FastPow(2, q), Size);
            ResetAll(qs);
        }
        until (cls(Start, res) == Start and res > 0);
    }
```

```
    return res;
}
```

Ordnungsabschätzung - Quantenbasierter Teil von Shors Algorithm (zu Kapitel 4.6):

```
internal function MulByPow(sv : Int, bs : Int,
                           ex : Int, Mod : Int) : Int
{
    return (sv*FastPowMod(bs, ex, Mod)) % Mod;
}

internal operation QMulByPow(A : Int, Mod : Int,
                           trg : Qubit[], rep : Int) : Unit is Ctl
{
    MulMod(FastPowMod(A, rep, Mod), QIntR(trg), Mod);
}

operation ShorOrderFinder(A : Int, Mod : Int) : Int
{
    return FindPeriod(1, QMulByPow(A, Mod, _, _),
                           MulByPow(_, A, _, Mod), Mod, 20);
    // 20 Repetitions are enough for small QInt-sizes
    // As soon as we are able to use more Qubits,
    // Change this values to O(log Mod)
}
```

7.4 Redlichkeitserklärung