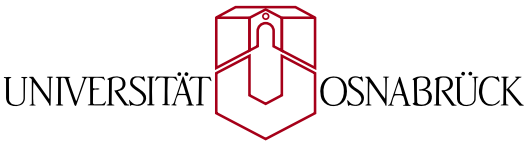
Untersuchungen zum Shor-Algorithmus

Bachelor-Arbeit

Nils Rosemann



12. Juli 2004

Inhaltsverzeichnis

1. Einleitung

In dieser Arbeit geht es um den Shor-Algorithmus. Der Shor-Algorithmus wurde 1994 von Peter W. Shor in den AT&T-Labs entwickelt und gilt als die zentrale Entwicklung in der Quanteninformatik. Denn der Shor-Algorithmus löst die Aufgabe, große natürliche Zahlen zu faktorisieren, in polynomieller Komplexität, während jeder bekannte klassische Algorithmus exponentielle Komplexität besitzt. Dadurch wird die RSA-Kryptografie gefährdet, da sie darauf basiert, dass Faktorisieren schwierig ist. Der Shor-Algorithmus ist damit ein Indiz, dass ein Quantencomputer wichtige Probleme effizienter lösen kann als ein klassischer Computer.

Um zu verstehen, warum das so ist, braucht man die Grundlagen der Quanteninformatik (Abschnitt 2). Die Quanteninformatik ist eine Verknüpfung von Quantenmechanik und Informationstheorie. Der Unterschied zur bisherigen Informatik liegt in der kleinsten Informationseinheit: Anstatt des mathematischen Bits betrachtet man das einfachste System der Quantenmechanik, ein Zwei-Zustands-System. Ein solches System nennt man Quantenbit. Damit ist eine völlig neue Sichtweise entstanden, Information ist eine physikalische Ressource!

Die Besonderheiten der Quantenmechanik ermöglichen es, die Fourier-Transformation effizient zu implementieren. Diese Quanten-Fourier-Transformation wird in Abschnitt 3 erläutert. Sie bildet das Rückgrat für viele Quanten-Algorithmen, so auch für den Shor-Algorithmus. Denn Shor zeigte, dass sich das Problem der Faktorisierung auf die Quanten-Fourier-Transformation zurückführen lässt. Die dazu nötigen Sätze und Reduktionsschritte werden in Abschnitt 4 zusammengefasst.

Obwohl der Shor-Algorithmus nur auf einem Quantencomputer effizient ist, kann man ihn auch auf einem klassischen Computer durchführen. Dazu muss der klassische Computer die Eigenschaften eines Quantencomputers simulieren. Abschnitt 5 diskutiert diese Simulation. Sie wird auf Grund der exponentiell anwachsenden Ressourcen (Speicher) aber nur bis zu einer trivialen Grenze möglich sein. Dennoch lassen sich die zentralen Aspekte und Konzepte gut daran verdeutlichen. Das Ergebnis ist eine Menge von Java-Klassen, die für die Simulation von Quanten-Algorithmen geeignet sind. Damit kann die Quanten-Fourier-Transformation und der Shor-Algorithmus intuitiv implementiert werden. Doch schon bei Zahlen, die größer sind als 63, wird der Speicheraufwand zu groß für normale Computer.

Die Simulationssoftware wird mit den Techniken des objektorientierten Software-Designs erstellt. Dabei stellt sich heraus, dass diese Techniken nur in begrenztem Maße zur Darstellung quantenmechanischer Zusammenhänge geeignet sind. Dies liegt daran, dass die Objektorientierung unserer klassischen Intuition entspricht. Abschnitt 5.1.1 führt diesen Gedanken weiter, indem nach Programmierparadigmen für Quanten-Algorithmen gefragt wird.

2. Quantencomputer

> *"Nobody unterstands quantum mechanics. No, you're not going to be able to understand it... You see, my physics students don't understand it either. That is because I don't understand it. Nobody does... The theory of quantum electrodynamics describes Nature as absurd from the point of view of common sense. And it agrees fully with experiment. So I hope you can accept Nature as She is — absurd."*
> Richard Feynman

> *"For reasons which nobody fully understands, entangled states play a crucial role in quantum computation [...]."*
> Michael Nielsen and Isaac Chuang

2.1. Was ist ein Quantencomputer?

Ein Quantencomputer ist ein mikroskopisches System, dessen Verhalten von den Gesetzen der Quantenmechanik bestimmt wird. Dieses System enthält Informationen, die durch Zeitentwicklung verändert werden. Schließlich überführen wir die "Quanteninformation" durch eine Messung in eine klassische Information. Durch das geschickte Ausnutzen quantenmechanischer Besonderheiten kann ein Quantencomputer bestimmte Berechnungen prinzipiell schneller ausführen als sein klassisches Pendant. Es können aber nur bestimmte Arten von Problemen schneller berechnet werden. Daher braucht man neue Algorithmen, Quantenalgorithmen (siehe Abschnitt 2.6), die die Besonderheiten des Quantencomputers ausnutzen. Die zwei wichtigsten sind der Shor-Algorithmus zur Faktorisierung von natürlichen Zahlen und der Grover-Algorithmus zur effizienten Suche in unstrukturierten Datenbanken. Die bei der Entwicklung dieser beiden Algorithmen benutzten Techniken bilden die Grundlage für viele andere Algorithmen.
Die beiden zentralen Besonderheiten der Quantenmechanik, die einen Quantencomputer schneller als jeden klassischen Computer machen, sind Superposition und Verschränkung. Diese Eigenschaften ermöglichen massiven Parallelismus und Effekte wie Quantenteleportation oder Quantenkryptografie.
Durch die Ausnutzung dieser Phänomene, die kein klassisches Analogon haben, sind Quantencomputer prädestiniert, quantenmechanische Systeme zu simulieren. Dies war auch die ursprüngliche Idee Richard Feynmans [Fey82], der sich in den 80er Jahren Gedanken darüber machte, wie Computern aussehen müssten, um diese Simulationen effizient durchführen zu können.
Ein Quantencomputer und ein klassischer Computer können sich gegenseitig simulieren, allerdings mit unterschiedlichem Aufwand. Da ein Quantencomputer das Repertoire eines klassischen Computers umfasst, kann er alle klassischen Algorithmen mit ähnlicher

Effizienz berechnen. Umgekehrt lassen sich die erweiterten Fähigkeiten, die sich durch Superposition und Verschränkung ergeben, auf einem klassischen Computer nur mit exponentiellem Aufwand abbilden.
Auch für einen Quantencomputer kennt man eine kleinste Informationseinheit: das Quantenbit, kurz Qubit, auf das in Abschnitt 2.2.2 näher eingegangen wird. Das Qubit kann jedoch exponentiell mehr Information tragen als das klassische Bit. Wie im klassischen Computer werden mehrere Qubits zu Registern zusammengefasst, auf denen man mit Quantengattern arbeitet. Diese Quantengatter entsprechen unitären Operatoren auf Hilberträumen, die wir in den Abschnitten 2.3 und 2.4 näher erklären. Man kann zeigen, dass bestimmte Sätze von Gattern universell sind [BBC$^+$95], das heißt, dass man jedes beliebige Gatter mit beliebiger Genauigkeit als Folge von Gattern aus dem universellen Satz darstellen kann. So ist es möglich, jeden Algorithmus durch einen universellen Satz zu implementieren, da er sich als Verkettung unitärer Operatoren darstellen lässt. Details zu Quantenschaltkreisen und deren Notation bietet Abschnitt 2.5.
Es gibt verschiedene Ansätze, Qubits und Gatter physikalisch zu realisieren: Ionenfallen (Qubits als Anregungszustände der Ionen, Gatter durch gezielte Manipulation mit Lasern) [CZ95], Nuclear Magnetic Resonance (Qubits als Kernspin, Gatter durch magnetische Felder) [Vin95] und Festkörpersysteme (Qubits durch Elektronenpaare, Gatter durch Elektrostatik) [NPT99, LD98]. Diese Verfahren stellen wir in Abschnitt 2.7 kurz dar.
Ein großes Problem bei dem Bau von Quantencomputern ist Dekohärenz: Die Interaktion des quantenmechanischen Systems mit seiner Umgebung stört die gespeicherte Quanteninformation und führt damit zu Informationsverlust. Die Wechselwirkung mit der Umgebung wirkt wie ein Messprozess, der Superposition und Verschränkung aufhebt. Die Güte eines Quantencomputers hängt entscheidend von der Zahl der möglichen Operationen ab, die durchgeführt werden können, bevor Dekohärenzeffekte auftreten.

2.2. Quantenbits

2.2.1. Formalismus

Quantenmechanische Zustände (wie z. B. der Zustand eines Quantenbits oder eines Quantenregisters) lassen sich durch Vektoren in einem Hilbertraum beschreiben. In dieser Arbeit wird die Dirac-Notation benutzt. Dabei wird ein Vektor aus dem betrachteten Hilbertraum "Ket" genannt und mit $|\phi\rangle$ (sprich: Ket Phi) bezeichnet. Die Klammer $|\rangle$ deutet dabei den Vektorcharakter an, während der Name (hier ϕ) zur Unterscheidung verschiedener Vektoren dient. In endlich-dimensionalen Hilberträumen lässt sich ein Ket als Spaltenvektor mit im Allgemeinen komplexen Einträgen schreiben.
Das Dualraumäquivalent eines Kets $|\phi\rangle$ bezeichnet man mit $\langle\phi|$ und nennt es "Bra". In endlich-dimensionalen Hilberträumen lässt sich ein Bra als Zeilenvektor schreiben. Als Element des Dualraumes ist ein Bra eine lineare Abbildung vom betrachteten Hilbertraum in die zugrunde liegenden komplexen Zahlen. Bildet man mit dem Bra ein Ket

ab, so schreibt man $\langle\phi|\psi\rangle$. Das Ergebnis ist also ein Skalar.
Eine wichtige Operation mit Kets ist das Tensorprodukt. Das Tensorprodukt verknüpft zwei Vektorräume U und V zu einem größeren Vektorraum $W = U \otimes V$. Eine Basis des Produkt-Hilbertraumes W erhält man durch das Tensorprodukt von jedem Basisvektor $|i\rangle$ aus U mit jedem Basisvektor $|j\rangle$ aus V.
Das Tensorprodukt für zwei Vektoren ist über ihre Indizes definiert. Wenn i ein Index für die Elemente ϕ_i vom Vektor ϕ ist und j ein Index für die Elemente ψ_j von ψ, dann ist ij ein Index für das Tensorprodukt. Dabei werden i und j aber nicht multipliziert, sondern als Ziffern angesehen. Wenn i und j beispielsweise von 0 bis 1 laufen, dann durchläuft ij die Werte 00, 01, 10 und 11, also vier Werte.
Damit ergibt sich die Formel:

$$(\phi \otimes \psi)_{ij} \equiv \phi_i \, \psi_j \quad . \tag{1}$$

Man schreibt auch $|ij\rangle$ anstatt $|i\rangle \otimes |j\rangle$. Jeder Basiszustand von $U \otimes V$ ist damit ein Produktzustand. Viele Superpositionen von Basiszuständen lassen sich nicht als Tensorprodukt schreiben, zum Beispiel $\frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$. Solche Zustände nennt man *verschränkt*. Sie spielen eine wichtige Rolle (siehe Zitat am Anfang des Kapitels) in der Quanteninformatik, vor allem in Bereichen wie Quantenkryptografie oder Quantenteleportation.
Entsprechend lässt sich das Tensorprodukt für Operatoren $A : U \to U$ und $B : V \to V$ definieren. Es wirkt einfach jeder Operator auf seinen Vektorraum: $(A \otimes B) : U \otimes V \to U \otimes V$. Damit gilt für $|a\rangle \in A$ und $|b\rangle \in B$:

$$(A \otimes B)(|a\rangle \otimes |b\rangle) \equiv (A\,|a\rangle) \otimes (B\,|b\rangle) \tag{2}$$

oder

$$(A \otimes B)_{i\nu,j\mu} \equiv A_{ij} B_{\nu\mu} \quad . \tag{3}$$

Da das Tensorprodukt von A und B auch auf Vektoren wirken kann, die sich nicht als Tensorprodukt von zwei Vektoren aus U und V darstellen lassen, muss man es noch linear fortsetzen.
Ein Beispiel: Seien

$$A = \begin{pmatrix} 2 & 0 \\ -1 & 1 \end{pmatrix}, \; B = \begin{pmatrix} 1 & 1 \\ -2 & -1 \end{pmatrix} \quad . \tag{4}$$

Dann ist das Tensorprodukt

$$(A \otimes B) = \begin{pmatrix} 2 \cdot B & 0 \cdot B \\ -1 \cdot B & 1 \cdot B \end{pmatrix} = \begin{pmatrix} 2 & 2 & 0 & 0 \\ -4 & -2 & 0 & 0 \\ -1 & -1 & 1 & 1 \\ 2 & 1 & -2 & -1 \end{pmatrix} \quad . \tag{5}$$

2.2.2. Vom Bit zum Qubit

In der klassischen Informatik ist die kleinste Informationseinheit das Bit. Ein Bit hat entweder den Zustand 0 oder 1, so dass sich mit n Bits 2^n Werte darstellen lassen.
In der Quanteninformatik gibt es als analoge kleinste Informationseinheit das Quantenbit oder Qubit [Sch95]. Ein Qubit ist ein quantenmechanischer Zustand in einem zweidimensionalen Hilbertraum. Dieser wird von zwei Basiszuständen aufgespannt, die man üblicherweise mit $|0\rangle$ und $|1\rangle$ bezeichnet. Sie bilden die sogenannte Rechenbasis (bei Registern, die aus mehreren Qubits bestehen, heißt die Basis $\{|00...0\rangle, |00...1\rangle, \cdots, |11...1\rangle$ Rechenbasis). Ein Qubit mit dem Zustand $|0\rangle$ (bzw. $|1\rangle$) ist vergleichbar mit dem klassischen Zustand 0 (bzw. 1), allerdings kann ein Qubit auch in einem Superpositionszustand dieser beiden Basiskets sein. Dabei ist jede Linearkombination $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$ erlaubt mit $|\alpha|^2 + |\beta|^2 = 1$. α und β sind komplexe Zahlen. Die Quanteninformatik interpretiert eine Superposition von $|0\rangle$ und $|1\rangle$ als *gleichzeitige* Speicherung von 0 und 1. Ein Qubit ist damit nicht wie das klassische Bit entweder im Zustand 0 oder im Zustand 1, sondern in beliebiger Überlagerung.
Im Gegensatz zum klassischen Bit, bei dem es jederzeit durch eine Messung möglich ist, seinen Zustand (0 oder 1) genau zu bestimmen, lässt sich bei einem Qubit sein Quantenzustand, also die Werte von α und β, nicht feststellen. Man kann genau eine Messung durchführen, das Ergebnis ist zufällig. Dies bedeutet, dass man den Zustand $|0\rangle$ mit der Wahrscheinlichkeit $|\alpha|^2$ und den Zustand $|1\rangle$ mit der Wahrscheinlichkeit $|\beta|^2$ misst. Da die Messung den ursprünglichen Quantenzustand auf den gemessenen Zustand projiziert, ist die gespeicherte Quanteninformation also nicht komplett als klassische Information verfügbar, was zur Folge hat, dass wir mit versteckten Informationen arbeiten, die wir nie (durch Beobachtung oder Messung) zu Gesicht bekommen. Gute Quantenalgorithmen müssen also möglichst ohne Zwischenmessungen auskommen, da bei Messungen Quanteninformation verloren geht und damit der Vorteil des Quantencomputers gegenüber dem klassischen Computer.
Als Beispiel betrachten wir den Zustand $|\phi\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$. Die Wahrscheinlichkeit, den Zustand $|0\rangle$ zu messen beträgt also

$$\left(\frac{1}{\sqrt{2}}\right)^2 = \frac{1}{2} = 50\%.$$

Die Wahrscheinlichkeit für $|1\rangle$ ist genauso groß.

2.3. Operationen auf Qubits

Möchte man mit Qubits arbeiten, dann benötigt man genau wie im klassischen Fall Operationen, die Qubits manipulieren (Gatter [BBC+95], [BCDP96]). Das Ziel ist, den durch die Qubits repräsentierten Anfangszustand in einen Zustand zu überführen, aus dessen Messung wir ein sinnvolles Ergebnis bekommen.

2.3.1. Ein-Qubit-Gatter

Zunächst soll nur ein einziges Qubit betrachtet werden, sozusagen der einfachste Quantencomputer, den man sich denken kann. Wie können wir dieses Qubit manipulieren? Klassische Computer arbeiten mit Gattern, ein triviales Beispiel ist das NOT-Gatter X, welches jedes Bit invertiert. Aus 0 wird 1, aus 1 wird 0. Wie man sich leicht vorstellen kann, existiert eine analoge Operation für Qubits. Aus dem Zustand $|0\rangle$ wird der Zustand $|1\rangle$ und umgekehrt. Im Falle einer Superposition zweier Zustände, z. B. dem Zustand $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$, arbeitet das Quanten-NOT-Gatter linear, das heißt in diesem Fall, die Rollen von $|0\rangle$ und $|1\rangle$ werden vertauscht: $|\phi'\rangle = X|\phi\rangle = \alpha|1\rangle + \beta|0\rangle$. Diese Linearität ist eine wichtige Eigenschaft in der Quanteninformatik. Verallgemeinert können wir die NOT-Operation als $X|a\rangle = |a \oplus 1\rangle$ für einen beliebigen Basiszustand $|a\rangle$, $a \in \{0, 1\}$, schreiben, wobei $\oplus$ die Addition modulo 2 meint. Mathematisch lässt sich eine solche Operation, nennen wir sie U, durch eine 2×2-Matrix repräsentieren. Damit ist sicher gestellt, dass es sich um eine lineare Operation handelt. Weiterhin muss die Normierungsbedingung $|\alpha|^2 + |\beta|^2 = 1$ auch nach der Matrixoperation gelten, weswegen die Matrix unitär sein muss. Es muss also gelten $U^\dagger U = I$. Dies ist die einzige Bedingung, die eine Matrix erfüllen muss, um ein erlaubtes Quantengatter darzustellen. So gibt es theoretisch unendlich viele Gatter für einzelne Qubits.
Eine weitere Folge dieser einen Bedingung ist die Umkehrbarkeit der Operation, das heißt im Gegensatz zu klassischen Gattern ist ein Quantengatter immer reversibel. Im klassischen Fall gibt es zum Beispiel die Gatter XOR oder NAND, die nicht invertierbar sind und somit zu einem Informationsverlust führen. Dies ist bei Quantengattern nicht möglich. Die Gesamtwahrscheinlichkeit des Systems muss vor und nach einer Operation gleich sein, was zusammen mit der Linearitätsbedingung nur umkehrbare Operatoren zulässt. Dieser Punkt bedarf besonderer Beachtung, wenn man versucht, klassische Algorithmen und Schaltlogik auf Quantencomputer zu übertragen.
Wenn man Hilfs-Qubits einführt, kann man allgemein auch nicht reversible Funktionen in einem Quantenschaltkreis berechnen. Dazu benötigt man zwei n-Qubit-Register. Die Funktion sei $f : \{0, 1\}^n \to \{0, 1\}^n$. Dann ist die Abbildung $U_f : |x\rangle|y\rangle \to |x\rangle|y \oplus f(x)\rangle$ unitär, also auf einem Quantencomputer ausführbar, und für $|y\rangle = |0 \ldots 0\rangle$ erhält man den Funktionswert im zweiten Register.
Bezogen auf unser Beispiel können wir die Matrix, die das NOT-Gatter darstellt, wie folgt definieren:

$$X \equiv \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{6}$$

oder in Dirac-Notation:

$$X = |0\rangle\langle 1| + |1\rangle\langle 0| \quad . \tag{7}$$

Zur Verdeutlichung schreiben wir den Zustand ϕ in Vektorschreibweise als

$$\phi = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}.$$

Wenden wir X auf diesen Vektor an, erhalten wir

$$X \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} \beta \\ \alpha \end{pmatrix}.$$

Dass X unitär ist, lässt sich leicht erkennen.
X ist eine von vier Pauli-Matrizen, die zu den wichtigsten Ein-Qubit-Quantengattern gehören:

$$I \equiv \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}; \; X \equiv \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}; \; Y \equiv \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}; \; Z \equiv \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \quad . \tag{8}$$

Die verwendete grafische Darstellung orientiert sich an [NC00] und entspricht damit der gängigen Notation in der Fachliteratur.

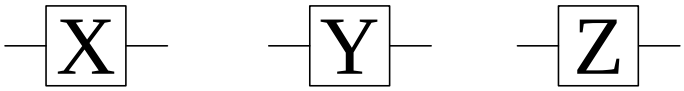


Abbildung 1: Die drei wichtigen Pauli-Matrizen in der für Quantenschaltkreise üblichen Notation

Daneben spielen noch drei weitere Matrizen eine entscheidende Rolle in der Quanteninformatik: das Hadamard-Gatter H, das Phasen-Gatter S und das $\pi/8$-Gatter T:

$$H \equiv \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}; \; S \equiv \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}; \; T \equiv \begin{pmatrix} 1 & 0 \\ 0 & \exp(i\pi/4) \end{pmatrix} \quad . \tag{9}$$

Dabei ist anzumerken, dass T historisch bedingt $\pi/8$-Gatter heißt, obwohl in der Definition $\pi/4$ auftaucht (Wenn man $e^{i\pi/8}$ aus der Matrix ausklammert, so taucht auch in der Matrix in den Exponenten $1/8$ auf).

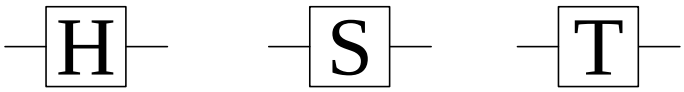


Abbildung 2: Von links nach rechts: Symbol für das Hadamard-Gatter, das Phasen-Gatter und das $\pi/8$-Gatter.

2.3.2. Mehr-Qubit-Gatter

Weil niemand einen Quantencomputer mit nur einem Qubit haben möchte, gibt es Quantengatter, die auf mehreren Qubits arbeiten. Auch diese werden durch unitäre $2^n \times 2^n$ Matrizen repräsentiert, wobei n die Anzahl der Qubits ist. Der Anfangszustand wird hierbei mathematisch durch das Tensorprodukt aller Qubits repräsentiert. Am Beispiel des CNOT-Gatters (Abbildung 3), welches dem klassischen XOR entspricht, soll das Manipulieren mehrerer Qubits verdeutlicht werden.

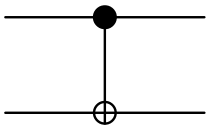


Abbildung 3: Das CNOT-Gatter. Die obere Linie repräsentiert das Kontroll-Qubit c, die untere Linie das Ziel-Qubit q. Ist c auf 1 gesetzt, wird q invertiert. Die Notation geht auf Feynman zurück [Fey85].

CNOT steht für *controlled*-NOT, was bedeutet, dass es bedingte Operationen nach dem Schema "Wenn x wahr ist, tue y" ausführt. Das CNOT-Gatter bekommt zwei Qubits als Eingänge, ein *Kontroll*-Qubit und ein *Ziel*-Qubit. Das Kontroll-Qubit wird dabei nicht verändert. Das Ziel-Qubit wird geflippt (invertiert), falls das Kontroll-Qubit auf 1 gesetzt ist. Andernfalls (Kontroll-Qubit auf 0) wird nichts verändert.

$$|00\rangle \rightarrow |00\rangle$$

$$|01\rangle \rightarrow |01\rangle$$

$$|10\rangle \rightarrow |11\rangle$$

$$|11\rangle \rightarrow |10\rangle$$

Verkürzt kann man die Wirkung des Gatters auch durch $|x, y\rangle \rightarrow |x, y \oplus x\rangle$ beschreiben. Dabei meint das Symbol $\oplus$ eine Addition modulo 2. Das CNOT-Gatter stellt somit eine Verallgemeinerung des klassischen XOR-Gatters dar. In Matrixform schreibt sich CNOT wie folgt:

$$U_{CNOT} \equiv \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

Man kann leicht zeigen, dass U_{CNOT} unitär ist, so dass die Normierungsbedingung auch hier eingehalten wird. Und wie alle Quantengatter ist U_{CNOT} natürlich reversibel, das heißt von den Ausgangszuständen kann man immer auf die Eingangszustände rückschließen.
An diesem Beispiel zeigt sich auch das Phänomen der Verschränkung. Die zwei Qubits sind nach der CNOT-Operation verschränkt, sie lassen sich nicht mehr als Tensorprodukt der Zustände der Einzel-Qubits schreiben.
Natürlich kann nicht nur die NOT-Operation bedingt ausgeführt werden, sondern jede unitäre Operation U. Man spricht dann von einer *controlled-U* Operation, bei welcher ein Kontroll-Qubit bestimmt, ob U auf das Ziel-Qubit angewandt wird.
Ein-Qubit-Gatter U lassen sich zerlegen in

$$U = \exp(i\alpha) AXBXC \tag{10}$$

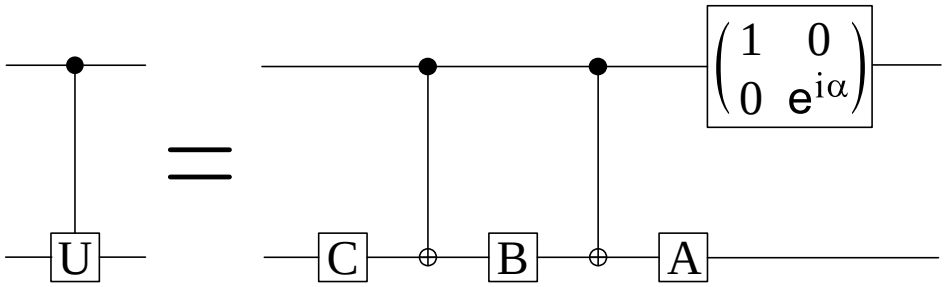


Abbildung 4: Zerlegung einer beliebigen bedingten Operation U in Ein-Qubit-Gatter A, B, C mit $ABC = I$ und CNOT-Gatter.

mit A, B und C als beliebige unitäre Operatoren, die $ABC = I$ erfüllen. Dadurch lässt sich die *controlled-U* Operation durch Ein-Qubit-Gatter und zwei CNOT-Operationen darstellen (Abbildung 4).
Dabei muss das Kontroll-Qubit nicht zwangsläufig auf 1 gesetzt sein, damit U auf dem Ziel ausgeführt wird. Genauso gut ist es möglich, dass die 0 die bedingte Operation auslöst. Für beide Möglichkeiten gibt es unterschiedliche Notationen (Abbildung 5, nach [NC00]).

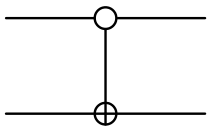
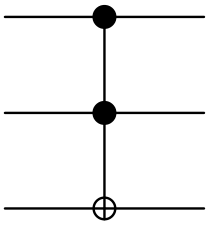


Abbildung 5: Symbol für das CNOT-Gatter, bei dem das Ziel-Qubit invertiert wird, wenn das Kontroll-Qubit auf 0 gesetzt ist.

Außerdem kann es sein, dass U von mehr als einem Kontroll-Qubit abhängt, so zum Beispiel beim Toffoli-Gatter (Abbildung 6), wo beide Kontroll-Qubits gesetzt sein müssen, damit U auf das Ziel-Qubit angewandt wird. Anders herum kann ein Kontroll-Qubit auch mehrere Ziel-Qubits steuern (Abbildung 7).

Abbildung 6: Symbol für das Toffoli-Gatter



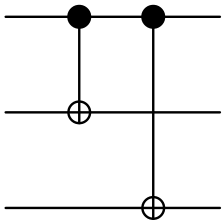


Abbildung 7: Ein Kontroll-Qubit steuert mehrere Ziel-Qubits.

2.4. Gatter als unitäre Operationen

In der klassischen Informatik betrachtet man n-Bit-Register und darauf arbeitende Abbildungen. Eine solche Abbildung ordnet jedem möglichen Zustand des Eingangsregisters einen Zustand des Ausgaberegisters zu. Da man die beiden möglichen Zustände des Bits

als *wahr* und *falsch* deuten kann, nennt man die Abbildungen logische oder Boolsche Funktionen. Sowohl Eingabe- als auch Ausgaberegister können aber nur endlich viele Zustände annehmen. Es gibt damit auch nur endlich viele Boolsche Funktionen für ein gegebenes Register.
Ein wichtiger Satz ist nun, dass es Sätze von einfachen Boolschen Funktionen gibt (normalerweise Funktionen die von einem Ein- oder Zwei-Bit-Register auf ein Ein-Bit-Register abbilden), die ausreichen, um damit alle anderen Funktionen zusammenzusetzen.
Die Nicht-Und (NAND) Funktion ist durch die Wahrheitstabelle in 1 gegeben.

Tabelle 1: Die klassische NAND-Verknüpfung

Bit 1	Bit 2		Ergebnis
0	0	$\rightarrow$	1
0	1	$\rightarrow$	1
1	0	$\rightarrow$	1
1	1	$\rightarrow$	0

Es lässt sich nun zeigen, dass allein dieses NAND-Gatter ausreicht, um alle anderen Boolschen Funktionen zu erzeugen. In dieser Hinsicht ist das NAND-Gatter also ein Erzeugendensystem der Menge aller Boolscher Funktionen.
Eine gegebene Boolsche Funktion nur durch NAND-Gatter darzustellen kann aber ungünstig sein. Welche Gatter man nutzt, um daraus größere Gatter zu bauen, spielt auch für Quantencomputer eine Rolle und wird später diskutiert.
Wie oben schon erwähnt sind die möglichen Operationen auf Qubit-Gattern genau die Operationen, die sich durch unitäre Abbildungen auf Hilberträumen darstellen lassen. Aber selbst für den Fall von Ein-Qubit-Gattern gibt es unendlich viele solcher Operationen, zumal die Menge der möglichen Zustände eines einzelnen Qubits schon überabzählbar ist.
Umso erstaunlicher ist es, dass es auch hier universelle Sätze gibt [BMP$^+$99], aus denen sich jeweils alle möglichen Gatter erzeugen lassen. Dabei muss man allerdings gewisse Einschränkungen in Kauf nehmen.
Die wichtigste Überlegung ist zunächst, dass sich jede unitäre Operation als Verkettung von einfachen unitären Operationen schreiben lässt. Dabei meint *einfach*, dass die Operation maximal zwei Qubits verändert. Um also Operationen auf beliebig vielen Qubits durchführen zu können, braucht man nach dieser Überlegung nur alle Ein- und Zwei-Qubit-Operationen. Dies sind aber immer noch unendlich viele.
Die zweite Überlegung ist, dass man nur ein Zwei-Qubit-Gatter braucht, quasi um Informationen zwischen Qubits auszutauschen, und ansonsten nur alle weiteren Ein-Qubit-Gatter. Dazu kann man aber nicht jedes Zwei-Qubit-Gatter verwenden. Es muss ein

Zwei-Qubit-Gatter sein, welches sich nicht als Tensorprodukt von Ein-Qubit-Gattern darstellen lässt. Leider hat man damit immer noch unendlich viele Gatter.
Nun gibt es zwei Möglichkeiten der Vereinfachung. Die erste stützt sich darauf, dass sich jede unitäre 2×2-Matrix (bis auf eine Phase) exakt durch die Verkettung von Rotationsmatrizen darstellen lässt [NC00]. Der Winkel, um den bei jeder Rotation gedreht wird, ist dabei ein Parameter. Wenn man also Gatter für die Rotationen sowie die Phase hat, die jeweils durch einen kontinuierlichen Parameter anpassbar sind, so hätte man zumindest im Prinzip endlich viele Gatter. Aber natürlich ist ein kontinuierlicher Parameter in der Realität nur näherungsweise möglich. Je nach Genauigkeit wird eine beliebige unitäre Operation also nur approximiert.
Die zweite Möglichkeit ist, einen geschickten Satz von festen, also parameterunabhängigen Gattern zu nutzen. Dieser lässt sich so wählen, dass sich jede beliebige Ein-Qubit-Operation mit beliebiger Genauigkeit approximieren lässt. Ob sich aber für jede Operation die gewünschte Genauigkeit effizient, also mit polynomiell anwachsendem Ressourcenaufwand, erreichen lässt, ist dadurch nicht gesagt.
Ein solcher Satz ist zum Beispiel $\{H, T, CNOT\}$ mit CNOT als Zwei-Qubit-Gatter. Die Vollständigkeit dieses Gatter-Satzes wurde in [BMP+99] bewiesen.

2.5. Schaltkreise

So wie es für klassische Computer grafische Notationen zum Beschreiben von Schaltkreisen oder Gattern gibt, so benutzt man eine äquivalente Notation für Quantencomputer, die auf Feynman [Fey85] zurückgeht. Auch hier gibt es "Drähte" und logische Gatter, welche die durch den Schaltkreis transportierten Informationen verändern. Ein Beispiel für eine solche grafische Repräsentation eines Quantenschaltkreises ist in Abbildung 8 zu sehen. Es handelt sich dabei um einen Schaltkreis, der mit zwei Qubits arbeitet und deren Zustände vertauscht,

$$|ab\rangle \rightarrow |ba\rangle \,.$$

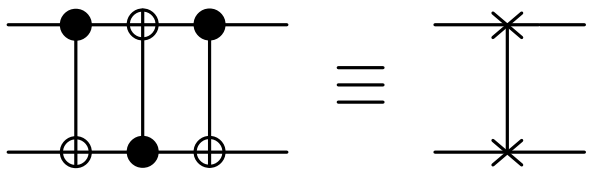


Abbildung 8: Quantenschaltkreis, der zwei Qubits vertauscht, mit einer alternativen Notation

Entsprechend gibt es zwei horizontale Linien, die die "Drähte" darstellen sollen. Wir benutzen die Anführungszeichen um zu verdeutlichen, dass dies keine Drähte im herkömmlichen physikalischen (oder elektrotechnischen) Sinn sein müssen. Durch die Linien soll vielmehr veranschaulicht werden, dass sich dort eine Information in dem Schaltkreis entwickelt bzw. verändert. Es könnte zum Beispiel ein zeitlicher Verlauf dargestellt sein oder die Bewegung eines Teilchens (z. B. eines Photons), das ein Qubit repräsentiert.

Die Abbildung, oder besser gesagt der Schaltkreis, muss von links nach rechts gelesen werden. Links ist also der Anfangszustand (Input) und rechts der Endzustand (Output). Der Anfangszustand kann links angegeben sein, muss es aber nicht. Bei vielen Quantenschaltkreisen ist der Anfangszustand $|00\cdots0\rangle$, da es zu den Forderungen an einen Quantencomputer gehört [Div95], dass er in diesem Zustand initialisierbar ist.
Unser Beispiel zeigt drei CNOT-Operationen auf den zwei Qubits. Das Kontroll-Qubit ist dabei durch den dicken schwarzen Punkt angedeutet, das Ziel-Qubit dementsprechend durch den Kreis, der das Plus-Zeichen umschließt. Wie im Abschnitt 2.3.2 bereits erwähnt, lassen sich beliebige unitäre Operationen bedingt ausführen. Die Notation ändert sich dabei nicht, man ersetzt lediglich das eingekreiste Plus-Zeichen durch einen quadratischen Kasten mit dem Buchstaben 'U' darin (Abbildung 9). Ein CNOT ist also nur ein Spezialfall mit $U = X$.

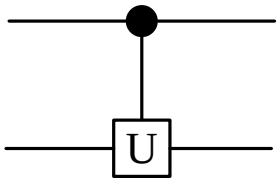


Abbildung 9: Symbol für eine beliebige bedingte unitäre Operation U

Darüberhinaus gibt es ein letztes wichtiges Symbol in Quantenschaltkreisen, nämlich das für die Messung. Diese macht aus einem Qubit ein klassisches Bit. War das Qubit vor der Messung im Zustand $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$, dann ist das klassische Bit nach der Messung mit der Wahrscheinlichkeit $|\alpha|^2$ im Zustand 0 und mit der Wahrscheinlichkeit $|\beta|^2$ im Zustand 1. Zur Verdeutlichung, dass es sich um ein klassisches Bit handelt, werden zwei horizontale Linien gezeichnet, die aus dem Messoperator hinausführen (Abbildung 10). Da die Messung die quantenmechanischen Informationen zerstört und somit nicht reversibel ist, steht sie im Allgemeinen am Ende eines Quantenschaltkreises.
Dazu sind zwei Prinzipien wichtig: Das *Prinzip der aufgeschobenen Messung* sagt, dass man Messungen immer erst am Ende durchführen kann, anstatt sie innerhalb des Quantenschaltkreises durchzuführen. Sollen Messinformationen den weiteren Verlauf des Schaltkreises bestimmen, so kann man diesen (klassischen) Einfluss durch kontrollierte Quantengatter ersetzen.
Das *Prinzip der impliziten Messung* besagt, dass jedes Qubit am Ende eines Schaltkreises, welches nicht explizit gemessen wird oder werden muss, dennoch als gemessen angenommen werden kann.
Einige aus klassischen Schaltkreisen bekannten Eigenschaften können in Quantenschaltkreisen nicht benutzt werden, weil wir immer beachten müssen, dass die Operationen reversibel sein müssen. Eine Anweisung wie "$a = 5$" ist nicht möglich, da wir danach nicht mehr auf den ursprünglichen Wert der Variable a schließen können. Im Gegensatz dazu ist die Anweisung "$a = a + 5$" erlaubt, weil wir hier aus dem Wert nach der Addition den Anfangswert rekonstruieren können.

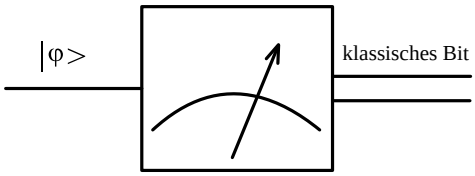


Abbildung 10: Symbol für einen Messvorgang am Quantenschaltkreis

Desweiteren verbietet das *no-cloning Theorem* das Kopieren von Qubits [Die82], während das in klassischen Schaltkreisen an der Tagesordnung ist.

2.6. Algorithmen

Die ersten Entwicklungen im Bereich der Algorithmik liegen weit in der Vergangenheit: Keilschriftfunde belegen, dass schon etwa 1750 v. Chr. bei den Babyloniern fortgeschrittene Algorithmen bekannt waren. Doch erst 1936 wurde dieses Gebiet auf ein festes Fundament gestellt, als Alan Turing die abstrakte Turingmaschine vorschlug.
In seiner Arbeit zeigte er, dass es eine universelle Turingmaschine gibt, die jede andere Turingmaschine simulieren kann. Zusätzlich behauptete er, dass die universelle Turingmaschine exakt das berechnen kann, was mit algorithmischen Prinzipien berechenbar ist (Church-Turing-These). Das bedeutet, dass wenn ein Algorithmus auf irgendeinem Gerät berechenbar ist, dann sollte es einen äquivalenten Algorithmus für die universelle Turingmaschine geben.
Beobachtungen an analogen Computern und an probabilistischen Systemen führten zu einigen Modifikationen der Church-Turing-These, aber erst David Deutsch [Deu85, Deu89] schaffte es, eine Formulierung zu finden, die nicht nur auf Beobachtungen basierte, sondern sich auf physikalische Theorien stützt.
Dazu ersann er einen Computer, der beliebige physikalische Systeme effizient simulieren kann, indem er quantenmechanische Prinzipien nutzt. Dieses Konzept eines Quantencomputers, eine Art Quantenanalogon zur Turingmaschine, könnte prinzipiell Algorithmen ermöglichen, die weder auf einer normalen noch auf einer probabilistischen Turingmaschine effizient berechnet werden können, aber auf einem Quantencomputer schon.
Er schlug auch einen Algorithmus vor, der schneller funktioniert als jeder bekannte klassische Algorithmus. Doch erst die Entdeckung des Shor-Algorithmus [Sho94], der anders als der Deutsch-Josza-Algorithmus praktische Relevanz hat, führte zu allgemeinem Interesse an dem Konzept des Quantencomputers.
Es ist jedoch problematisch, definitive Aussagen über Algorithmen zu treffen. Denn obwohl man keinen klassischen Algorithmus kennt, der so effizient ist wie der Deutsch-Josza-Algorithmus, ist nicht für alle Zeiten ausgeschlossen, dass irgendwann einer gefunden wird. Bisher ist weder die Existenz noch die Nicht-Existenz bewiesen.
Man kann auch nicht sagen, dass es Quantenalgorithmen in großer Anzahl gibt oder regelmäßig neue gefunden werden. Denn ein sinnvoller Quantenalgorithmus muss sich

der sehr anti-intuitiven Quantenmechanik bedienen und zudem noch besser sein als jeder bekannte klassische. Niemand weiß, für welche Art von Problemen sich eine solche Suche lohnt, zumal auch niemand weiß, ob das Konzept des Quantencomputers dem Konzept der Turingmaschine wirklich überlegen ist.
Fest steht nur, dass zum Zeitpunkt der Erstellung dieser Arbeit im Wesentlichen drei Bereiche bekannt sind, in denen Quantencomputer Vorteile gegenüber aktuellen klassischen Verfahren bieten: das Problem der versteckten Untergruppe [Sim94, Sim97], das Problem der Suche in unstrukturierten Daten [Gro97] und die Simulation von quantenmechanischen Systemen.
Das Problem der versteckten Untergruppe besteht darin, ein Erzeugendensystem einer Untergruppe (die versteckte Untergruppe) zu finden, auf der eine bestimmte Funktion konstant ist. Diese sehr abstrakte Formulierung umfasst sehr viele Probleme, unter anderem die Faktorisierung von natürlichen Zahlen (Shor-Algorithmus). Bei diesen Problemen kann eine Verbesserung von exponentieller zu polynomieller Komplexität erreicht werden. Komplexität beschreibt den Aufwand an Zeit und Hardware, den ein Algorithmus für seine Ausführung benötigt. Die Komplexitätsklasse beschreibt, wie dieser Aufwand von der Bitbreite der Eingabe abhängt.
Das Problem der Suche in unstrukturierten Daten wird durch den Grover-Algorithmus gelöst, der sich eines *Orakels*[1] bedient. Dabei wird die klassische lineare Komplexität zu Quadratwurzel-Komplexität verbessert. Es wurde gezeigt, dass bei einem Orakelbasiertem Algorithmus keine Verbesserung über diese Komplexität hinaus möglich ist [BBBV97].
Bei der Simulation von quantenmechanischen Systemen besteht das Problem, dass noch nicht bewiesen wurde, ob sich wirklich jedes System simulieren lässt. Für die Systeme, für die es aber schon gezeigt wurde, ergibt sich eine exponentielle Verbesserung des Ressourcenaufwandes, da die Dimension der Qubit-Hilberträume genauso schnell wächst wie die Dimension von zu simulierenden Systemen.

2.7. Physikalische Realisierung von Quantencomputern

Die Quanteninformatik ist als theoretisches Themengebiet schon enorm spannend und lehrreich, da sie eine tiefgreifende Verknüpfung von Physik und klassischer Informatik aufzeigt. Dennoch stellt sich natürlich die Frage, ob sich ein Quantencomputer tatsächlich realisieren lässt, und wenn ja, wie.
Da es die verschiedensten quantenmechanischen Systeme gibt, gibt es entsprechend viele Ansätze für die Realisierung von Quantencomputern. Einen Überblick liefert Tabelle 2 aus [NC00].
Um als Quantencomputer geeignet zu sein verlangt man im Allgemeinen fünf Eigenschaften von einem System, die auf [Div95] zurückgehen:

[1] Begriff aus der Informationstheorie. Ein Orakel ist eine Black Box, die eine Funktion in einem Schritt berechnet.

Tabelle 2: Übersicht über verschiedene Ansätze zur Realisierung von Quantencomputern. Aus: [NC00], Seite 278. Legende: Dekohärenzzeit τ_Q in Sekunden, Operationszeit τ_{Op} in Sekunden, maximale Anzahl von Operationen n_{Op}, geschätzte Werte

System	τ_Q	τ_{Op}	$n_{Op} = \frac{\tau_Q}{\tau_{Op}}$
Kernspin	$10^{-2} - 10^{8}$	$10^{-3} - 10^{-6}$	$10^{5} - 10^{14}$
Elektronenspin	10^{-3}	10^{-7}	10^{4}
Ionenfalle	10^{-1}	10^{-14}	10^{13}
Elektronen-Au	10^{-8}	10^{-14}	10^{6}
Elektronen-GaAs	10^{-10}	10^{-13}	10^{3}
Quanten-Punkt	10^{-6}	10^{-9}	10^{3}
Optische Kavität	10^{-5}	10^{-14}	10^{9}
Mikrowellenkavität	10^{0}	10^{-4}	10^{4}

- **Qubits**: Das System muss in der Lage sein, Quanteninformation zu speichern, das bedeutet Information mit der Möglichkeit zur Superposition und Verschränkung. Dazu ist vor allem wichtig, dass der Hilbertraum des Systems endlich ist, damit man die einzelnen Zustände unterscheiden kann. Auch muss die Dekohärenzzeit lang sein, damit möglichst viele Operationen durchführbar sind, bevor die Quanteninformation verloren geht.

- **Gatter**: Es muss möglich sein, die Zeitentwicklung des Systems zumindest so genau steuern zu können, dass die elementaren Gatteroperationen (H, T, CNOT) ausgeführt werden können. Damit lässt sich jede Zeitentwicklung approximieren, also jede Quantenrechnung durchführen.

- **Anfangszustand**: Das System muss sich in einen definierten Anfangszustand bringen lassen.

- **Messung**: Die Messung des Endzustands muss die möglichen Ergebnisse mit den quantenmechanischen Wahrscheinlichkeiten liefern. Denn nur so lassen sich Algorithmen, die probabilistisch arbeiten, sinnvoll ausführen.

- **Skalierbarkeit**: Die Schwierigkeit, Gatter und Messungen durchzuführen, darf nicht zu stark von der Anzahl der Qubits abhängen, da ansonsten die Vorteile der Quantenalgorithmen verloren gehen.

Im Laufe der letzten Jahre haben sich im Wesentlichen drei Ansätze als erfolgsversprechend herausgestellt, die im Folgenden näher erläutert werden: Quantenoptische Systeme, Kernmagnetische Resonanz (NMR) und Festkörpersysteme.

2.7.1. Quantenoptische Systeme

Bei diesen Systemen werden die Qubits durch Atome oder Ionen repräsentiert. Die beiden Basiszustände der Qubits sind zwei Energieniveaus oder Spinzustände. Durch die präzise Nutzung von Lasern werden diese Zuständen manipuliert und damit Gatteroperationen implementiert. Messungen werden durch Laser und anschließende Fluorenszensmessung realisiert.
Diese Systeme erreichen eine enorme Präzision. Quantenmechanische Effekte lassen sich sehr gut beobachten, weil die Wechselwirkung mit der Umgebung auf ein Minimum reduziert ist. In vielen Experimenten (zum Beispiel mit Ionenfallen [CZ95]) wurden erfolgreich Gatteroperationen mit mehreren Qubits demonstriert.
Das große Problem bei diesen Systemen ist, dass sie nur sehr schlecht skalierbar sind, da die verschiedenen Energieniveaus immer dichter zusammenrücken und nur noch sehr schwer unterschieden werden können. Rechnungen mit mehr als nur ein paar Qubits sind noch nicht durchgeführt worden.
Einen Ausweg bieten so genannte Quantennetzwerke [KMW02], die verschiedene kleine Einheiten mit wenigen Qubits so verbinden, dass das ganze wie ein großes System mit vielen Qubits wirkt.

2.7.2. Kernmagnetische Resonanz

In diesem Verfahren werden Qubits durch den Kernspin von Atomen in großen Molekülen repräsentiert. Man verwendet Kernspins mit $s = \frac{1}{2}$, so dass der Hilbertraum zweidimensional ist. Durch Radiowellen bestimmter Frequenz und Dauer kann man die Kernspins manipulieren und auf diese Weise den Qubitzustand verändern [Vin95]. Die dafür notwendigen Apparate sind (z. B. in der Medizin und Chemie) bereits vorhanden und ausgereift. Normalerweise betrachtet man ein ganzes System von Molekülen (Größenordnung 10^{25}) und nicht nur ein einzelnes.
Durch starke Magnetfeldpulse werden die Spins manipuliert (NMR) und damit Gatteroperationen implementiert. Durch entsprechende Vorgänge können die Spins wieder ausgelesen, also gemessen werden.
Der Anfangszustand muss in einem NMR-System nicht unbedingt ein reiner Zustand sein. Statt dessen kann es auch ein "pseudoreiner" Zustand sein. Das bedeutet gemischt, aber mit starken Einschränkungen, auf die in [CFH97] eingegangen wird.
Dadurch ist der Endzustand im Allgemeinen kein reiner Zustand. Messungen an einem gemischten Zustand liefern aber nur Mittelwerte, was bei bestimmten Anwendungen (zum Beispiel bei der Kettenbruchentwicklung beim Shor-Algorithmus) zu Problemen führt. Prinzipiell ist das Auslesen, auch praktisch, möglich.
Die Messung wird zudem immer schwieriger, je mehr Qubits man hat. Daher ist eine Skalierung nur sehr begrenzt möglich.

2.7.3. Festkörpersysteme

Bei diesem System betrachtet man Elektronenpaare (Cooperpaare [NPT99] oder Quantenpunkte [LD98]) in Halbleitern. Ein Qubit ist ein solches Paar. Durch kontrollierte Coulomb-Wechselwirkungen (und andere Effekte) können Operationen auf Qubits durchgeführt werden und Messungen vorgenommen werden (heutige Feldeffekttransistoren können die Bewegung einzelner Ladungen detektieren).
Dieser Ansatz ist prinzipiell vielversprechend, da ein solches System sehr gut skalierbar wäre. Hinzu kommt, dass die Mikroelektronik schon durch die Entwicklung bei klassischen Computern sehr ausgereift ist. Nachteilig ist jedoch, dass eine Abkopplung von der Umgebung prinzipiell nicht möglich ist, da sich das Geschehen in einem Festkörper abspielt. Damit ist die Dekohärenzzeit sehr kurz.

2.7.4. Zusammenfassung

Die betrachteten Ansätze haben alle vielversprechende Eigenschaften, aber auch teils gravierende praktische Nachteile, die bis heute die Entwicklung eines brauchbaren Quantencomputers verhindert haben. Brauchbar meint, dass ausreichend viele Qubits vorhanden sind, um komplexere Aufgaben zu verrichten, und dass das System skalierbar ist.
Im Jahr 2001 hat IBM einen 7-Qubit-Quantencomputer auf der Basis von NMR vorgestellt [VSB+01]. Dieses System konnte den Shor-Algorithmus für die Zahl 15 ausführen. Bis heute ist es nicht gelungen, ein System mit wesentlich mehr Qubits zu konstruieren, auf dem noch sinnvoll gerechnet werden kann.
Auf Grund der offensichtlichen Schwierigkeiten vertreten viele Wissenschaftler die Meinung, dass ein nutzbarer Quantencomputer nicht möglich ist und damit niemals gebaut wird. Andere Wissenschaftler lassen sich eher von den bisherigen Erfolgen beflügeln und meinen, dass in absehbarer Zukunft ein Quantencomputer gebaut werden kann, zumal es keine prinzipiellen Hindernisse gibt, nur praktische. Ob es allerdings so weit kommt, dass jeder, der heute einen klassischen Computer auf dem Schreibtisch stehen hat, zukünftig einen Quantencomputer besitzt, ist mehr als fraglich. Die Stärken von Quantencomputern liegen eindeutig im wissenschaftlichen Bereich, wo die enorme Leistungsfähigkeit zum Tragen kommt. Klassische Computer werden ihre heutigen Aufgaben weitestgehend behalten, nur in Spezialbereichen wie der Faktorisierung oder der Datenbanksuche, wo Quantencomputer grundlegend schneller sind, werden diese die klassischen Rechner ersetzen. Ein "Quantencomputer von Aldi" scheint aus heutiger Sicht unrealistisch. Allerdings dachte man so auch über klassische Computer:

"I think there is a world market for maybe five computers."
Thomas Watson, Vorsitzender von IBM, 1943

3. Quanten-Fourier-Transformation

Ein wesentliches Element im Shor-Algorithmus zur Faktorisierung natürlicher Zahlen ist die Quanten-Fourier-Transformation. Dabei werden ähnlich wie im klassischen Fall Periodizitäten aufgezeigt. Eine vollständige Analogie zum klassischen Fall ist jedoch nicht gegeben, da quantenmechanische Amplituden transformiert werden. Die Transformation klassischer Daten lässt sich mit diesem Verfahren nicht beschleunigen (ansonsten wäre der Quantencomputer in unglaublich vielen Bereichen besser als ein klassischer Computer). Man nennt die hier behandelte Transformation Quanten-Fourier-Transformation, um den Unterschied deutlich zu machen.

3.1. Definition

Die Quanten-Fourier-Transformation QFT ist eine Abbildung von einem N-dimensionalen Hilbertraum in sich selbst. Dies lässt sich erstmal mit der Definition der klassischen Fourier-Transformation vergleichen: Auch hier ist die Transformation eine Abbildung von einem N-dimensionalen Vektorraum in sich selbst. Die Komponenten y_k des Ergebnisvektors hängen über die folgende Definition von den Komponenten x_k des Eingangsvektors ab:

$$y_k := \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{2\pi i j k/N} \quad . \tag{11}$$

Die inverse Fourier-Transformation unterscheidet sich nur durch ein Minuszeichen im Exponenten. Welche der beiden Transformationen man als Fourier-Transformation und welche als inverse Fourier-Transformation bezeichnet, ist daher egal.
Die Quanten-Fourier-Transformation definiert man analog durch ihre Wirkung auf eine Basis des Hilbertraumes:

$$|j\rangle \rightarrow \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{2\pi i j k/N} |k\rangle \quad . \tag{12}$$

Insgesamt wird also ein beliebiger Zustand (mit Komponenten x_j) auf die folgende Art und Weise transformiert:

$$\sum_{j=0}^{N-1} x_j |j\rangle \rightarrow \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} y_k |k\rangle \quad . \tag{13}$$

In einem Quantencomputer ist der Hilbertraum im Allgemeinen eine Potenz von 2. Dies ermöglicht eine Darstellung [NC00] der Fouriertransformation, an der sich die Realisierung als Quanten-Schaltkreis mehr oder weniger direkt ablesen lässt.

Sei also jetzt $N = 2^n$ für eine natürliche Zahl n. Die Basiszustände seien mit $|a\rangle$ ($a = 0, \ldots, 2^n - 1$) bezeichnet. Mit a_i seien die Ziffern von a in binärer Darstellung bezeichnet. Dann gilt:

$$\sum_{k=0}^{2^n-1} e^{2\pi ijk/2^n} |k\rangle \tag{14}$$

$$= \underbrace{\sum_{k_1=0}^{1} \cdots \sum_{k_n=0}^{1}}_{\text{n Summen}} e^{2\pi ij(\sum_{l=1}^{n} k_l 2^{-l})} |k_1 \cdots k_n\rangle \tag{15}$$

$$= \underbrace{\sum_{k_1=0}^{1} \cdots \sum_{k_n=0}^{1}}_{\text{n Summen}} \bigotimes_{l=1}^{n} e^{2\pi ijk_l 2^{-l}} |k_l\rangle \tag{16}$$

$$= \bigotimes_{l=1}^{n} \left[\sum_{k_l=0}^{1} e^{2\pi ijk_l 2^{-l}} |k_l\rangle \right] \tag{17}$$

$$= \bigotimes_{l=0}^{n} \left[|0\rangle + e^{2\pi ij2^{-l}} |1\rangle \right] \tag{18}$$

$$= \left(|0\rangle + e^{2\pi ij_n/2} |1\rangle\right) \left(|0\rangle + e^{2\pi i(j_{n-1}/2 + j_n/4)} |1\rangle\right) \ldots \left(|0\rangle + e^{2\pi i(j_1/2 + \ldots j_n/2^n)} |1\rangle\right) \tag{19}$$

Statt der Summe in Gleichung 12 kann man Ausdruck 19 einsetzen. Die Quanten-Fourier-Transformation sieht dann so aus:

$$|j\rangle \rightarrow \frac{\left(|0\rangle + e^{2\pi ij_n/2} |1\rangle\right) \left(|0\rangle + e^{2\pi i(j_{n-1}/2 + j_n/4)} |1\rangle\right) \ldots \left(|0\rangle + e^{2\pi i(j_1/2 + \ldots j_n/2^n)} |1\rangle\right)}{2^{n/2}} \; . \tag{20}$$

3.2. Realisierung

Mit Darstellung 20 lässt sich nun ein Quantenschaltkreis entwerfen. Der hier vorgestellte Schaltkreis besteht aus zwei Arten von Gattern: dem Hadamard-Gatter und den kontrollierten R_k-Gattern. Letztere sind Ein-Qubit-Gatter mit der Matrixdarstellung

$$R_k := \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i/2^k} \end{pmatrix} \; , \tag{21}$$

welche wie in Abschnitt 2.3.2 beschrieben nur dann angewendet werden, wenn das kontrollierende Bit gesetzt ist.

Anstatt zu versuchen, die Transformation direkt durch die Gatter zu implementieren, beschränkt man sich zunächst darauf, die Koeffizienten der Qubits des Transformationszustandes prinzipiell herzustellen. Es zeigt sich, dass es einfacher ist, diese Koeffizienten in verkehrter Reihenfolge herzustellen. Denn Qubits, die noch als Kontrollqubit für kontrollierte R_k-Gatter fungieren müssen, dürfen vorher noch durch kein Gatter verändert worden sein, ansonsten transformiert man nicht den Originalzustand. Hinterher stellt man durch SWAP-Operationen die richtige Reihenfolge her.
Das letzte Qubit in Darstellung 20 hat die $|1\rangle$-Komponente $e^{2\pi i(j_1/2+\dots j_n/2^n)}$. Dies entspricht der Anwendung des Hadamard-Gatters und aller kontrollierten R_k-Gatter ($k = 2, \dots, n$), wobei das k-te Bit das k-te Gatter kontrolliert.
Analog lässt sich die $|1\rangle$-Komponente des vorletzten Qubits in Darstellung 20 herstellen. Dabei läuft k aber nur von 2 bis $n-1$. Entsprechend ergeben sich die Gatter für alle übrigen Qubits. Auf das letzte Qubit schließlich wirkt nur ein Hadamard-Gatter.
Abbildung 11 zeigt den schematischen Aufbau. Die Swap-Operation am Ende kehrt die Reihenfolge der Qubits genau um (1 vertauscht mit n, 2 vertauscht mit $n-1$, ...).

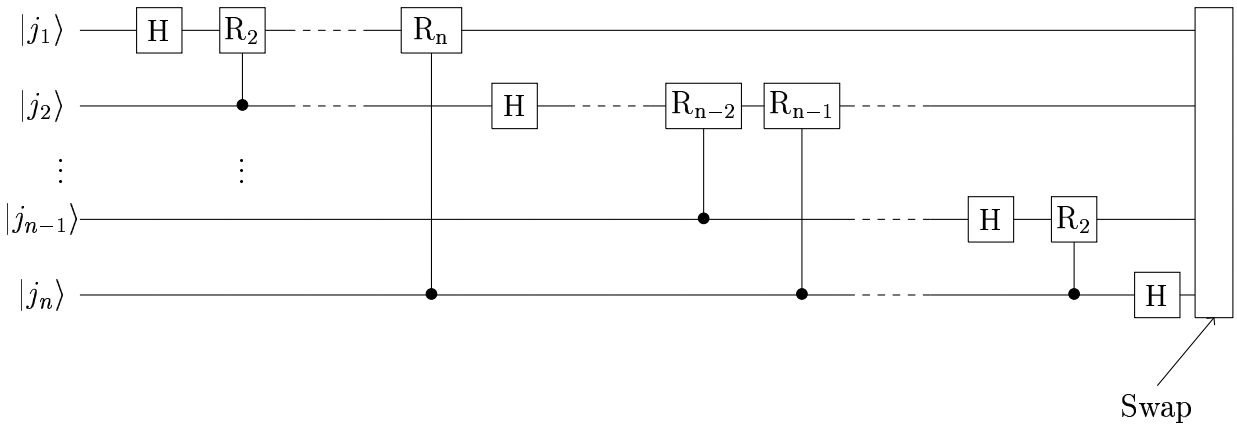


Abbildung 11: Quantenschaltkreis zur Berechnung der Quanten-Fourier-Transformation

Um sich die genaue Funktionsweise des Schaltkreises klar zu machen, betrachtet man die Wirkung auf einen beliebigen Basiszustand $|a\rangle$. Wieder bezeichnet a_i die i-te Ziffer in der Binärdarstellung von a. Man kann also auch schreiben

$$|a\rangle \;=\; |a_1 a_2 \dots a_n\rangle \quad . \tag{22}$$

Das Hadamard-Gatter transformiert den Zustand $|0\rangle$ in $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ und den Zustand $|1\rangle$ in $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. Daher kann man auch schreiben, dass das Qubit $|a_i\rangle$ in den Zustand $\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i a_i/2}\,|1\rangle$ transformiert wird.

Ein Hadamard-Gatter mit nachgeschaltetem kontrolliertem R_2-Gatter liefert insgesamt die Transformation

$$|a\rangle \to \frac{1}{\sqrt{2}} \left(|0\rangle + e^{2\pi i(a_1/2+a_2/4)} |1\rangle\right) |a_2 \dots a_n\rangle \quad . \tag{23}$$

Nun folgt leicht die Wirkung zusammen mit den weiteren kontrollierten R_k-Gattern:

$$|a\rangle \to \frac{1}{\sqrt{2}} \left(|0\rangle + e^{2\pi i(a_1/2+a_2/4+\dots+a_n/2^n)} |1\rangle\right) |a_2 \dots a_n\rangle \quad . \tag{24}$$

Ein entsprechendes "Paket" von Gattern transformiert auch die anderen Qubits in der gewünschten Weise. Nach den abschließenden SWAP-Operationen ist genau die Transformation 20 implementiert, also die QFT. Die ausschließliche Verwendung von unitären Gattern beweist, dass die QFT eine unitäre Transformation ist. Ein Quantencomputer kann sie also berechnen.

3.3. Komplexität

In der Informatik interessiert man sich für die Komplexität von Algorithmen. Es gibt viele Methoden, Aussagen über die Komplexität eines Algorithmus' zu machen. Eine weit verbreitete Technik ist, das asymptotische Laufzeitverhalten des Algorithmus' als Funktion der Bitbreite des Inputs auszudrücken.
Wenn die Laufzeit proportional zur Bitbreite des Inputs ist, so sagt man, dass der Algorithmus in der *Komplexitätsklasse* $\Theta(n)$ ist.
Mit dieser Technik lässt sich die Komplexitätsklasse der QFT ermitteln: Auf das erste Qubit wirken das Hadamard-Gatter und alle $(n-1)$ R_k-Gatter. Auf das zweite Qubit wirken das Hadamard-Gatter und alle R_k-Gatter bis auf R_n. Entsprechend ergibt sich eine Gesamtzahl von $n + (n-1) + \dots + 1 = n(n+1)/2$ Gattern. Schließlich müssen noch $n/2$ SWAP-Operationen durchgeführt werden. Jede SWAP-Operation lässt sich zum Beispiel durch drei CNOT-Gatter implementieren (vgl. Abbildung 8). Die Gesamtzahl an Gattern ergibt sich damit zu $n^2/2 + 2n$. Asymptotisch wächst die Gesamtzahl an Gattern quadratisch, die Komplexitätsklasse ist $\Theta(n^2)$.
Auf der Grundlage von Komplexitätsklassen wird der Begriff *Effizienz* definiert: Ein Algorithmus heißt effizient, wenn er in der Komplexitätsklasse $\Theta(g(n))$ liegt und g eine Polynomfunktion ist. Ansonsten heißt der Algorithmus *nicht effizient*.
In diesem begrifflichen Rahmen ist die QFT also effizient. Wie sieht es mit klassischen Algorithmen aus? Es zeigt sich, dass der schnellste bekannte klassische Algorithmus zur diskreten Fourier-Transformation die schnelle Fourier-Transformation (Fast Fourier Transform, FFT) ist. Die Komplexitätsklasse dieses Verfahrens ist $\Theta(n2^n)$. Die Laufzeit wächst also exponentiell an. Die Fourier-Transformation ist nach heutiger Kenntnis also nicht effizient auf einem klassischen Computer. Es existiert aber kein Beweis, dass wirklich kein Algorithmus existiert, der die Fourier-Transformation effizient auf einem klassischen Computer berechnet.

Zusammenfassend lässt sich sagen, dass der vorgestellte Schaltkreis die Quanten-Fourier-Transformation auf einem Quantencomputer effizient berechnet. Da sich der Anfangszustand nicht mit beliebiger Genauigkeit effizient einstellen lässt und das Ergebnis der Transformation nicht vollständig ausgelesen werden kann, ist die QFT kein Ersatz für die klassische Fourier-Transformation. Probleme, bei denen es ausreicht, den Basiszustand zu ermitteln, der die höchste Amplitude bei der QFT erhält, lassen sich aber verbessert lösen. So nutzt der Shor-Algorithmus die QFT zur Faktorisierung von natürlichen Zahlen.

4. Shor-Algorithmus

Die Bezeichnung "Shor-Algorithmus" ist nicht eindeutig, denn Shors bahnbrechende Veröffentlichung [Sho94] enthielt zwei Algorithmen: einen Algorithmus zur Faktorisierung von ganzen Zahlen und einen zur Bestimmung von diskreten Logarithmen. In dieser Arbeit wird nur der erste, also der Faktorisierungsalgorithmus behandelt.
Aus der Zahlentheorie ist bekannt, dass jede natürliche Zahl das Produkt von Primzahlen ist. Die Darstellung einer Zahl als Primzahlenprodukt ist zudem eindeutig. Die Aufgabe ist nun, diese Faktoren einer Zahl herauszufinden. Bei der Faktorisierung lässt man aber nicht nur Primfaktoren zu, sondern sucht nach zwei beliebigen Faktoren. Diese Faktoren können dann weiter faktorisiert werden, bis am Ende nur noch Primzahlen übrig bleiben. Gesucht sind also die Faktoren p und q einer Zahl $N = pq$. Die Idee von Shor ist nun, das Problem auf ein schon gelöstes zurückzuführen: die Quanten-Fourier-Transformation. Denn mit Hilfe dieser Transformation kann man die so genannte Phasenabschätzung (Abschnitt 4.2) effizient implementieren. Dies ermöglicht es, die Ordnung r einer Zahl x bezüglich N zu bestimmen (Abschnitt 4.3). Und mit dieser Information lassen sich mit hoher Wahrscheinlichkeit die gesuchten Faktoren von N herausfinden (Abschnitt 4.4).

4.1. Zahlentheorie

Um den Shor-Algorithmus verstehen und genau analysieren zu können, ist einiges an Zahlentheorie notwendig. Einige zentrale Beweise werde ich hier ausführen, doch für ein tieferes Eindringen in die Materie sei auf [HW60] verwiesen. Die hier vorgestellten Beweise orientieren sich im Wesentlichen an [Pit99].
Sei nun $\mathrm{ggT}(p,q)$ der größte gemeinsame Teiler von den beiden natürlichen Zahlen p und q und $\mathrm{kgV}(p,q)$ ihr kleinstes gemeinsames Vielfaches.

Satz 1 Chinesischer Restsatz*: Sei* $\mathrm{ggT}(p_1,p_2) = 1$, $0 \leq a < p_1$ *und* $0 \leq b < p_2$. *Dann gibt es ein genau bestimmtes* $0 \leq x < p_1p_2$, *so dass* $x = a(\mathrm{mod}\, p_1)$ *und* $x = b(\mathrm{mod}\, p_2)$.

Beweis: Sei p_i^{-1} das multiplikative Inverse von p_i bezüglich p_{1-i}. Beide p_i^{-1} existieren, da nach Voraussetzung $\mathrm{ggT}(p_1,p_2) = 1$. Sei nun $x := ap_1p_1^{-1} + bp_2p_2^{-1}$. Man sieht leicht, dass $p_ip_i^{-1} = 1\,(\mathrm{mod}\, p_i)$ und $p_ip_i^{-1} = 0\,(\mathrm{mod}\, p_j)$ für $i \neq j$. Dies beweist die Existenz.

Sei nun x' eine weitere Lösung für die Gleichungen. Es folgt, dass $x - x' = 0 \,(\mathrm{mod}\, p_i)$ für jedes i. Und da die p_i nach Voraussetzung teilerfremd sind, teilt das Produkt $M := p_1 p_2$ die Differenz $x - x'$. Also ist $x = x' \,(\mathrm{mod}\, M)$. Im Bereich $0 \leq x < p_1 p_2$ ist x also genau bestimmt. q.e.d.
Demnach kann man jede Zahl $x(\mathrm{mod}\, pq)$ durch $a(\mathrm{mod}\, p)$ und $b(\mathrm{mod}\, q)$ angeben. Dies kann man auch in der Wahrscheinlichkeitstheorie deuten: Wenn $\{x : 0 \leq x < pq\}$ gleichverteilt ist, dann sind $x \,(\mathrm{mod}\, p)$ und $x \,(\mathrm{mod}\, q)$ unabhängige Zufallsvariablen.
Wichtig für die weitere Diskussion ist die *Euler-Φ-Funktion*: $\Phi(N)$ ist die Anzahl aller $0 < y < N$ mit $\mathrm{ggT}(y, N) = 1$. Damit ist für jede Primzahl p $\Phi(p) = p - 1$ und $\Phi(p^k) = p^{k-1}(p-1)$. Für verschiedene Primzahlen p und q gilt $\Phi(pq) = (p-1)(q-1)$. Allgemein gilt: Wenn $\mathrm{ggT}(m, n) = 1$, dann $\Phi(mn) = \Phi(m)\Phi(n)$.

Satz 2 *(Euler-Fermat) Sei* $\mathrm{ggT}(y, N) = 1$. *Dann ist* $y^{\Phi(N)} = 1(\mathrm{mod}\, N)$. *Außerdem existiert zu jeder Primzahl* p *ein* z, *so dass*

$$\{z^k(\mathrm{mod}\, p),\, 1 \leq k < p\} = \{a : 1 \leq a < p\}.$$

Beweis siehe [HW60], Satz 111.
Um die zentrale Idee von Shor verstehen zu können, brauchen wir noch den Begriff *Ordnung*. Die Ordnung r einer Zahl x bezüglich N ist definiert als

$$r = \inf\{k > 0 \,|\, x^k = 1 \,(\mathrm{mod}\, N)\} \ . \tag{25}$$

Sie ist also die kleinste natürliche Zahl k, für die die Potenz x^k gleich 1 modulo N ist. Daraus folgt direkt, dass die Abbildung $f(a) = x^a(\mathrm{mod}\, N)$ periodisch ist mit der Periode r.
Der nächste Satz ist essentiell für den Shor-Algorithmus. Denn der erste Schritt im Shor-Algorithmus (nach der Vorbereitung) ist es, ein zufälliges $1 \leq y < N$ mit $\mathrm{ggT}(y, N) = 1)$ zu wählen. Mit dieser Zahl lässt sich ein Faktor von N bestimmen, wenn die Ordnung von y bezüglich N gerade und $y^{r/2} \neq -1 \,(\mathrm{mod}\, N)$ ist.

Satz 3 *Sei* $N = pq$ *und* $S := \{y : 1 \leq y < N, \mathrm{ggT}(y, N) = 1\}$. *Dann gilt, dass mindestens die Hälfte der Elemente von* S *eine gerade Ordnung* $r = 2k$ *haben und der Gleichung* $y^k \neq -1 \,(\mathrm{mod}\, N)$ *genügen.*

Beweis: Sei $y \in S$ mit Ordnung $r = \mathrm{kgV}(s, t)$ (s und t seien die Ordnungen von y bezüglich p und q). Wenn man s als $s = 2^i u$ schreibt und t als $t = 2^j v$, wobei u und v ungerade sind, dann ist $r = 2^{max(i,j)}\mathrm{kgV}(u, v)$. Damit ist r nur ungerade, wenn sowohl i und j Null sind.
Nehmen wir an, dass $r = 2k$ gerade ist und $y^k = -1 \,(\mathrm{mod}\, N)$. Aus Satz 1 folgt, dass $y^k = -1 \,(\mathrm{mod}\, N)$ dann und nur dann, wenn $y^k = -1 \,(\mathrm{mod}\, p)$ und $y^k = -1 \,(\mathrm{mod}\, q)$. Aber wenn $i < j$, dann ist k ein Vielfaches von s, und $y^k = 1 \,(\mathrm{mod}\, p)$. Dies ist ein Widerspruch zur Annahme. Eine analoge Aussage gilt für $j < i$, und damit gilt, dass

$i = j$, also dass $y^k = -1 \pmod N$ dann und nur dann, wenn die Ordnungen s und t gleich oft den Faktor 2 in ihrer Primfaktorzerlegung haben.
Sei nun $p-1 = 2^m x$, wobei x eine ungerade Zahl ist. Satz 2 sagt, dass die natürlichen Zahlen $(\mathrm{mod}\, p)$ von den ersten $(p-1)$ Potenzen von einem z mit $\mathrm{ggT}(z,p) = 1$ erzeugt werden. Folglich hat eine Zahl b nur dann eine ungerade Ordnung r_b bezüglich p, wenn $b = z^{2^m w}$ mit $1 \leq w \leq x$. Denn die Ordnung r_b muss r teilen. Folglich ist der Anteil von Zahlen, die eine ungerade Ordnung haben, 2^{-m}. Es zeigt sich zudem, dass genau die Zahlen b mit $b = z^{m-k} w$, w ungerade, eine Zweierpotenz 2^k als Ordnung bezüglich p haben. Damit kann w ungerade Werte zwischen 1 und $2^k x - 1$ annehmen. Von diesen Zahlen gibt es genau $2^{k-1}x$. Der Anteil von Elementen von S, deren Ordnung eine Potenz von 2 ist, ist 2^{k-1-m}.
Sei nun $p-1 = 2^m x$ und $q-1 = 2^n w$, wobei $1 \leq m \leq n$ und x und w ungerade Zahlen sind. Die Wahrscheinlichkeitsdeutung von Satz 1 besagt, dass man die Zahlenverhältnisse für die Fälle $(\mathrm{mod}\, p)$ und $(\mathrm{mod}\, q)$ auf S übertragen kann. Der Anteil von Elementen in S, die eine ungerade Ordnung oder eine gerade Ordnung mit $y^k = -1 \pmod N$ haben, ist daher:

$$\left(\frac{1}{2}\right)^{m+n} + \sum_{k=1}^{m} \left(\frac{1}{2}\right)^{m+n-2k+2} = \left(\frac{1}{2}\right)^{m+n} \left(1 + \sum_{j=0}^{m-1} 4^j\right) \leq \frac{1}{2} \ . \tag{26}$$

Diese Argumentation lässt sich noch weiterführen [Sho94], indem die Anzahl der unterschiedlichen Primfaktoren von N berücksichtigt wird. Dies liefert aber kein wesentlich anderes Ergebnis, sondern verbessert nur die Abschätzung für die Erfolgswahrscheinlichkeit.
Der Shor-Algorithmus hat nach diesem Satz eine Erfolgswahrscheinlichkeit, die größer als $1 - 2^{-m+1}$ ist, wenn m die Anzahl der unterschiedlichen Primfaktoren von N ist. Ein letzter Satz vervollständigt das Kapitel:

Satz 4 *Sei N eine zusammengesetzte Zahl, die mit L Bits dargestellt werden kann. Sei weiter x eine nicht triviale Lösung der Gleichung $x^2 = 1 \pmod N$ mit $1 \leq x \leq N$ (also weder $x = 1 \pmod N$ und $x = -1 \pmod N$). Dann ist mindestens eine der beiden Zahlen $\mathrm{ggT}(x-1, N)$ und $\mathrm{ggT}(x+1, N)$ ein Faktor von N, der nicht 1 oder N ist.*

Beweis: Da $x^2 = 1 \pmod N$, ist N ein Teiler von $x^2 - 1 = (x-1)(x+1)$. Daher muss N einen Faktor mit $(x+1)$ oder $(x-1)$ gemeinsam haben. Da aber nach Voraussetzung $1 < x < N-1$, gilt $x-1 < x+1 < N$, also ist der gemeinsame Faktor nicht N. Mit dem Euklidischen Algorithmus kann $\mathrm{ggT}(x-1, N)$ und $\mathrm{ggT}(x+1, N)$ in $O(L^3)$ bestimmt werden.
Der Shor-Algorithmus ist nach heutigem Wissen eine Anwendung einer viel grundlegenderen Prozedur. Diese Prozedur basiert auf der Quanten-Fourier-Transformation und wird im folgenden Abschnitt erläutert.

4.2. Phasenabschätzung

Die Quanten-Fourier-Transformation ist für jede Anzahl von Qubits definiert und hängt ansonsten von keinen weiteren Parametern ab. Sie ist damit eine sehr spezielle Prozedur. Es gibt aber auch allgemeine Prozeduren, die durch den Austausch eines oder mehrerer zentraler Gatter auf viele verschiedene Probleme angepasst werden können.
Eine solche Prozedur ist die Phasenabschätzung. Sie dient dazu, Aussagen über den Eigenwert einer unitären Transformation U zu einem bestimmten Eigenvektor $|u\rangle$ zu machen. Solche Transformationen haben ausschließlich komplexe Eigenwerte vom Betrag 1, welche sich folglich als $e^{2\pi i\phi}$ mit $0 \leq \phi < 1$ schreiben lassen. Die Phasenabschätzung macht nun das, was ihr Name schon aussagt: Sie schätzt die Phase ϕ ab.
Zur allgemeinen Beschreibung dieser Prozedur werden zwei Annahmen gemacht: Der Zustand $|u\rangle$ sei präparierbar und es existiere ein Orakel-Gatter, das die Operationen controlled-U^{2^j} zusammen in einem Schritt berechnen kann. Für eine konkrete Anwendung müssen diese Annahmen realisierbar sein! Abschnitt 4.3 greift dieses Problem für die Bestimmung der Ordnung wieder auf.
Die Prozedur arbeitet auf zwei Quantenregistern. Das zweite Register ist der Arbeitsspeicher für das Orakel-Gatter und von Anfang bis Ende der Prozedur im Zustand $|u\rangle$. Es muss genau so viele Qubits haben, dass $|u\rangle$ dargestellt werden kann. Bei der weiteren Analyse braucht also nur das erste Register betrachtet werden. Im ersten Register passiert das eigentlich Spannende, denn nach der Prozedur ist hier das Ergebnis, also die abgeschätzte Phase, gespeichert. Im einfachsten Fall ist ϕ exakt durch n Bits darstellbar. Dann reicht es, im ersten Register n Qubits zu haben. Allgemein braucht man für eine auf n Stellen genaue Abschätzung mehr als n Qubits. Dies wird sich aber in der genauen Analyse zeigen. Die Anzahl der Qubits im ersten Register sei mit t bezeichnet.

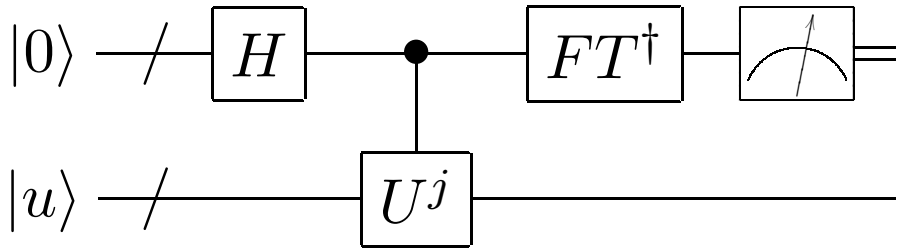


Abbildung 12: Schematischer Schaltkreis der Prozedur zur Phasenabschätzung.

Der erste Schritt der Prozedur (Abbildung 12) besteht darin, auf alle Qubits im ersten Register das Hadamard-Gatter anzuwenden. Im zweiten Schritt werden die controlled-U^{2^j} Operationen durchgeführt, so dass jedes Qubit des ersten Registers beginnend mit dem letzten genau eine dieser Operationen kontrolliert hat. Die Operation controlled-U^{2^j} wirkt auf das zweite Register und wird vom $(t-j)$-ten Qubit des ersten Registers

kontrolliert. j läuft also von 0 bis $t-1$.
Nach dem zweiten Schritt befindet sich das erste Register damit im Zustand

$$|\phi_2\rangle = \frac{1}{2^{t/2}} \sum_{k=0}^{2^t-1} e^{2\pi i \phi k} |k\rangle \quad . \tag{27}$$

Im dritten Schritt wird die inverse QFT auf das erste Register angewendet. Dies ergibt:

$$|\phi_3\rangle = \frac{1}{2^t} \sum_{k,l=0}^{2^t-1} e^{-\frac{2\pi i k l}{2^t}} e^{2\pi i \phi k} |l\rangle \quad . \tag{28}$$

Die Messung des ersten Registers ist der abschließende vierte Schritt.
Der Wert von ϕ kann zwischen 0 und 1 gewählt werden. Die Darstellung von ϕ als (im Allgemeinen unendlicher) dyadischer Bruch sei mit $\phi = 0.\phi_1\phi_2\ldots$ ($\phi_i \in \{0,1\}$) bezeichnet und mit b die Zahl zwischen 0 und 2^t-1, so dass $b/2^t = 0.b_1b_2\ldots b_t$ die beste t-Bit-Näherung von ϕ mit $b/2^t \leq \phi$ ist.
Lässt sich b im vierten Schritt messen? Eine detaillierte Analyse [NC00] zeigt, dass das Messergebnis m eine schlechtere Genauigkeit als b hat. Zudem hat $|\phi_3\rangle$ nicht verschwindende Amplituden auch bei Basiszuständen, die nicht in der Nähe von b sind. Die Prozedur zur Phasenabschätzung liefert also mit einer gewissen Wahrscheinlichkeit ein nicht erwünschtes (also falsches) Ergebnis. Die Analyse zeigt aber, dass m mit einer Wahrscheinlichkeit von $1-\epsilon$ auf n Stellen genau ist, wenn das erste Register

$$t = n + \left\lceil \log\left(2 + \frac{1}{2\epsilon}\right) \right\rceil \tag{29}$$

Qubits enthält. Ein ausreichend groß dimensioniertes erstes Register ermöglicht also eine beliebige hohe Genauigkeit und beliebig kleine Wahrscheinlichkeit für einen Misserfolg.
Zusätzlich zu diesem sehr nützlichen Ergebnis zeigt sich, dass die Phasenabschätzung relativ robust gegenüber Ungenauigkeiten ist. Die Wahrscheinlichkeit, eine gute Näherung für die Phase ϕ zu messen, verringert sie nur unwesentlich, wenn $|u\rangle$ nicht optimal präpariert ist. Denn der wirklich präparierte Zustand lässt sich in der Eigenbasis von U entwickeln. Und wenn die Präparation nicht völlig schlecht ist, dann sollte $|u\rangle$ die bei Weitem höchste Amplitude in dieser Darstellung haben. Und diese Wahrscheinlichkeitsamplitude überträgt sich auf den Ergebniszustand. Die Erfolgswahrscheinlichkeit ist damit $|c_u|^2(1-\epsilon)$, wenn c_u die Amplitude von $|u\rangle$ ist.
Abgesehen von dem einen Aufruf des Orakels und den vorbereitenden Hadamard-Gattern besteht die Prozedur zur Phasenabschätzung aus der inversen QFT. Die Phasenabschätzung ist damit in der gleichen Komplexitätsklasse wie die QFT, also in $\Theta(t^2)$.

4.3. Ordnung bestimmen

Eine wichtige Anwendung für die Phasenabschätzung ist die Bestimmung der Ordnung r einer Zahl x bezüglich N. Dabei ist $x < N$ und L die Anzahl von Bits, die man braucht, um x darzustellen. Zur Definition siehe Abschnitt 4.1.
Die Ordnungsbestimmung ist einfach die Prozedur zur Phasenabschätzung, wenn man als unitären Operator U die Abbildung

$$U : |y\rangle \longrightarrow |xy(\mathrm{mod}\, N)\rangle \tag{30}$$

wählt ($y \in \{0,1\}^L$). In der Phasenabschätzung lassen sich die controlled-U^{2^j}-Operationen mit der *modularen Exponentiation* effizient implementieren. Dies ist eine Technik, bei der die nötigen Potenzen $x^k \,(\mathrm{mod}\, N)$ aus Potenzen $x^{2^j} \,(\mathrm{mod}\, N)$ zusammengesetzt werden. Die Komplexität dieses Verfahrens ist $\Theta(L^3)$.
Es zeigt sich, dass die Zustände

$$|u_s\rangle \;=\; \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} \exp\left[\frac{-2\pi i s k}{r}\right] |x^k \,(\mathrm{mod}\, N)\rangle \tag{31}$$

Eigenvektoren von U zum Eigenwert $\exp\left[\frac{2\pi i s}{r}\right]$ sind. Denn in (31) gibt es die Zustände $|x^0\rangle$, $|x^1\rangle, \ldots, |x^{r-1}\rangle \,(\mathrm{mod}\, N)$. Durch die Anwendung von U wird zu jeder Potenz ein weiteres x hinzumultipliziert. Es ergeben sich die Potenzen $|x^1\rangle$, $|x^2\rangle, \ldots, |x^r\rangle \,(\mathrm{mod}\, N)$. Die letzte Potenz, x^r, ist aber auf Grund der Definition von r gleich 1 bzw. x^0. Damit erhält man die gleichen Potenzen. Wenn man nun noch aus jedem Summanden den Faktor $\exp\left[\frac{2\pi i s}{r}\right]$ ausklammert, ergibt sich

$$U\,|u_s\rangle \;=\; \exp\left[\frac{2\pi i s}{r}\right] |u_s\rangle \quad . \tag{32}$$

Die Prozedur zur Phasenabschätzung ermittelt die Phase, in diesem Fall $\frac{s}{r}$, mit hoher Genauigkeit, und daraus lässt sich die Ordnung r ermitteln. Um die Phasenabschätzung durchführen zu können muss ein Eigenzustand präpariert sein. Jeder Eigenzustand ist aber vom gesuchten r abhängig! Der Ausweg aus diesem Dilemma liegt in einer speziellen Eigenschaft der Eigenzustände. Dazu ist die folgende Summe wichtig:

$$\sum_{s=0}^{r-1} \exp(-2\pi i k s/r) \;=\; r\delta_{k0} \quad . \tag{33}$$

Falls $k \neq 0$ werden r komplexe Zahlen vom Betrag 1, deren Richtungen in der komplexen Zahlenebene gleichmäßig verteilt sind, aufaddiert. Dies ergibt 0. Falls aber $k = 0$, so wird

einfach r-mal die Zahl 1 addiert mit dem Ergebnis r. Nun folgt weiter:

$$\begin{aligned}
&\frac{1}{\sqrt{r}}\sum_{s=0}^{r-1}\frac{1}{\sqrt{r}}\sum_{k=0}^{r-1}\exp(-2\pi iks/r)\,|x^k(\mathrm{mod}\,N)\rangle \\
&\quad=\frac{1}{r}\sum_{s=0}^{r-1}\sum_{k=0}^{r-1}\exp(-2\pi iks/r)\,|x^k(\mathrm{mod}\,N)\rangle \\
&\quad=\frac{1}{r}\sum_{k=0}^{r-1}\sum_{s=0}^{r-1}\exp(-2\pi iks/r)\,|x^k(\mathrm{mod}\,N)\rangle \\
&\qquad\qquad=\frac{1}{r}\sum_{k=0}^{r-1}r\delta_{k0}\,|x^k(\mathrm{mod}\,N)\rangle \\
&\qquad\qquad=|x^0(\mathrm{mod}\,N)\rangle\;=\;|1\rangle\,. \qquad (34)
\end{aligned}$$

Der Zustand $|1\rangle$ ist somit Eigenzustand und sehr einfach zu präparieren. Damit ist dieses Problem gelöst und die Phasenabschätzung liefert eine Näherung für s/r. Da alle Eigenzustände und damit alle möglichen Werte für s im Zustand $|1\rangle$ vorkommen, ergibt die Messung ein zufälliges s/r. Um aus dieser Zahl eine gute Näherung für r zu extrahieren bedient man sich der *Kettenbruchentwicklung*. Ein Kettenbruch ist ein Gebilde der Form

$$a_0+\cfrac{1}{a_1+\cfrac{1}{a_2+\ldots}} \qquad (35)$$

oder kurz

$$[a_0,a_1,a_2,\ldots]\;. \qquad (36)$$

Für jede reelle Zahl ist die Kettenbruchentwicklung eindeutig, für rationale Zahlen sogar endlich. Dazu ist der folgende Satz wichtig:

Satz 5 *Sei s/r eine rationale Zahl mit*

$$\left|\frac{s}{r}-\phi\right|\le\frac{1}{2r^2}\;. \qquad (37)$$

Dann ist s/r eine Konvergente der Kettenbruchentwicklung von ϕ und kann daher in $O(L^3)$ Schritten mit dem Algorithmus zur Kettenbruchentwicklung berechnet werden.

Und da die Phasenabschätzung beliebig genau sein kann, gilt dieser Satz für die Bestimmung der Ordnung. Ein formaler Beweis findet sich zum Beispiel in [NC00]. Die Kettenbruchentwicklung von dem Messwert ϕ, der eine n-Bit-Näherung von s/r ist, ergibt einen gekürzten Bruch s'/r'. Und der Nenner dieses Bruchs ist mit hoher Wahrscheinlichkeit die gesuchte Ordnung!
Abgesehen von der Möglichkeit, dass die gemessene Näherung zu schlecht ist — die Wahrscheinlichkeit dafür kann durch ein großes erstes Register beliebig klein gemacht

werden— bleibt das Problem, dass s und r in manchen Fällen nicht teilerfremd sind und damit r' nur ein Teiler von r ist.
Es gibt einige Techniken, um dennoch die Ordnung zu bestimmen. Eine Idee ist, die Prozedur zur Phasenabschätzung einfach oft genug zu wiederholen, bis man eine teilerfremdes Paar (s', r') ermittelt. Die Wahrscheinlichkeit, ein teilerfremdes s zu messen, ist mindestens $1/2log(r) > 1/2log(N)$. Eine $2log(N)$-malige Wiederholung führt mit hoher Wahrscheinlichkeit zum Ziel. Es gibt noch weitere Ideen, deren Erwähnung hier aber zu weit führen würde (siehe zum Beispiel [Sho94]). Wichtig ist nur, dass der Aufwand zur Korrektur bei besseren Techniken konstant ist.
Insgesamt liegt der Algorithmus zur Bestimmung der Ordnung in der Komplexitätsklasse $\Theta(L^3)$: Die Hadamard-Transformationen tragen dazu $\Theta(L)$ bei, die inverse QFT $\Theta(L^2)$ und die modulare Exponentiation den größten Beitrag $\Theta(L^3)$. Die Kettenbruchentwicklung liegt in der Klasse $\Theta(L^3)$. Und die nötigen Wiederholungen, um teilerfremde s' und r' auszugleichen, lassen sich konstant halten.

4.4. Faktorisierung

Im letzten Abschnitt wurde ein erster konkreter Quantenalgorithmus vorgestellt. Dies wird auch der einzige bleiben, denn Shor hat gezeigt [Sho94], dass das Problem der Ordnungsbestimmung und das Problem der Faktorisierung äquivalent sind. Der Schlüssel zu dieser Äquivalenz liegt in Satz 4. Dort wird gezeigt, wie die Ordnung mit der Faktorisierung zusammenhängt.
Wie lässt sich dieser Satz praktisch nutzen? Die Idee von Shor ist, von einer zufälligen Zahl $0 < x < N$ die Ordnung zu bestimmen, indem der Algorithmus zur Bestimmung der Ordnung verwendet wird, und dann eine modifizierte Version von Satz 4 anzuwenden. Damit ist eine präzise Beschreibung möglich:
Eingabe: Eine zusammengesetzte Zahl N.
Ausgabe: Ein nicht trivialer Faktor von N.
Laufzeit: $O((\log N)^3)$ Gatteroperationen.
Erfolgswahrscheinlichkeit: $O(1)$.
Ablauf:

1. Wenn N gerade ist, gib 2 als Ergebnis zurück.
2. Falls $N = a^b$ für Zahlen $a \geq 2$ und $b \geq 2$, gib a als Ergebnis zurück.
3. Generiere eine Zufallszahl x mit $1 \leq x \leq N - 1$. Wenn $\mathrm{ggT}(x, N) > 1$, gib $\mathrm{ggT}(x, N) > 1$ als Ergebnis zurück.
4. Benutze die Prozedur *Bestimmung der Ordnung*, um r zu erhalten.
5. Wenn r gerade ist und $x^{r/2} \neq -1 \,(\mathrm{mod}\, N)$, dann berechne $\mathrm{ggT}(x^{r/2} - 1, N)$ und $\mathrm{ggT}(x^{r/2} + 1, N)$. Wenn dabei ein Faktor von N bestimmt wurde, gib ihn als

Ergebnis zurück. Wenn nicht, ist der Algorithmus fehlgeschlagen (Wiederholung ab Schritt 3).

Satz 3 stellt die Funktionsweise des Algorithmus sicher. Denn nach diesem Satz erfüllt mehr als die Hälfte der Zahlen, die in Schritt 3 zufällig gewählt werden können, die Bedingung, dass die Ordnung r gerade und $x^{r/2} \neq -1 \,(\mathrm{mod}\, N)$ ist. Und nach Satz 4 ist sicher gestellt, dass in diesem Fall in Schritt 5 auch ein Faktor bestimmt wird.

5. Simulation

Mit der mathematischen Darstellung kann die allgemeine Funktionsfähigkeit des Shor-Algorithmus' von Hand gezeigt werden. Die Rechenschritte für einen konkreten Fall sind aber zu kompliziert, um manuell durchgeführt werden zu können. Um Details untersuchen zu können, braucht man einen Quantencomputer. Dieser muss auf einem klassischen Computer simuliert werden, da passende Geräte noch nicht verfügbar sind.
Der Begriff der Berechenbarkeit ist für klassische und für Quantencomputer gleich. Ein Problem, welches auf einem Quantencomputer berechenbar ist, ist damit auch auf einem klassischen Computer berechenbar. In dieser abstrakten, mathematischen Sichtweise sind die beiden Arten von Computern also äquivalent. Aus einer praktischen Sichtweise ist diese Äquivalenz aber nicht sinnvoll. Denn die Simulation von großen quantenmechanischen Systemen zum Beispiel braucht mehr Speicher, als klassisch im ganzen Universum vorhanden ist. Ein solches Problem ist aus Sicht der Physik auf einem klassischen Computer also nicht berechenbar, da es in unserem Universum nie berechnet werden kann. Ein Quantencomputer kann eine solche Simulation unter Umständen aber schon in sehr kurzer Zeit durchführen und hat in dieser Perspektive ein größeres Repertoire von Problemen, die er jemals berechnen könnte. Mathematisch ist aber auch jeder klassische Computer in der Lage, den Shor-Algorithmus zu berechnen.
Die Simulation eines Quantencomputers auf einem klassischen Computer ist nicht effizient, da die Dimension des Hilbertraumes zur Beschreibung der Qubits exponentiell mit der Anzahl der Qubits steigt. Zur vollständigen Beschreibung von n Qubits braucht man 2^n komplexe Zahlen. Diese werden im klassischen Computer durch je zwei Gleitkommazahlen repräsentiert. Bei 64-Bit-Gleitkommazahlen braucht man für ein n-Qubit-Register $SR(n)$ Bytes mit

$$SR(n) = \frac{64}{8} \cdot 2 \cdot 2^n = 2^{n+4} \quad . \tag{38}$$

Für die Repräsentation von Gattern braucht man quadratische, komplexe Matrizen. Ein Gatter, welches auf ein n-Qubit-Register wirken soll, hat 4^n komplexe Einträge. Analog wie für die Register ergibt sich ein Speicherbedarf von $SG(n)$ Bytes mit

$$SG(n) = \frac{64}{8} \cdot 2 \cdot 4^n = 4^{n+2} \quad . \tag{39}$$

Dieses exponentielle Wachstum schränkt die Größe der simulierbaren Systeme stark ein. Selbst auf aktuellen Großrechnern mit optimierter Software ist die Grenze bei wenigen Qubits erreicht. So schafft ein Linux-Cluster des Fraunhofer-Instituts (56 Gigabyte Arbeitsspeicher) die Simulation von 31 Qubits.
Das Ziel meiner Arbeit ist aber nicht die Programmierung einer optimierten Simulationssoftware für Quantenalgorithmen, sondern die prinzipielle Simulation, mit der sich bestimmte Eigenschaften und Konzepte verdeutlichen lassen.

5.1. Struktur der Simulationssoftware

5.1.1. Objektorientierung aus Sicht der Quanteninformatik

Als mächtigstes Hilfsmittel zum Design von erweiter- und wartbarer, verständlicher und vor allem umfangreicher Software hat sich das objektorientierte Paradigma erwiesen. Dabei werden Programme als Interaktion von *Objekten* geschrieben, wobei diese Objekte eine Abstraktion von dem gewohnten Begriff "Objekt" sind. Dies bedeutet vor allem, dass Objekte eine gekapselte Menge von Daten plus auf diesen Daten arbeitenden Funktionen (*Methoden*) sind. Ein Programm besteht nun aus der Erschaffung und Initialisierung von Objekten, die dann sich selbst überlassen werden. Die Interaktion der Objekte liefert schließlich das gewünschte Ergebnis. Zur Definition eines Objektes schreibt man nun eine Art Bauplan, die *Klasse*. In einer solchen Klasse ist genau aufgeführt, welche Daten und Funktionen zu einem Objekt dieser Klasse gehören. Obwohl die Idee des objektorientierten Softwaredesigns sehr jung ist, ist das Konzept schon sehr alt: Der griechische Philosoph Platon (427 - 347 v. Chr.) vertrat die Ansicht, dass die Welt zweigeteilt ist, in die *Sinneswelt* und die *Ideenwelt*. In der Sinneswelt besteht alles aus "Material". Alle Objekte in der Sinneswelt entstehen und vergehen irgendwann wieder, nachdem sie eine Zeit lang mit anderen Objekten interagiert haben. Aber gleichzeitig ist jedes Objekt nach einer zeitlosen *Form* oder *Idee* gebildet, die ewig und unveränderlich ist. Und diese Ideen existieren in der Ideenwelt. So ist nach Platon jedes Pferd anders, es hat bestimmte äußere und innere Merkmale, die es von jedem andere Pferd unterscheiden. Doch alle Pferde sind Ausprägungen der eigentlichen "Pferdeform". Es ist erstaunlich, dass die Ideenlehre von Platon sich nahezu vollständig mit dem objektorientierten Paradigma deckt! Und genauso, wie Platon versuchte, durch seine Lehre seine Sicht von der Welt zu ordnen, so versuchen Programmierer heute, ihre Sicht von ihrer kleinen "Programmwelt" zu ordnen.
Das objektorientierte Paradigma ist jedoch vor allem ein Ergebnis der Entwicklung von Computern und dem Bedarf nach Techniken, die dem menschlichen Denkmuster angepasst sind. Es lohnt sich nun ein Vergleich der Entwicklung von klassischen und Quanten-Computern. Denn es gibt Parallelen: Vor dem eigentlichen Bau gab es bei klassischen Computern Ideen für Algorithmen. Diese abstrakten Ideen wurden dann an die praktischen Anforderungen angepasst, denn die massiven Einschränkungen von Speicherkapazität und Prozessorfähigkeiten verlangten den Programmierern einiges an Kreativität

ab. In der Quanteninformatik ist es ähnlich. Die Algorithmen sind da, aber Quantenhardware ist doch nur recht beschränkt verfügbar. Neben der Optimierung der Algorithmen werden Fehlerkorrekturmechanismen entwickelt. Dadurch darf Quantenhardware störanfälliger sein, so dass die technischen Anforderungen sinken. Wie in der klassischen Entwicklung findet also ein ähnliches Zusammenwachsen von Soft- und Hardware statt: Beide Gebiete werden auf die Anforderungen jeweils des anderen optimiert.

Mit der Verfügbarkeit von Speicher und Rechenleistung hat sich der Fokus der Programmierung verschoben. Über Hochsprachen, Programmierparadigmen und Techniken des Software-Engineerings wurden immer kompliziertere Daten- und Programmstrukturen nutz- und beherrschbar. Dabei drängt sich die Frage auf, ob sich solche Entwicklungen auf die Quanteninformatik übertragen lassen oder ob es völlig neue Konzepte geben muss. Das objektorientierte Paradigma ist deshalb so nützlich, weil es den Prozess der Programmierung der menschlichen Denkweise zugänglicher macht. Menschen denken nunmal objektorientiert. Die Frage ist nun, ob dieses Paradigma auch in der Lage ist, die Besonderheiten der Quanteninformatik und damit der Quantenmechanik "in unsere Köpfe zu kriegen", also der menschlichen Denkweise näherzubringen. Das entspricht einer Idee aus der Quanteninformatik [NC00]: Nachdem wir physikalisch an die Informatik herangegangen sind, sollten wir jetzt vielleicht mit dem Blickwinkel der Informatik über die Physik nachdenken.

Ein Anhaltspunkt, dass das objektorientierte Paradigma nicht für die Darstellung quantenmechanischer Zusammenhänge geeignet ist, liegt in den Eigenschaften des Quantenregisters. Ein Quantenregister besteht aus N Qubits, aber ein allgemeiner Registerzustand lässt sich auf Grund von Superposition bzw. Verschränkung nicht als Produktzustand von N Qubits schreiben. Ein Quantenregister ist mehr als die Summe seiner Teile! In der objektorientierten Sichtweise ist es aber gerade die Regel, Objekte als Zusammensetzungen anderer Objekte aufzufassen. Dies deutet darauf hin, dass die Objektorientierung eine ganz und gar klassische Sichtweise und aus diesem Grund kein geeignetes Konzept für die Quanteninformatik ist.

Mit Hilfe der Mathematik ist der menschliche Verstand aber durchaus in der Lage, quantenmechanische Probleme zu beschreiben und zu begreifen, während das ohne mathematische Hilfen enorm schwierig ist. Es ist daher logisch, dass die Objektorientierung die menschliche Denkstruktur nicht vollständig erfasst. Dies wiederum impliziert, dass es ein noch besseres Programmierparadigma geben muss, welches unserem Denken noch angepasster ist. Und die Tatsache, dass die menschliche Denkweise mit Hilfe von mathematischen Techniken quantenmechanische Zusammenhänge begreifen kann, könnte darauf hindeuten, dass die Objektorientierung mit gewissen Erweiterungen in der Quanteninformatik nützlich sein kann. Da die Objektorientierung aber eine sehr klassische Sichtweise ist (zumal sie aus der Antike stammt!), ist die Möglichkeit wahrscheinlicher, dass ein völlig neues Konzept erdacht werden muss, welches die Objektorientierung ersetzt.

5.1.2. Simulation eines Quantencomputers

Um den Ablauf von Quantenalgorithmen verdeutlichen zu können, müssen die Begriffe der Quanteninformatik zugänglich sein. Die erste Aufgabe ist also, quanteninformatische "Objekte" wie Register und Gatter zu simulieren. In der objektorientierten Programmierung verschafft man sich einen ersten Eindruck der nötigen Klassen, indem man das Problem in Sätzen formuliert und dann eine *Hauptwortanalyse* durchführt. Denn die Hauptwörter (oder Substantive) sind die beteiligten Objekte. Erster Versuch:
In einem Quantencomputer wirken *Quantengatter* auf *Quantenregister*. Quantengatter sind spezielle *quadratische Matrizen* über den *komplexen Zahlen*. Quantenregister bestehen aus mehreren *Qubits*.
Es müsste demnach fünf grundlegende Klassen geben: Quantengatter, Quantenregister, quadratische Matrizen, komplexe Zahlen und Qubits. Quadratische Matrizen wären Zusammenschlüsse von komplexen Zahlen, Quantengatter eine Spezialisierung von quadratischen Matrizen und Quantenregister Zusammenschlüsse von Qubits. Formal wären quadratische Matrizen und Register von gleicher Struktur: ein Zusammenschluss von einfachen Objekten. Die Struktur ist aber völlig unterschiedlich! Denn bei einer quadratischen Matrix kann man zu jedem Zeitpunkt genau sagen, in welchem Zustand sich die einzelnen Bestandteile befinden, welchen Wert also jede einzelne komplexe Zahl hat. Bei einem Quantenregister ist das nur möglich, wenn es in einem Produktzustand ist. Dies ist aber ein Sonderfall. Im Normalfall liegt ein verschränkter Zustand vor. Und dieser lässt sich nicht als Zusammenschluss von Qubits darstellen. Es führt also kein Weg daran vorbei, sich von der anschaulichen Vorstellung vom Quantenregister als Zusammenschluss von Qubits zu lösen und die mathematische Beschreibung zu bemühen.
Zweiter Versuch:
In einem Quantencomputer wirken *Quantengatter* auf *Quantenregister*. Quantengatter sind spezielle *quadratische Matrizen* über den *komplexen Zahlen*. Quantenregister werden durch mehrere komplexe Zahlen beschrieben.
Mit diesen vier Klassen ist nun die Grundlage geschaffen. Konkrete Gatter sind Spezialfälle von Quantengattern. Initialisierung und Messungen werden durch das Register gekapselt, also von der Register-Klasse übernommen. Damit sind Berechnungen auf Quantencomputern simulierbar: Quantenregister werden initialisiert, durch verschiedene Gatter verändert und schließlich gemessen.

5.1.3. Schaltkreise

Die grundlegenden Klassen ermöglichen die prinzipielle Simulation eines Quantencomputers. In Abschnitt 2.4 wurde gezeigt, dass ein geschickt gewählter Satz von Gattern universell ist. Diese universellen Gatter werden jedoch oft zu logischen Einheiten zusammengefasst. Je nach dem, wie komplex und flexibel eine Einheit ist, kann man sie als zusammengesetztes Gatter oder als *Schaltkreis* bezeichnen. Ein Schaltkreis ist im Allgemeinen von weiteren Parametern abhängig, zum Beispiel von der Anzahl der Qubits.

Anstatt die SWAP-Operation an jeder Stelle als drei CNOT-Gatter zu schreiben, fasst man drei CNOT-Gatter zum SWAP-Gatter zusammen. Dies ändert nichts an der inneren Struktur, verbessert aber die Übersicht.
Genauso fasst man die vielen Gatter der Quanten-Fourier-Transformation zu einem Schaltkreis zusammen. So kapselt eine Klasse die Idee hinter der Transformation und kann für jeden konkreten Fall (also für jede konkrete Anzahl von Qubits) die benötigten Gatter zusammenstellen.
Eine Besonderheit bildet das kontrollierte Gatter. Diese Klasse hängt von dem zu kontrollierenden Gatter und der Anzahl der zwischenliegenden Qubits ab. Trotz dieser Parameter wird das kontrollierte Gatter im Java-Code nicht als Schaltkreis, sondern als Gatter implementiert, da der Einbau in größere Schaltkreise dann besser funktioniert. Die Unterscheidung in Schaltkreise und Gatter ist daher nicht fundamental, sondern dient nur manchmal der Übersicht oder der Anschaulichkeit. Wenn man sich aber darauf beschränkt, einen Algorithmus nur mit einem festen Satz von Gattern zu implementieren, gewinnt die Unterteilung an Bedeutung: Denn dann ist ein Gatter ein Element dieser ausgewählten Menge und ein Schaltkreis jegliche Verbindung dieser Gatter.

5.1.4. Algorithmen

Der Einsatz der meisten Quantenschaltkreise muss vor- und nachbereitet werden. So muss im Shor-Algorithmus zunächst eine zufällige Zahl erzeugt werden. Die Kettenbruchentwicklung und der Euklidische Algorithmus wandeln das Messergebnis schließlich in einen gesuchten Faktor.
Diese Vor- und Nachbereitungen gehören logisch gesehen zu dem Schaltkreis. Es liegt also nahe, diese drei Aspekte in *Algorithmus*-Klassen zu kapseln. Damit wird das eigentliche Ziel der Bemühungen klar: Gesucht ist eine Klasse *ShorAlgorithmus*, die die Vorbereitungen, den Schaltkreis und die Nachbearbeitung zusammenfügt und somit zur Faktorisierung von Zahlen geeignet ist. Diese Klasse wird in Abschnitt 5.2.4 beschrieben.

5.2. Details der wichtigen Klassen

Viele Passagen des Quelltextes sind selbsterklärend oder ergeben sich, weil das Programm objektorientiert ist. Andere Abschnitte erhöhen den Komfort, indem zum Beispiel Konstruktoren für verschiedene Situationen vorhanden sind. Eine detaillierte Analyse des kompletten Quelltextes wäre mühselig und wenig gewinnbringend, daher werden nur die zentralen Passagen und die Klassenstrukturen erklärt.

5.2.1. Grundlegende Klassen

Die Grundlagen für die Simulation von Quantencomputern sind die Klassen `Complex`, `SquareMatrix`, `Gate` und `Register`. Sie ermöglichen komplizierte Strukturen wie kontrollierte Gatter und schließlich die direkte Umsetzung von Quantenschaltkreisen.

Da Java keinen eingebauten Datentyp zur Repräsentation komplexer Zahlen hat, werden die Eigenschaften dieser Zahlen durch die Klasse `Complex` nachgebildet. Den Kern von `Complex`-Objekten bilden zwei Gleitkomma-Zahlen. Die Klasse stellt nun Methoden zur Addition und Multiplikation bereit. Die komplexe Exponentialfunktion ist notwendig für die Quanten-Fourier-Transformation und wird daher in einer speziellen Form implementiert. Dies ist einfach, da Java mit der Klasse `java.lang.Math` Konstanten wie π und Funktionen wie sin und cos zur Verfügung stellt. Um Speicher zu sparen werden oft vorkommende Zahlen wie 0, 1 und $\frac{1}{\sqrt{2}}$ als Konstanten definiert.
Die Klasse `SquareMatrix` besteht im Kern aus einer zweidimensionalen Matrix von `Complex`-Objekten. Die Implementation des Matrix- und des Tensorproduktes folgt direkt aus der mathematische Definition. Mit dieser Klasse ist es daher auf sehr natürlich Art und Weise möglich, das mathematische Objekt einer quadratischen Matrix zu benutzen. Diese Einfachheit wird aber erst deutlich durch die Funktion der Klasse als Repräsentant für Gatter.
Die Klasse `Gate` stellt ein solches Gatter dar. Ein Gatter ist eine unitäre Transformation und damit eine spezielle quadratische Matrix. Spezialisierungen werden in der Objektorientierung durch Vererbung modelliert. Die Klasse `Gate` hat daher alle Eigenschaften der Klasse `SquareMatrix` mit bestimmten Veränderungen und Erweiterungen. So soll ein Gatter auf ein Register wirken können, daher enthält die Klasse `Gate` die Methode `operateOn(Register reg)`. Die Methoden zur Matrix- und Tensormultiplikation werden so verändert, dass sie wieder Gatter zurückliefern — eine reine Komfortangelegenheit.
Bei der Modellierung von speziellen Gattern offenbart sich, wie nützlich dieses Konzept ist: Die Klasse `Gate` vererbt alle Eigenschaften eines Gatters an das spezielle Gatter. Nur die Besonderheiten müssen implementiert werden. Die Klasse `NotGate` beispielsweise repräsentiert die Nicht-Operation:

```
package components;

/**
 * Diese Klasse repraesentiert das Nicht-Gatter (NOT)
 * @author Nils Rosemann
 *
 */
public class NotGate extends Gate {
public NotGate() {
super(2);
values[0][0] = Complex.Zero;
values[0][1] = Complex.One;
values[1][0] = Complex.One;
values[1][1] = Complex.Zero;
}
```

```
}
```

Die einzigen Informationen, die genannt werden, sind die Dimension 2 des Nicht-Gatters und die Matrixeinträge. Das reicht völlig aus!
Mit der Anweisung `NotGate not = new NotGate()` erzeugt man ein solches Nicht-Gatter: Der Befehl `new NotGate()` spezifiziert den Konstruktor und das Ergebnis (das neue Objekt) wird in der Variablen `not` gespeichert. Dieses Gatter kann man durch die beiden Verknüpfungen `tensor` und `matrix` mit anderen Gattern zu einem Schaltkreis verbinden und schließlich mit `operateOn` auf ein Register wirken lassen:

```
// Not-Gatter und Zwei-Qubit-Register erzeugen
NotGate not = new NotGate();
Register reg = new Register(2);

// Zwei Not-Gatter parallel schalten
Gate g = not.tensor(not);

// Auf Register wirken lassen
g.operateOn(reg);
```

Die letzte grundlegende Klasse ist `Register`, welche schon im letzten Beispiel benutzt wurde. Dieser Klasse wird die Anzahl der Qubits übergeben, wenn ein neues Objekt erzeugt wird. Dies entspricht der Stelle `Register reg = new Register(2)`, bei der ein Register mit zwei Qubits konstruiert wird. Zu Testzwecken kann jederzeit der vollständige Registerzustand durch die `show()`- oder `showStatistic()`-Operationen angezeigt werden. Da dies in der quantenmechanischen Wirklichkeit nicht möglich ist, sollte man sich auf die Methode `measure()` beschränken, die einen zufälligen Messwert zurückliefert. Die Wahrscheinlichkeiten entsprechen sinnvollerweise den quantenmechanischen Wahrscheinlichkeiten: den Betragsquadraten der komplexen Amplituden. Für manche Anwendungen ist die Operation `mergeWith(Register reg)` wichtig; sie verbindet zwei Register $|a\rangle$ und $|b\rangle$ zu einem neuen: $|a\rangle$`.mergeWith(`$|b\rangle$`)` $= |a\rangle \otimes |b\rangle$.
Der Vektor, der das Register beschreibt, kann mit beliebigen Werten besetzt werden (über die `set()`-Methode). Man verlangt von einem Quantenregister gewöhnlich aber nur, dass es in den Zustand $|0\rangle$ präparierbar ist. Dies ist daher die Standardeinstellung.

5.2.2. Spezielle Gatter

Zentral für die meisten Schaltkreise sind kontrollierte Gatter. Zur einfachen Nutzung dieser Gatter gibt es die Klassen `ControlledGate` und `ReverseControlledGate`. Sie unterscheiden sich nur darin, wo das kontrollierende Qubit liegt, wenn man den Schaltkreise grafischen darstellt: Liegt es oberhalb von dem Gatter, dann benutzt man die Klasse `ControlledGate`, liegt es unterhalb, die andere.
Klassen für spezielle kontrollierte Gatter sehen nun sehr einfach aus. Das CNOT-Gatter ist im Kern ein Ein-Zeiler:

```
public class CNotGate extends ControlledGate {
        public CNotGate(int interiors) {
                super(interiors, new NotGate());
        }
}
```

Abgesehen von dem notwendigen Rahmen (`public class ...`) wird nur der Konstruktor der Mutterklasse (`super(...)`) mit den Informationen über Zwischenqubits und zu kontrollierendem `Not`-Gatter aufgerufen. Auch die anderen Klassen für kontrollierte Gatter sind so einfach. Im kompliziertesten Fall müssen die Einträge für das kontrollierte Gatter erst noch festgelegt werden:

```
public class RkGate extends ReverseControlledGate {
        public RkGate(int k) {
                Complex[][] val = new Complex[2][2];
                val[0][0]=Complex.One;
                val[0][1]=Complex.Zero;
                val[1][0]=Complex.Zero;
                val[1][1]=Complex.exp2pii(1.0/Math.pow(2,k));
                this.set(k-2, new Gate(val)); // anstatt super(...)
        }
}
```

Hier wird die `set`-Methode benutzt, um die Matrix zu erschaffen, denn der Konstruktor der Mutterklasse `super()` müsste vor allen andere Anweisungen kommen. Und das ist hier nicht möglich, da vorher die Matrix definiert werden muss. Ein Blick in den Quelltext von `ReverseControlledGate` zeigt, dass der Konstruktor dieser Klasse ebenfalls die `set`-Methode aufruft. Daher ist das Verfahren identisch.
Die einzigen kontrollierten Gatter im Schaltkreis des Shor-Algorithmus' sind kontrollierte Ein-Qubit-Gatter mit nur einem Kontrollbit. Mehrere Kontrollbits unterstützen meine Klassen daher nicht. Falls man mehrfach kontrollierte Gatter simulieren will, kann man im Notfall ein entsprechend großes `Gate`-Objekt erzeugen und ihm die Matrix-Darstellung übergeben.
Die Klasse `ControlledGate` konstruiert aus dem übergebenen Gatter und der Anzahl der Zwischen-Qubits die Matrix des gewünschten Gatters. Dies erfolgt über eine implizite Wertetabelle: Bei jedem Matrixeintrag wird geprüft, ob das kontrollierende Bit gesetzt ist oder nicht. Wenn ja, wird der entsprechende Eintrag des kontrollierten Gatters eingesetzt, ansonsten eine Eins.
Mit so einer impliziten Wertetabelle wird auch die Klasse `ModGate` implementiert. Sie repräsentiert die Abbildung

$$|a\rangle\,|b\rangle \longrightarrow |a\rangle\,|b \oplus x^a \,(\mathrm{mod}\, N)\rangle \quad . \tag{40}$$

Dabei sind $|a\rangle$ und $|b\rangle$ Register mit der gleichen Anzahl von Qubits. x, N und die Gesamtregisterbreite (von $|a\rangle$ und $|b\rangle$ zusammen) sind die Parameter für den Konstruktor von `ModGate`. Zur Erstellung der Matrixeinträge müssen die Werte a und b zunächst aus dem Zustand $|ab\rangle$ errechnet werden. Durch Division mit Rest kann man beide Informationen gewinnen. Mit diesen Informationen kann dann leicht berechnet werden, welche Matrixeinträge wie gesetzt werden müssen.
Kontrollierte Gatter und Gatter zur modularen Exponentiation zu erstellen, wie sie im Shor-Algorithmus vorkommen, ist damit automatisiert möglich. Das Programm findet aber zu einer gefundenen Matrix keine Zerlegung in elementare Gatter. Diese Aufgabe war im Rahmen meiner Bachelorarbeit aus zeitlichen Gründen nicht durchführbar.

5.2.3. Schaltkreise

Wie in Abschnitt 3 und 4 gezeigt wurde ist die Quanten-Fourier-Transformation der Schlüssel zum Shor-Algorithmus. Die Klasse `FourierTransformCircuit`, die diesen Schaltkreis implementiert, ist damit eine der wichtigsten. Nach der geleisteten Vorarbeit ist die Konstruktion aber eine geradlinige Aufgabe: Die Realisierung der QFT wurde in Abschnitt 3.2 vorgestellt und gipfelte in einem Schaltbild (Abbildung 11). Mit den vorhandenen Klassen muss nur noch dieses Schaltbild aufgebaut werden.
Dazu sind vier Arten von Gattern nötig: Identitäts-, Hadamard-, controlled-R_k- und SWAP-Gatter. Entsprechend gibt es die vier Klassen `IdentityGate`, `HadamardGate`, `RkGate` und `SwapGate`.
Der Schaltkreis wird schichtweise aufgebaut: Zu Beginn wirkt die erste Reihe von Gattern auf alle Qubits, dann die zweite Reihe, usw. In der Aufeinanderfolge von Schichten gibt es ein System. Die R_k-Gatter werden in fester Reihenfolge durchlaufen, aber vorher kommt immer ein Hadamard-Gatter. Es liegt daher in der logischen Struktur, das Hadamard-Gatter und die R_k-Gatter zu einem Feld `rkgates` zusammenzufassen:

```
Gate[] rkgates = new Gate[qubitNumber];

rkgates[0] = new HadamardGate(); // Hadamard-Gatter erschaffen

for(int i = 1; i<qubitNumber; i++) {
        rkgates[i] = new RkGate(i+1);   // R_k-Gatter erschaffen
}
```

Ober- und unterhalb eines Gatters aus diesem Feld passiert nichts. Dies wird durch ein Feld von Identitätsgattern modelliert:

```
Gate[] identities = new Gate[qubitNumber];
for(int i = 0; i<qubitNumber; i++) {
        // Identitaeten erschaffen
        identities[i] = new IdentityGate((int)Math.pow(2,i));
}
```

Nun wird in jeder Schicht zuerst eine passende Identität mit einem `rkgate` und dann wieder mit einer Identität multipliziert (Tensorprodukt). Das Ergebnis wird per Matrixmultiplikation an die vorhergehende Schicht angebaut. Die Anfangsschicht ist aus technischen Gründen eine Identität auf allen Qubits:

```
// Schaltkreis zusammenbauen
this.gate = identities[qubitNumber-1].tensor(identities[1]);
for(int i = 0; i<qubitNumber; i++) {
    for(int j = 0; j<qubitNumber-i; j++) {
        Gate helper = identities[0];
        helper = helper.tensor(identities[i]);
        helper = helper.tensor(rkgates[j]);
        helper = helper.tensor(identities[qubitNumber-j-i-1]);
        gate = gate.matrix(helper);
    }
}
```

Wie in Abschnitt 3 gezeigt wurde, haben die Qubits nach den bisherigen Gatter-Operationen die falsche Reihenfolge. Daher stellen Swap-Operationen die richtige Reihenfolge wieder her:

```
for(int i=0; i<qubitNumber/2; i++) {
    Gate helper = identities[0];
    helper=helper.tensor(identities[i]);
    helper=helper.tensor(new SwapGate(qubitNumber-2*i));
    helper=helper.tensor(identities[i]);
    this.gate = gate.matrix(helper);
}
```

Damit ist die Quanten-Fourier-Transformation vollständig. Der Vorteil der Objektorientierung ist hier klar erkennbar: Der Zusammenbau des Schaltkreises erfolgt völlig intuitiv und ist für jede beliebige Anzahl von Qubits durchführbar! Verbesserungen am Quelltext der Gatter oder der grundlegenden Klassen können durchgeführt werden, ohne dass die Klasse der QFT verändert werden müsste. Wäre an den klassischen Computer ein Quantencomputer-Modul angeschlossen, so könnte der Zugriff über die Gatter-Klassen gekapselt werden, und die QFT-Klasse würde in unveränderter Form einen Quantencomputer steuern und seine Vorteile nutzen können!
Die inverse Fourier-Transformation, wie sie für den Shor-Algorithmus benötigt wird, ist völlig analog zur Fourier-Transformation aufgebaut. Sie folgt sogar direkt, indem man konjugiert und transponiert. In einem Quantencomputer würde man aber nicht die Fourier-Transformation implementieren, um sie dann zu konjugieren, sondern man würde den Schaltkreis umbauen. Und den Weg gehe ich auch. Daher gibt es eine ganz

ähnliche Klasse `InverseFourierTransformCircuit`. Sie besteht fast aus den gleichen Klassen. Anstatt controlled-R_k-Gattern werden controlled-Q_k-Gatter verwendet:

$$Q_k = \begin{pmatrix} 1 & 0 \\ 0 & e^{-2\pi i/2^k} \end{pmatrix} \quad . \tag{41}$$

5.2.4. Shor-Algorithmus

Die Klasse `ShorAlgorithm` implementiert den Shor-Algorithmus so, wie man ihn auf einem Quantencomputer realisieren würde. Das bedeutet, dass bestimmte Teile durch klassische Algorithmen berechnet werden; nur der Kern, die Bestimmung der Ordnung, wird (simuliert) quantenmechanisch durchgeführt.
Abschnitt 4.4 führt die einzelnen Schritte auf. Sie sind aber nicht so linear im Quelltext zu finden, sondern geteilt:
Beim Start wird die Methode `main` ausgeführt. Über die Kommandozeile können dieser Methode als Parameter die zu faktorisierende Zahl N und die Anzahl der zu verwendenden Qubits übergeben werden. Ohne diese Parameter wird die Zahl 15 mit acht Qubits faktorisiert. Als nächstes wird ein `ShorAlgorithm`-Objekt mit der gewünschten Anzahl von Qubits erstellt und dann die zu faktorisierende Zahl an die Methode `factorize` übergeben. Liefert diese einen Faktor zurück, so wird er ausgegeben.
Im Konstruktor von `ShorAlgorithm` wird Vorarbeit geleistet. Denn die QFT und die Hadamard-Gatter sind unabhängig von der Zahl, die faktorisiert werden soll, und können daher schon in der nötigen Größe erstellt werden:

```
public ShorAlgorithm(int qubitNumber) {
    this.qubitNumber = qubitNumber;
    IdentityGate id = new IdentityGate((int)Math.pow(2, qubitNumber/2));
    this.prepare = HadamardGate.multipleH(qubitNumber/2).tensor(id);
    this.ift = (new InverseFourierTransformCircuit(
        qubitNumber/2)).getGate().tensor(id);
}
```

Die Qubit-Anzahl wird gespeichert und dann werden Identitätsgatter, Hadamard-Gatter und der Schaltkreis zur inversen Fourier-Transformation erzeugt und verknüpft. Denn die Hadamard-Gatter und die inverse QFT dürfen nur auf die erste Hälfte des Registers wirken und werden daher durch die Identität trivial auf die gesamte Registerbreite ausgedehnt (`tensor()`).
In der Methode `factorize` passiert nun die eigentliche Arbeit. Zuerst sortieren zwei Anweisungen gerade Zahlen, Primzahlen und Potenzen aus (Schritt 1 und 2 aus Abschnitt 4.4):

```
if(N%2==0)return 2;          // gerade Zahlen aussortieren
if(isPrimeOrPower(N)){       // Primzahlen und Potenzen aussortieren
```

```
    System.err.println("Primzahl oder Potenz!");
    System.exit(0);
}
```

Die Methode `isPrimeOrPower` testet durch naives Ausprobieren, ob eine Primzahl oder eine Potenz vorliegt. Bessere Algorithmen für diesen Test sind hier nicht nötig, da nur sehr kleine Zahlen untersucht werden.
Im dritten Schritt wird eine Zufallszahl x erzeugt. Falls sie einen gemeinsamen Teiler mit N hat, wird er zurückgegeben und der Algorithmus hat einen Glückstreffer gelandet:

```
x = (int)(N*Math.random()+1); // Zufallszahl
if(ggT(x, N)>1){ // Gl"uckstreffer!
    System.out.println("Zufallszahl hat schon gemeinsamen Faktor!");
    return ggT(x, N);
}
```

Die Methode `ggT(x, N)` ist der Euklidische Algorithmus und liefert daher den größten gemeinsamen Teiler von x und N. Falls das Ergebnis 1 ist, müssen die nächsten Schritte durchgeführt werden. Und jetzt kommt die Quantenmechanik ins Spiel (Schritt 4). Ein Register und ein `ModGate` werden erzeugt; das `ModGate`-Objekt passt den Schaltkreis an die Informationen x, N und Anzahl der Qubits an, denn die anderen Komponenten sind immer gleich. Daher kann es auch erst in der Methode `factorize` erzeugt werden: Vorher sind x und N nicht bekannt! Im Quelltext:

```
Register reg = new Register(qubitNumber); // Register vorbereiten
ModGate mod = new ModGate(x, N, qubitNumber); // Modulare Exponentiation
System.out.println("Gatter bereit");

prepare.operateOn(reg); // Gatter nach einander anwenden
mod.operateOn(reg);
ift.operateOn(reg);

System.out.println("Gatter angewendet");

measure = reg.measure()/((int)Math.pow(2, qubitNumber/2)); // Messung
System.out.println("Zufallszahl: " + x + " Messung: " + measure);
```

Nacheinander werden die Gatter auf das Register angewendet. Der abschließende Schritt des quantenmechanischen Teils ist die Messung. Diese liefert eine Zahl zurück, die dem Gesamtzustand entspricht. Interessant ist aber nur der Zustand des ersten Registers, daher wird der Messwert durch die Dimension des halben Registers geteilt. Wie in Abschnitt 4.3 beschrieben muss dieses Messergebnis mit der Kettenbruchentwicklung ausgewertet werden. Dazu verwende ich die zwei Klassen `Fraction` und `ContinuedFractions`.

Beide wurden nicht von mir geschrieben, sondern sind freie Software. `Fraction` taucht nur als Hilfsklasse für `ContinuedFractions` auf, welche aus einem Bruch einen Kettenbruch machen kann und umgekehrt. Die Kettenbruchentwicklung liefert mit hoher Wahrscheinlichkeit die gesuchte Ordnung r:

```
//  Kettenbruchentwicklung
Vector vec = ContinuedFractions.to(new Fraction(
              measure, (long)Math.pow(2, qubitNumber/2)));

long r = ((Long)vec.lastElement()).longValue();
System.out.println("Kettenbruchentwicklung ergibt r="+ r);
```

In Schritt 5 wird diese Information schließlich ausgewertet:

```
// verschiedene F"alle ausprobieren, einer davon sollte Faktor sein
long test;
test = ggT((long)Math.pow(x, r/2)+1,N);
if(check(test,N))faktor=test;
test = ggT((long)Math.pow(x, r/2)-1,N);
if(check(test,N))faktor=test;
test = ggT((long)Math.pow(x, r)+1,N);
if(check(test,N))faktor=test;
test = ggT((long)Math.pow(x, r)-1,N);
if(check(test,N))faktor=test;
```

Dabei werden verschiedene Möglichkeiten ausprobiert. Denn wenn sich durch r kein Faktor ergibt, könnte $2r$ einen ergeben. Zur Verbesserung könnte man hier noch weitere Fälle ausprobieren. Zur Demonstration ist das aber unübersichtlich, wenn an dieser Stelle noch zehn weitere Fälle und beliebig viele weitere Kettenbruchentwicklungen stehen.
Falls noch kein Faktor gefunden wurde, wird ab Schritt 3 wiederholt. Ansonsten wird der Faktor zurückgegeben:

```
return faktor;  // Faktor zur"uckgeben
```

Damit ist der Shor-Algorithmus implementiert.

5.3. Ergebnisse

Der Shor-Algorithmus faktorisiert natürliche Zahlen. Wenn in einem Programm ein Faktorisierungsalgorithmus implementiert wird, erwartet man natürlich, dass das Programm Zahlen faktorisieren kann. Obwohl die Klasse `ShorAlgorithm` das kann, ist das kein wirklicher Fortschritt. Denn selbst ein simpler Ausprobier-Algorithmus kann größere Zahlen faktorisieren als das oben vorgestellte Programm.

Das Ergebnis ist also nicht, dass man Zahlen faktorisieren kann, die man vorher nicht faktorisieren konnte. Stattdessen kann man den Shor-Algorithmus in Größenordnungen untersuchen, die man von Hand nicht untersuchen kann. Denn schon bei dem kleinsten nicht-trivialen Beispiel, der Zahl 15, ist die Rechnung von Hand sehr mühselig. Ein klassischer Computer kann dagegen schon in wenigen Sekunden die Wahrscheinlichkeiten des Endzustands berechnen.
Wenn man zu Testzwecken die Wahrscheinlichkeiten mit ausgibt, könnte eine mögliche Ausgabe des Programms so aussehen:

```
Gatter bereit
Gatter angewendet
Zufallszahl: 2 Messung: 4
0   0.24999999999999967
1   1.1801731080250095E-34
2   0.0
3   1.0598024865008074E-34
4   0.24999999999999967
5   1.1801731080250095E-34
6   0.0
7   2.1720813583139163E-35
8   0.24999999999999967
9   1.1801731080250095E-34
10   0.0
11   1.0598024865008074E-34
12   0.24999999999999967
13   1.1801731080250095E-34
14   0.0
15   2.1720813583139163E-35
Kettenbruchentwicklung ergibt r=4
Faktorisierung: 15 = 3 * 5
```

Die Zahl 15 wurde also erfolgreich faktorisiert mit der Zufallszahl 2. Hierbei sind die Wahrscheinlichkeiten bei den gewünschten Zuständen maximal, also 0.25. Die Chance, einen nicht erwünschten Zustand zu messen, ist vernachlässigbar klein. Und aus dem Ergebnis wird die richtige Ordnung $r = 4$ ermittelt:

$$x^r = 2^4 = 16 = 1 \,(\mathrm{mod}\, 15) \ . \tag{42}$$

Damit ergeben sich die Faktoren: $x^{r/2} - 1 = 4 - 1 = 3$ und $x^{r/2} + 1 = 4 + 1 = 5$. Der Algorithmus war erfolgreich. In vielen Fällen ergibt sich dagegen die Ausgabe:

```
Zufallszahl 10 hat gemeinsamen Faktor!
Faktorisierung: 15 = 5 * 3
```

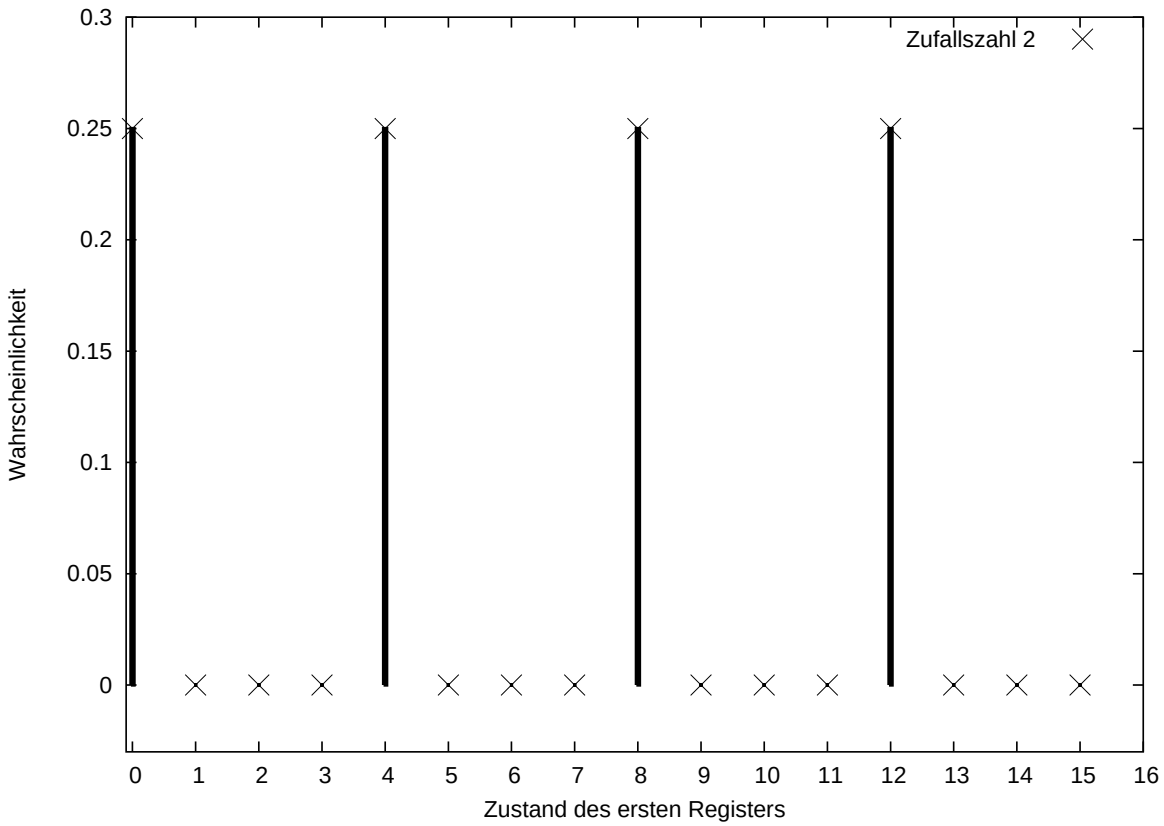


Abbildung 13: Mess-Wahrscheinlichkeiten bei der Faktorisierung von 15, wenn die Zufallszahl 2 ist. Die Wahrscheinlichkeiten bei 0, 4, 8 und 12 sind genau 0.25.

Denn von den Zahlen, die kleiner als 15 und größer als 1 sind, haben sechs schon einen gemeinsamen Teiler: 3, 5, 6, 9, 10 und 12. Bei ihnen kommt der Shor-Algorithmus also gar nicht in die quantenmechanische Phase.
Manchmal kann die Ordnung auch nicht bestimmt werden oder die Ordnung ist ungerade. Dann wird die Prozedur wiederholt:

```
Gatter bereit
Gatter angewendet
Zufallszahl: 8 Messung: 2
0   0.21874999999999972
1   0.0904859792057418
2   0.03448604345603976
3   0.02001292438186046
4   0.12499999999999985
5   0.024966445250020057
```

```
6   0.012388956543960179
7   0.020784651162377432
8   0.12499999999999985
9   0.02078465116237744
10   0.012388956543960179
11   0.02496644525002005
12   0.12499999999999985
13   0.02001292438186046
14   0.03448604345603976
15   0.0904859792057418
Kettenbruchentwicklung ergibt r=8

Gatter bereit
Gatter angewendet
Zufallszahl: 4 Messung: 8
0   0.4999999999999992
1   6.018531076210112E-35
2   5.783200004039984E-35
3   0.0
4   0.0
5   4.8148248609680896E-35
6   9.683751430718938E-36
7   8.305572885169955E-34
8   0.4999999999999992
9   6.018531076210112E-35
10   5.783200004039984E-35
11   0.0
12   0.0
13   4.8148248609680896E-35
14   9.683751430718938E-36
15   8.305572885169955E-34
Kettenbruchentwicklung ergibt r=2
Faktorisierung: 15 = 3 * 5
```

Mit den optionalen Parametern kann man das Programm auch andere Zahlen faktorisieren lassen. Zahlen, die kleiner als 15 sind, sind aber uninteressant, da sie entweder durch 2 teilbar, eine Primzahl oder eine Potenz sind. Mit acht Qubits (also vier pro Register) sind größere Zahlen nicht darstellbar. Um mehr Qubits zur Verfügung zu haben, kann man den zweiten Parameter benutzen. Er sollte ein Vielfaches von 2 sein, da die angegebene Anzahl zu gleichen Teilen auf beide Register verteilt wird.
Eine Erhöhung der Qubit-Anzahl lastet jeden Speicher schnell aus. Bei 12 Qubits werden bereits 1035 MB Speicher belegt. Und das Wachstum ist exponentiell: 14 Qubits würden

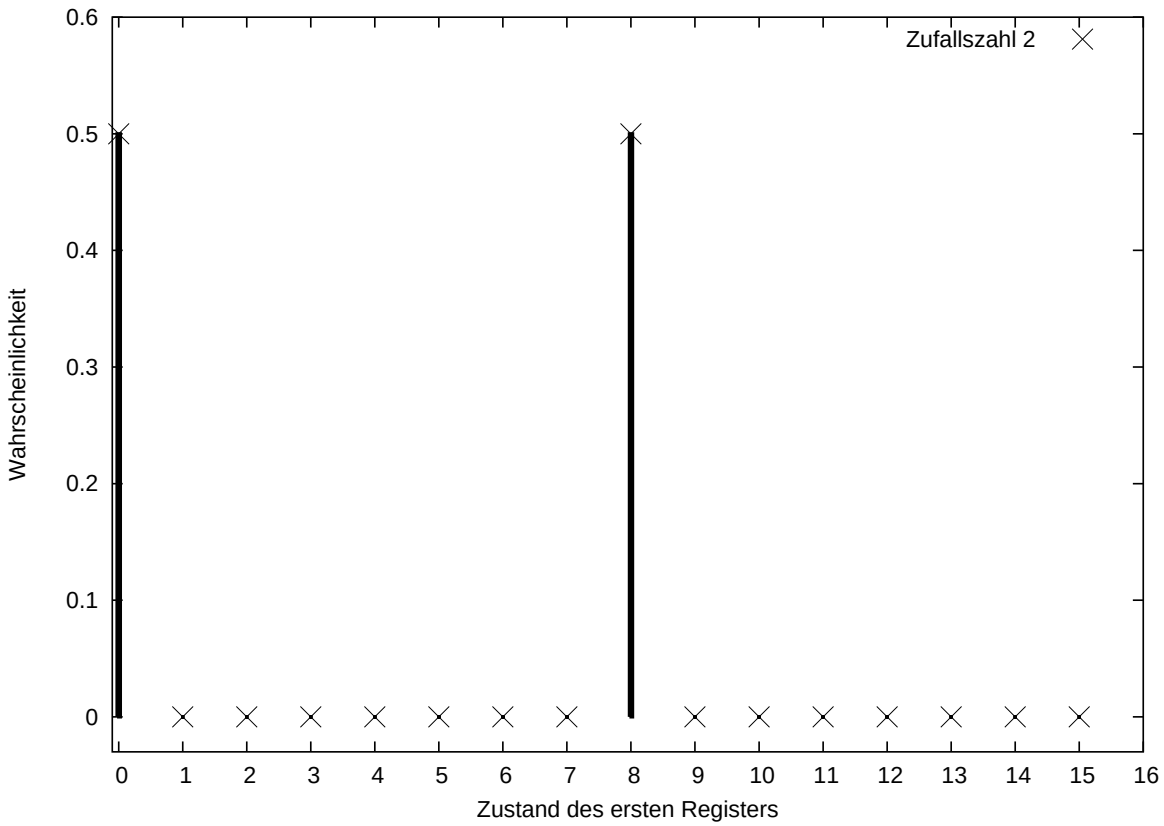


Abbildung 14: Mess-Wahrscheinlichkeiten bei der Faktorisierung von 15, wenn die Zufallszahl 4 ist. Bei 0 und 8 sind die höchsten Wahrscheinlichkeiten.

ca. den 16fachen Speicher brauchen! Hauptgrund hierfür ist nicht das Register, welches exponentiell mit der Basis 2 wächst, sondern die Darstellung der Gatter mit der Basis 4. Jedes Qubits mehr vervierfacht den Speicher, zwei Qubits mehr versechzehnfachen also. Die Implementation des Shor-Algorithmus speichert drei Gatter: `prepare`, `mod` und `ift`. Nach (38) und (39) ist der Speicherverbrauch:

$$S(n) \;=\; (2^{n+4} + 3 \cdot 4^{n+2})\,\text{Bytes} \quad . \tag{43}$$

Dies ergibt für 12 Qubits:

$$S(12) \;=\; (65536 + 3 \cdot 268435456)\,\text{Bytes} \;\approx\; 800\,\text{MB} \quad . \tag{44}$$

Hilfsvariablen, die Verwaltung der Objekte und die Java-Bibliotheken brauchen ebenfalls Speicher, so dass der beobachtete Bedarf von 1035 MB erklärbar ist.
Damit Java so viel Speicher reservieren kann, muss beim Start ein weiteres Kommando angegeben werden. Der Aufruf, um die Zahl 33 zu faktorisieren, sieht damit so aus:

```
java -Xmx1024m -jar shor.jar 33 12
```

Durch die Option `-Xmx1024m` kann Java genügend Speicher reservieren, um 12 Qubits zu simulieren. Und damit kann 33 (bzw. jede Zahl zwischen 2 und 63) faktorisiert werden. Für noch mehr Qubits braucht man eine Großrechneranlage.

6. Zusammenfassung

Der Shor-Algorithmus zur Faktorisierung natürlicher Zahlen lässt sich mit Hilfe der objektorientierten Programmierung intuitiv implementieren, indem man grundlegende Gatter- und Registerklassen benutzt. Denn sobald diese Bauteile zur Verfügung stehen, kann man sie genau wie in der abstrakten Algorithmus-Beschreibung zusammensetzen.

Die erstellte Klasse `ShorAlgorithm` kann somit den Verlauf des Shor-Algorithmus' berechnen und schließlich kleine Zahlen faktorisieren. Da die Simulation exponentielle Komplexität besitzt, erfordern selbst kleine Zahlen schon großen Speicher- und Rechenaufwand.

Warum besitzt der Shor-Algorithmus auf einem Quantencomputer polynomielle Komplexität? Die Antwort liegt in der Art und Weise, wie in der Quantenmechanik Teilsysteme zusammengefasst werden: Unsere Intuition fasst durch die *Vereinigung* zusammen, die Quantenmechanik durch das *Tensorprodukt*. In der klassischen Mechanik braucht man im Allgemeinen $3 + 3 = 6$ Koordinaten, um ein System zu beschreiben, das aus zwei Teilsystemen mit je 3 Koordinaten besteht. Ein solches System ist eine Vereinigung der Teilsysteme. In der Quantenmechanik ist die Zusammenfassung zweier 3-dimensionaler Systeme $(3 \cdot 3 = 9)$-dimensional. Ein solches System ist das Tensorprodukt der Teilsysteme.

Anders ausgedrückt hat die Zusammenfassung von zwei klassischen Systemen mit je N möglichen Zuständen $N \cdot N$ mögliche Zustände. Eine quantenmechanische Zusammensetzung hat dagegen $c^{N \cdot N}$ mögliche Zustände (c ist eine Konstante). Ein quantenmechanisches System trägt also exponentiell mehr Information.

Die Quanten-Fourier-Transformation nutzt diese Eigenschaft aus und kann damit exponentiell schneller arbeiten als jeder bekannte Algorithmus zur Fourier-Transformation. Der Shor-Algorithmus führt nun die Faktorisierung auf die Fourier-Transformation zurück und erreicht damit polynomielle Komplexität.

Die quantenmechanische Art, Systeme zusammenzufassen, entspricht nicht unserer Intuition und auch nicht dem objektorientierten Paradigma. Da wir unserer Intuition durch mathematische Hilfsmittel diese andere Art der Zusammenfassung näher bringen können, gibt es vielleicht ein Programmier-Paradigma, welches es erleichtert, Quantensoftware herzustellen.

7. Abstract

Shor's algorithm for factoring natural numbers can be implemented intuitively via object oriented programming by using basic classes for gates and registers. One can assemble these components exactly the same way as in the abstract definition.
This way the class `ShorAlgorithm` can calculate the steps of Shor's algorithm and finally factorise small natural numbers. But even small numbers require very much memory and processor power: The simulation has exponential complexity.
Why has Shor's algorithm polynomial complexity on a quantum computer? The answer is the way quantum mechanics merges subsystems: Our Intuition merges by taking the *union*, quantum mechanics by taking the tensor product. In classical mechanics one usually needs $3 + 3 = 6$ coordinates to describe a system of two subsytems with 3 coordinates each. In this case, the system is a union of the subsystems. But in quantum mechanics this system would be $(3 \cdot 3 = 9)$-dimensional. It would be the tensor product of the subsystems.
To put it another way, the union of two classical subsystems with N possible states each would have $N \cdot N$ possible states whereas in quantum mechanics the system would have $c^{N \cdot N}$ possible states (for c beeing constant). Thus quantum mechanical systems carry exponentially more information.
The quantum Fourier transform takes advantage of this capacity and this way can work exponentially faster than any known algorithm for the Fourier transform. The idea of Shor's algorithm is to reduce factoring to applying the quantum Fourier transform and thus to archieve polynomial complexity.
The quantum mechanical way of combining systems does not match the way of our intuition and not the way of the object oriented paradigm. And because we can to some degree acquire the quantum mechanical way by mathematical means, maybe there exists a programming paradigm which makes it easier to produce quantum software.

A. Quellcode

A.1. Grundlegende Klassen

A.1.1. Complex.java

```
package components;

/**
 * Diese Klasse repraesentiert komplexe Zahlen
 *
 * @author Nils Rosemann
 *
 */
```

```
public class Complex {
/** Null */
public static final Complex Zero = new Complex(0,0);
/** Eins */
public static final Complex One = new Complex(1,0);
/** Wurzel von 0.5 */
public static final Complex SqrtHalf = new Complex(1.0/Math.sqrt(2),0);
/** imaginaere Einheit */
public static final Complex I = new Complex(0,1);
// Realteil
protected double real = 0;
// Imaginaerteil
protected double imaginary = 0;
/** Erschafft komplexe Zahl
 * @param real Realteil
 * @param imaginary Imaginaerteil
 *
 */
public Complex(double real, double imaginary) {
 this.real = real;
 this.imaginary = imaginary;
}
/**
 * Erschafft komplexe Null
 *
 */
public Complex() {
 this(0, 0);
}
/**
 * Setzt Real- und Imaginaerteil
 * @param real Realteil
 * @param imaginary Imaginaerteil
 */
public void set(double real, double imaginary) {
 this.real = real;
 this.imaginary = imaginary;
}
/**
 * Setzt den Realteil
 * @param real Realteil
 */
```

```
public void setReal(double real) {
this.real = real;
}
/**
 * Setzt den Imginaerteil
 * @param imaginary Imaginaerteil
 */
public void setImg(double imaginary) {
 this.imaginary = imaginary;
}
/**
 *
 * @return Realteil
 */
public double getReal() {
 return this.real;
}
/**
 *
 * @return Imginaerteil
 */
public double getImg() {
 return this.imaginary;
}
/**
 *
 * @return Absolutbetrag der komplexen Zahl
 */
public double abs() {
 return Math.sqrt(real*real + imaginary*imaginary);
}
/**
 * Stringdarstellung
 */
public String toString() {
 if(imaginary==0)return ""+ real; else
 return "(" + real + " + " + imaginary + "i)";
}
/**
 * Berechnet das Produkt von sich selbst mit der Zahl c
 * @param c Faktor
 * @return Produkt
```

```
 */
public Complex mult(Complex c) {
 return new Complex(real*c.getReal()-imaginary*c.getImg(),
  real*c.getImg() + imaginary*c.getReal());
}
/**
 * Berechnet die Summe von sich selbst mit der Zahl c
 * @param c Summand
 * @return Summe
 */
public Complex add(Complex c) {
 return new Complex(real+c.getReal(),imaginary+c.getImg());
}
/**
 * Berechnet exp(2 \pi i d), wobei d ein Parameter ist.
 * @param d Parameter
 * @return Wert der komplexen e-Funktion
 */
public static Complex exp2pii(double d) {
 return new Complex(Math.cos(2*Math.PI*d),Math.sin(2*Math.PI*d));
}
/**
 * Berechnet exp(-2 \pi i d), wobei d ein Parameter ist.
 * @param d Parameter
 * @return Wert der komplexen e-Funktion
 */
public static Complex expminus2pii(double d) {
return new Complex(Math.cos(-2*Math.PI*d),Math.sin(-2*Math.PI*d));
}
}
```

A.1.2. SquareMatrix.java

```
package components;

/**
 * Diese Klasse repraesentiert quadratische Matrizen
 * mit komplexen Eintraegen.
 * @author Nils Rosemann
 *
 */
public class SquareMatrix {
```

```
/**
 * Zwei-dimensionale Matrix der komplexen Eintraege.
 */
protected Complex[][] values;
/**
 * Dimension der Matrix
 */
protected int dimension;
/**
 * Erschafft leere Matrix der Dimension dimension x dimension.
 * @param dimension Dimension der Matrix
 */
public SquareMatrix( int dimension ) {
this.dimension = dimension;
this.values = new Complex[dimension][dimension];
}
/**
 * Erschafft Matrix mit values als Eintraegen.
 * @param values Eintraege der Matrix
 */
public SquareMatrix( Complex[][] values ) {
set(values);
}
/**
 * Erschafft Matrix aus uebergebener Matrix, erstellt also eine Kopie.
 * @param matrix die zu kopierende Matrix
 */
public SquareMatrix( SquareMatrix matrix ) {
set(matrix.getValues());
}
/**
 * Setzt die Eintraege der Matrix auf values.
 * @param values die neuen Eintraege
 */
public void set( Complex[][] values ) {
this.values = values;
this.dimension = values.length;
}
/**
 * Gibt das Element mit der Position (i,j) zurueck.
 * @param i Zeile
 * @param j Spalte
```

```
 * @return Eintrag (i,j)
 */
public Complex get( int i, int j ) {
return values[i][j];
}
/**
 * Gibt alle Werte als Matrix zurueck.
 * @return Matrix der Eintraege
 */
public Complex[][] getValues() { return values; }
/**
 * Dimension der Matrix auslesen
 * @return Dimension der Matrix
 */
public int getDimension() { return this.dimension; }
/**
 * Berechnet das Matrixprodukt mit matrix.
 * @param matrix Ziel
 * @return Produktmatrix
 */
public SquareMatrix matrix ( SquareMatrix matrix ) {
  // Wenn Dimensionen nicht gleich, undefiniertes Ergebnis
  if(matrix.getDimension() != this.getDimension())return null;
  else {
    Complex[][] helper = new Complex[getDimension()][getDimension()];
    for(int i=0;i<getDimension();i++)
      for(int j=0;j<getDimension();j++) {
        Complex c = new Complex();
        for(int k=0;k<getDimension();k++) {
          // neuen Eintrag berechnen
          c = c.add(this.get(i,k).mult(matrix.get(k,j)));
        }
        helper[i][j]=c;
}
    return new SquareMatrix(helper);
  }
}
/**
 * Berechnet das Tensorprodukt mit matrix.
 * @param matrix Zielmatrix
 * @return Produktmatrix
 */
```

```
  public SquareMatrix tensor (SquareMatrix matrix) {
    //Dimension
    int newDim = this.getDimension()*matrix.getDimension();
    int mDim = matrix.getDimension();
    int tDim = this.getDimension();
    Complex[][] helper = new Complex[newDim][newDim];
    for(int i=0;i<tDim;i++) {
      for(int j=0;j<tDim;j++) {
        for(int k=0;k<mDim;k++) {
          for(int l=0;l<mDim;l++) {
            // neuen Eintrag berechnen
            helper[i*mDim+k][j*mDim+l]=this.get(i,j).mult(matrix.get(k,l));
          }
        }
      }
    }
    return new SquareMatrix(helper);
  }
  /**
   * Stellt die Matrix minimal formatiert dar
   *
   */
  public void show() {
    for(int i=0;i<this.getDimension();i++){
      for(int j=0;j<this.getDimension();j++) {
        System.out.print(this.get(i,j).toString()+" ");
      }
      System.out.println();
    }
  }
}
```

A.1.3. Gate.java

```
package components;

/**
 * Diese Klasse simuliert ein Quantengatter.
 * @author Nils Rosemann
 *
```

```
 */
public class Gate extends SquareMatrix {
/**
 * @param dimension Dimension der beschreibenden Matrix
 */
public Gate(int dimension) {
super(dimension);
}
/**
 * @param values Eintraege der beschreibenden Matrix
 */
public Gate(Complex[][] values) {
super(values);
}
/**
 * @param matrix beschreibende Matrix
 */
public Gate(SquareMatrix matrix) {
super(matrix);
}
/**
 * @return Dimension der beschreibenden Matrix
 */
public int getDimension() { return dimension; }
/**
 * Laesst das Gatter auf Register reg operieren.
 * @param reg
 */
  public void operateOn( Register reg ) {
    if(reg.getDimension()==this.getDimension()) {
      Complex[] helper = new Complex[getDimension()];
      for(int j=0;j<getDimension();j++) {
        Complex c = new Complex();
        for(int k=0;k<getDimension();k++) {
          c = c.add(this.get(j,k).mult(reg.get(k)));
        }
        helper[j]=c;
      }
      reg.set(helper);
    }
  }
/**
```

```
 * Setzt die beschreibende Matrix
 * @param matrix neue Matrix
 */
public void set(SquareMatrix matrix) {
this.values = matrix.getValues();
}
/**
 * Berechnet die Parallelschaltung mit Gatter g
 * @param g Gatter
 * @return Ergebnisgatter
 */
public Gate tensor(Gate g) {
return new Gate(super.tensor(g));
}
/**
 * Berechnet die Hintereinanderschaltung mit Gatter g
 * @param g Gatter
 * @return Ergebnisgatter
 */
public Gate matrix(Gate g) {
return new Gate(super.matrix(g));
}
/**
 * Gleicht numerische Ungenauigkeiten aus.
 */
  public void chop() {
    for(int i=0;i<getDimension();i++) {
      for(int j=0;j<getDimension();j++) {
        if(values[i][j].abs()<1E-15)values[i][j]=Complex.Zero;
        if(Math.abs(values[i][j].getReal()-1)<1E-15)
          values[i][j]=Complex.One;
      }
    }
  }
}
```

A.1.4. Register.java

```
package components;

/**
 * Diese Klasse simuliert ein Quantenregister.
```

```
 * @author Nils Rosemann
 *
 */
public class Register {
/**
 * Anzahl der Qubits im Register
 */
protected int qubitNumber;
/**
 * Dimension des Vektors zur Beschreibung des Registers
 */
protected int dimension;
/**
 * Vektor aus komplexen Zahlen
 */
protected Complex[] values;
/**
 * Erschafft Register mit qubitNumber Qubits. Im Anfangszustand sind
 * alle Qubits im Zustand \ket{0}
 * @param qubitNumber Anzahl der Qubits
 */
public Register ( int qubitNumber ) {
 this.dimension=(int)Math.pow(2, qubitNumber);
 this.values = new Complex[dimension];
 this.qubitNumber = qubitNumber;
 for ( int i = 1; i<dimension; i++ ) {
  this.values[i] = Complex.Zero;
 }
 this.values[0] = Complex.One;
}
/**
 * Erschafft Register aus komplexem Vektor values
 * @param values Vektor
 * @param qubitNumber Anzahl der Qubits
 */
public Register ( Complex[] values, int qubitNumber ) {
 this.values = values;
 this.qubitNumber = qubitNumber;
 this.dimension = values.length;
}
/**
 *
```

```
 * @return Anzahl der Qubits
 */
public int getQubitNumber() { return this.qubitNumber; }
/**
 *
 * @return Dimension
 */
public int getDimension() { return this.dimension; }
/**
 * Koeffizient in \ket{i}-Richtung
 */
public Complex get(int i) {
 return this.values[i];
}
/**
 * Setzt den Eintragsvektor auf values
 * @param values neue Eintraege
 */
public void set (Complex[] values) {
 this.values = values;
}
/**
 * Setzt den i-ten Eintrag des Registers auf c
 * @param i Index
 * @param c neuer Eintrag
 */
public void set (int i, Complex c) {
 this.values[i] = c;
}
/**
 * Zeigt die Betragsquadrate des Registers an.
 * Quantenmechanisch nicht erlaubt!
 *
 */
public void show() {
 for(int i=0;i<this.dimension;i++) {
  System.out.println(this.values[i].abs()*this.values[i].abs());
 }
}
/**
 * Fasst immer part Eintraege zusammen, berechnet die Betragsquadrate
 */
```

```
public void showStatistic(int part) {
 double[] results = new double[getDimension()/part];
 for(int i=0; i<getDimension(); i++) {
  results[i/part]+=values[i].abs()*values[i].abs();
 }
 for(int i=0; i<getDimension()/part; i++)
  System.out.println(i + "   " + results[i]);
}
/**
 * Verbindet Register mit register zu einem neuen.
 * @param register Zielregister
 * @return neues Register
 */
public Register mergeWith(Register register) {
 Register helper = new Register(
   this.getQubitNumber()+register.getQubitNumber()
 );
 for(int i=0; i<getDimension(); i++) {
  for(int j=0;j<register.getDimension(); j++) {
   helper.set(i*register.getDimension()+j,
   this.get(i).mult(register.get(j)));
  }
 }
 return helper;
}
/**
 * Simuliert den Messprozess. Eine Zustandsaenderung (post-measurement-
 * state) ist noch nicht implementiert.
 * @return zufaelliger Messwert
 */
public int measure () {
 double val = Math.random();
 for(int i=0;i<getDimension(); i++)
  if((val-=values[i].abs()*values[i].abs())<0) return i;
 return getDimension();
}
}
```

A.1.5. Circuit.java

```
package components;
```

```
/**
 * Interface fuer parameterabhaengige Gatterkonstruktionen
 * @author Nils Rosemann
 *
 */
public interface Circuit {
public void operateOn( Register register );
public Gate getGate();

}
```

A.2. Spezielle Gatter

A.2.1. ControlledGate.java

```
package components;

/**
 * Diese Klasse ermoeglicht die einfache Handhabung
 * von kontrollierten Gattern.
 * @author Nils Rosemann
 *
 */
public class ControlledGate extends Gate {
/**
 * Erschafft ein controlled-g-Gatter.
 * Dabei wird g als Ein-Qubit-Gatter angenommen.
 * Das kontrollierende Qubit befindet sich im Schaltbild
 * oberhalb. Zwischen kontrollierendem Qubit und dem Gatter
 * liegen interiors andere Qubits.
 * @param interiors Anzahl der Qubits zwischen Kontrollbit und Gatter
 * @param g kontrolliertes Ein-Qubit-Gatter
 */
public ControlledGate(int interiors, Gate g) {
 super((int)Math.pow(2,2+interiors));
 this.set(interiors, g);
}
/**
 * Setzt die Werte neu.
 * @param interiors Anzahl der Qubits zwischen Kontrollbit und Gatter
 * @param gate kontrolliertes Ein-Qubit-Gatter
 */
```

```
public void set(int interiors, Gate gate) {
 this.dimension = (int)Math.pow(2,2+interiors);
 this.values = new Complex[dimension][dimension];
 for(int i = 0; i<getDimension(); i++) {
  for(int j = 0; j<getDimension(); j++) {
   if(i==j)values[i][j]=Complex.One;
   else values[i][j]=Complex.Zero;
  }
 }
 for(int i = getDimension()/2; i<getDimension(); i++) {
  if(i%2==0) {
   values[i][i] = gate.get(0,0);
   values[i][i+1] = gate.get(0,1);
  } else {
   values[i][i-1] = gate.get(1,0);
   values[i][i] = gate.get(1,1);
  }
 }
}
/**
 * main-Methode zum Test
     */
public static void main(String[] args) {
  Complex[][] val = new Complex[2][2];
  val[0][0]=new Complex(11,0);
  val[0][1]=new Complex(12,0);
  val[1][0]=new Complex(21,0);
  val[1][1]=new Complex(22,0);
  ControlledGate g = new ControlledGate(2, new Gate(val));
  g.show();
 }
}
```

A.2.2. ReverseControlledGate.java

```
package components;

/**
 * Diese Klasse ermoeglicht die einfache Handhabung
 * von kontrollierten Gattern.
 * @author Nils Rosemann
 *
```

```
 */
public class ReverseControlledGate extends Gate {
/**
 * Erschafft ein controlled-gate-Gatter.
 * Dabei wird gate als Ein-Qubit-Gatter angenommen.
 * Das kontrollierende Qubit befindet sich im Schaltbild
 * unterhalb. Zwischen kontrollierendem Qubit und dem Gatter
 * liegen interiors andere Qubits.
 * @param interiors
 * @param gate
 */
public ReverseControlledGate (int interiors, Gate gate) {
 super((int)Math.pow(2,2+interiors));
 this.set(interiors, gate);
}
/**
 * Hilfskonstruktor fuer RkGate
 *
 */
public ReverseControlledGate() {
 super(2);
}
/**
 * Setzt die Werte neu.
 * @param interiors Anzahl der Qubits zwischen Kontrollbit und Gatter
 * @param gate kontrolliertes Ein-Qubit-Gatter
 */
public void set(int interiors, Gate gate) {
  this.dimension = (int)Math.pow(2,2+interiors);
  this.values = new Complex[dimension][dimension];
  //Alle Eintraege durchlaufen
  for(int i = 0; i<getDimension(); i++) {
   for(int j = 0; j<getDimension(); j++) {
    //Entscheiden, welcher Wert richtig ist
    if(i<getDimension()/2 && j<getDimension()/2){
     if(i!=j)values[i][j]=Complex.Zero; else
     if(i%2==1)values[i][j]=gate.get(0,0); else
      values[i][j]=Complex.One;
    }
    if(i<getDimension()/2 && j>=getDimension()/2){
      if(i%2==1 && i==j-getDimension()/2)values[i][j]=gate.get(1,0); else
       values[i][j]=Complex.Zero;
```

```
    }
    if(i>=getDimension()/2 && j<getDimension()/2){
      if(i%2==1 && i==j+getDimension()/2)values[i][j]=gate.get(0,1); else
      values[i][j]=Complex.Zero;
    }
    if(i>=getDimension()/2 && j>=getDimension()/2){
     if(i!=j)values[i][j]=Complex.Zero; else
      if(i%2==1)values[i][j]=gate.get(1,1); else
       values[i][j]=Complex.One;
    }
   }
  }
 }

}
```

A.2.3. HadamardGate.java

```
package components;

/**
 * Diese Klasse repraesentiert das Hadamard-Gatter
 * @author Nils Rosemann
 *
 */
public class HadamardGate extends Gate {
/**
 * Erschafft neues Hadamard-Gatter
 *
 */
public HadamardGate() {
 super(2);
 Complex c1 = Complex.SqrtHalf;
 Complex c2 = c1.mult(new Complex(-1,0));
 Complex[][] val = {{c1,c1},{c1,c2}};
 this.values = val;
}
/**
 * Erschafft number parallel geschaltete Hadamard-Gatter
 * @param number Anzahl der H-Gatter
 * @return Gatterdarstellung der Parallelschaltung
 */
```

```
public static Gate multipleH (int number) {
 Gate h = new HadamardGate();
 Gate g = new HadamardGate();
 for(int i=2;i<=number;i++)g=g.tensor(h);
 return g;
}
/**
 * main-Methode zum Testen.
 */
public static void main(String[] args) {
 Gate g = HadamardGate.multipleH(3);
 Register reg = new Register(3);
 g.operateOn(reg);
 reg.show();
}

}
```

A.2.4. IdentityGate.java

```
package components;

/**
 * Diese Klasse repraesentiert das Identitaetsgatter
 * @author Nils Rosemann
 *
 */
public class IdentityGate extends Gate {
/**
 * Erschafft ein Identitaetsgatter der Dimension dimension.
 * @param dimension Dimension des Gatters. Muss Potenz von 2 sein!
 */
public IdentityGate ( int dimension ) {
 super(dimension);
 for(int i=0;i<dimension;i++) {
  for(int j=0;j<dimension;j++) {
   if(i==j)values[i][j]=Complex.One;
   else values[i][j]=Complex.Zero;
  }
 }
}
```

```
}
```

A.2.5. ModGate.java

```
package components;

/**
 * Die Klasse repraesentiert Gatter
 * zur modularen Exponentiation fuer die Prozedur
 * zur Bestimmung der Ordnung
 * @author Nils Rosemann
 *
 */
public class ModGate extends Gate {
 /**
  * Erschafft ein Gatter zur modularen Exponentiation.
  * @param x Basis fuer die Bildung der Potenzen
  * @param N Alle Berechnungen (mod N)
  * @param qubitNumber Anzahl der Qubits. Muss eine gerade Zahl sein!
  */
 public ModGate(int x, int N, int qubitNumber) {
  super((int)Math.pow(2, qubitNumber)); // Gatter erschaffen
  int firstHalf, secHalf;
  int divider = (int)Math.pow(2, qubitNumber/2); // geteilte Wirkung
  for(int i=0; i<getDimension(); i++) { // fuer jede Zeile
   firstHalf = i / divider; // Wert des ersten Registerteils
   secHalf = i % divider;  // Wert des zweiten Registerteils
   int point = firstHalf*divider +  // hier muss die Eins gesetzt werden
      (secHalf+(int)Math.pow(x, firstHalf)%N)%divider;
   // Zeile mit Nullen und Einsen fuellen
   for(int j=0; j<getDimension(); j++) {
    if(j==point)values[j][i]=Complex.One; else
     values[j][i]=Complex.Zero;
   }
  }
 }

}
```

A.2.6. NotGate.java

```
package components;
```

```
/**
 * Diese Klasse repraesentiert das Nicht-Gatter (NOT)
 * @author Nils Rosemann
 *
 */
public class NotGate extends Gate {
public NotGate() {
super(2);
values[0][0] = Complex.Zero;
values[0][1] = Complex.One;
values[1][0] = Complex.One;
values[1][1] = Complex.Zero;
}
}
```

A.2.7. CNotGate.java

```
package components;

/**
 * Diese Klasse repraesentiert das controlled-Not-Gatter
 * @author Nils Rosemann
 *
 */
public class CNotGate extends ControlledGate {
 public CNotGate(int interiors) {
  super(interiors, new NotGate());
 }
}
```

A.2.8. RkGate.java

```
package components;

/**
 * Diese Klasse stellt die controlled-R_k-Gatter
 * fuer die Quanten-Fourier-Transformation
 * zu Verfuegung.
 * @author Nils Rosemann
 *
 */
```

```
public class RkGate extends ReverseControlledGate {
 public RkGate(int k) {
  Complex[][] val = new Complex[2][2];
  val[0][0]=Complex.One;
  val[0][1]=Complex.Zero;
  val[1][0]=Complex.Zero;
  val[1][1]=Complex.exp2pii(1.0/Math.pow(2,k));
  this.set(k-2, new Gate(val));
 }
}
```

A.2.9. QkGate.java

```
package components;

/**
 * Diese Klasse stellt die controlled-R_k-Gatter
 * fuer die Quanten-Fourier-Transformation
 * zu Verfuegung.
 * @author Nils Rosemann
 *
 */
public class RkGate extends ReverseControlledGate {
 public RkGate(int k) {
  Complex[][] val = new Complex[2][2];
  val[0][0]=Complex.One;
  val[0][1]=Complex.Zero;
  val[1][0]=Complex.Zero;
  val[1][1]=Complex.exp2pii(1.0/Math.pow(2,k));
  this.set(k-2, new Gate(val));
 }
}
```

A.2.10. SwapGate.java

```
package components;

/**
 * Diese Klasse repraesentiert das Swap-Gatter
 * @author Nils Rosemann
 *
 */
```

```
public class SwapGate extends Gate {
 public SwapGate ( int qubitNumber ) {
  super((int)Math.pow(2,qubitNumber));
  for(int i=0;i<getDimension();i++) {
   for(int j=0;j<getDimension();j++) {
    values[i][j]=Complex.Zero;
   }
  }
  char[] binary = new char[qubitNumber];
  for(int i=0;i<getDimension();i++) {
   char speicher;
   char[] helper = (Integer.toBinaryString(i)).toCharArray();
   for(int j = qubitNumber-helper.length; j<qubitNumber; j++) {
    binary[j]=helper[j-qubitNumber+helper.length];
   }
   for(int j = 0; j<qubitNumber-helper.length; j++) {
    binary[j]='0';
   }
   speicher=binary[0];
   binary[0]=binary[binary.length-1];
   binary[binary.length-1]=speicher;
   values[i][Integer.parseInt(new String(binary), 2)]=Complex.One;
  }
 }
}
```

A.3. Schaltkreise

A.3.1. FourierTransformCircuit.java

```
package components;

/**
 * Diese Klasse repraesentiert die Quanten-Fourier-Transformation
 * @author Nils Rosemann
 *
 */
public class FourierTransformCircuit implements Circuit {
/**
 * interne Darstellung als Gatter
 */
protected Gate gate;
```

```
/**
 * Erschafft den QFT-Schaltkreis auf qubitNumber Qubits.
 * @param qubitNumber Anzahl der Qubits
 */
public FourierTransformCircuit(int qubitNumber) {
 Gate[] rkgates = new Gate[qubitNumber];
 Gate[] identities = new Gate[qubitNumber];
 for(int i = 1; i<qubitNumber; i++) {
  rkgates[i] = new RkGate(i+1); // R_k-Gatter erschaffen
 }
 rkgates[0] = new HadamardGate(); // Hadamard-Gatter erschaffen
 for(int i = 0; i<qubitNumber; i++) {
  // Identitaeten erschaffen
 identities[i] = new IdentityGate((int)Math.pow(2,i));
 }
 // Schaltkreis zusammenbauen
 this.gate = identities[qubitNumber-1].tensor(identities[1]);
 for(int i = 0; i<qubitNumber; i++) {
  for(int j = 0; j<qubitNumber-i; j++) {
   Gate helper = identities[0];
   helper = helper.tensor(identities[i]);
   helper = helper.tensor(rkgates[j]);
   helper = helper.tensor(identities[qubitNumber-j-i-1]);
   gate = gate.matrix(helper);
  }
 }
 for(int i=0; i<qubitNumber/2; i++) {
  Gate helper = identities[0];
  helper=helper.tensor(identities[i]);
  helper=helper.tensor(new SwapGate(qubitNumber-2*i));
  helper=helper.tensor(identities[i]);
  this.gate = gate.matrix(helper);
 }
}
/**
 * Fouriertransformation auf Register loslassen
 *
 */
public void operateOn(Register register) {
 this.gate.operateOn(register);
}
```

```
/**
 * Gatter fuer weitere Verwendung auslesen.
 */
public Gate getGate() {
 return this.gate;
}
/**
 * Darstellung
 *
 */
public void show() {
 this.gate.show();
}
}
```

A.3.2. InverseFourierTransformCircuit.java

```
package components;

/**
 * Diese Klasse repraesentiert die inverse
 * Quanten-Fourier-Transformation
 * @author Nils Rosemann
 *
 */
public class InverseFourierTransformCircuit implements Circuit {
/**
 * interne Darstellung als Gatter
 */
protected Gate gate;
/**
 * Erschafft den inversen QFT-Schaltkreis auf qubitNumber Qubits.
 * @param qubitNumber Anzahl der Qubits
 */
public InverseFourierTransformCircuit(int qubitNumber) {
 Gate[] qkgates = new Gate[qubitNumber];
 Gate[] identities = new Gate[qubitNumber];
 for(int i = 1; i<qubitNumber; i++) {
 // R_k-Gatter erschaffen
 qkgates[i] = new QkGate(i+1);
 }
 // Hadamard-Gatter erschaffen
```

```
 qkgates[0] = new HadamardGate();
 for(int i = 0; i<qubitNumber; i++) {
  // Identitaeten erschaffen
  identities[i] = new IdentityGate((int)Math.pow(2,i));
 }
 // Schaltkreis zusammenbauen
 this.gate = identities[qubitNumber-1].tensor(identities[1]);
 for(int i = 0; i<qubitNumber; i++) {
  for(int j = 0; j<qubitNumber-i; j++) {
   Gate helper = identities[0];
   helper = helper.tensor(identities[i]);
   helper = helper.tensor(qkgates[j]);
   helper = helper.tensor(identities[qubitNumber-j-i-1]);
   gate = gate.matrix(helper);
  }
 }
 for(int i=0; i<qubitNumber/2; i++) {
  Gate helper = identities[0];
  helper=helper.tensor(identities[i]);
  helper=helper.tensor(new SwapGate(qubitNumber-2*i));
  helper=helper.tensor(identities[i]);
  this.gate = gate.matrix(helper);
 }
}
/**
 * Fouriertransformation auf Register loslassen
 *
 */
public void operateOn(Register register) {
 this.gate.operateOn(register);
}

/**
 * Gatter fuer weitere Verwendung auslesen.
 */
public Gate getGate() {
 return this.gate;
}
/**
 * Darstellung
 *
 */
```

```
public void show() {
 this.gate.show();
}
}
```

A.4. Bruch-Klassen

A.4.1. Fraction.java

```
/*
  File: Fraction.java

  Originally written by Doug Lea and released into the public domain.
  This may be used for any purposes whatsoever without acknowledgment.
  Thanks for the assistance and support of Sun Microsystems Labs,
  and everyone contributing, testing, and using this code.

  History:
  Date       Who                What
  7Jul1998  dl               Create public version
  11Oct1999 dl               add hashCode
*/

package fractions;

/**
 * An immutable class representing fractions as pairs of longs.
 * Fractions are always maintained in reduced form.
 **/
public class Fraction implements Cloneable,
     Comparable, java.io.Serializable {
  protected final long numerator_;
  protected final long denominator_;

  /** Return the numerator **/
  public final long numerator() { return numerator_; }

  /** Return the denominator **/
  public final long denominator() { return denominator_; }

  /** Create a Fraction equal in value to num / den **/
  public Fraction(long num, long den) {
```

```
// normalize while constructing
boolean numNonnegative = (num >= 0);
boolean denNonnegative = (den >= 0);
long a = numNonnegative? num : -num;
long b = denNonnegative? den : -den;
long g = gcd(a, b);
numerator_ = (numNonnegative == denNonnegative)? (a / g) : (-a / g);
denominator_ = b / g;
  }

  /** Create a fraction with the same value as Fraction f **/
  public Fraction(Fraction f) {
numerator_ = f.numerator();
denominator_ = f.denominator();
  }

  public String toString() {
if (denominator() == 1)
  return "" + numerator();
else
return numerator() + "/" + denominator();
}

public Object clone() { return new Fraction(this); }

/** Return the value of the Fraction as a double **/
public double asDouble() {
  return ((double)(numerator())) / ((double)(denominator()));
}

/**
 * Compute the nonnegative greatest common divisor of a and b.
 * (This is needed for normalizing Fractions, but can be
 * useful on its own.)
 **/
public static long gcd(long a, long b) {
  long x;
  long y;

  if (a < 0) a = -a;
  if (b < 0) b = -b;
```

```
  if (a >= b) { x = a; y = b; }
  else        { x = b; y = a; }

  while (y != 0) {
long t = x % y;
x = y;
y = t;
  }
  return x;
}

/** return a Fraction representing the negated value of this Fraction **/
public Fraction negative() {
  long an = numerator();
  long ad = denominator();
  return new Fraction(-an, ad);
}

/** return a Fraction representing 1 / this Fraction **/
public Fraction inverse() {
  long an = numerator();
  long ad = denominator();
  return new Fraction(ad, an);
}


/** return a Fraction representing this Fraction plus b **/
public Fraction plus(Fraction b) {
  long an = numerator();
  long ad = denominator();
  long bn = b.numerator();
  long bd = b.denominator();
   return new Fraction(an*bd+bn*ad, ad*bd);
 }

 /** return a Fraction representing this Fraction plus n **/
 public Fraction plus(long n) {
   long an = numerator();
   long ad = denominator();
   long bn = n;
   long bd = 1;
   return new Fraction(an*bd+bn*ad, ad*bd);
```

```
}

/** return a Fraction representing this Fraction minus b **/
public Fraction minus(Fraction b) {
  long an = numerator();
  long ad = denominator();
  long bn = b.numerator();
  long bd = b.denominator();
  return new Fraction(an*bd-bn*ad, ad*bd);
}

/** return a Fraction representing this Fraction minus n **/
public Fraction minus(long n) {
  long an = numerator();
  long ad = denominator();
  long bn = n;
  long bd = 1;
  return new Fraction(an*bd-bn*ad, ad*bd);
}


/** return a Fraction representing this Fraction times b **/
public Fraction times(Fraction b) {
  long an = numerator();
  long ad = denominator();
  long bn = b.numerator();
  long bd = b.denominator();
  return new Fraction(an*bn, ad*bd);
}

/** return a Fraction representing this Fraction times n **/
public Fraction times(long n) {
  long an = numerator();
  long ad = denominator();
  long bn = n;
  long bd = 1;
  return new Fraction(an*bn, ad*bd);
}

/** return a Fraction representing this Fraction divided by b **/
public Fraction dividedBy(Fraction b) {
  long an = numerator();
```

```
  long ad = denominator();
  long bn = b.numerator();
  long bd = b.denominator();
  return new Fraction(an*bd, ad*bn);
}

/** return a Fraction representing this Fraction divided by n **/
public Fraction dividedBy(long n) {
  long an = numerator();
  long ad = denominator();
  long bn = n;
  long bd = 1;
  return new Fraction(an*bd, ad*bn);
}

/** return a number less, equal, or greater than zero
 * reflecting whether this Fraction is less, equal or greater than
 * the value of Fraction other.
 **/
public int compareTo(Object other) {
  Fraction b = (Fraction)(other);
  long an = numerator();
  long ad = denominator();
  long bn = b.numerator();
  long bd = b.denominator();
  long l = an*bd;
  long r = bn*ad;
  return (l < r)? -1 : ((l == r)? 0: 1);
}

/** return a number less, equal, or greater than zero
 * reflecting whether this Fraction is less, equal or greater than n.
 **/

public int compareTo(long n) {
  long an = numerator();
  long ad = denominator();
  long bn = n;
  long bd = 1;
  long l = an*bd;
  long r = bn*ad;
  return (l < r)? -1 : ((l == r)? 0: 1);
```

```
 }

 public boolean equals(Object other) {
   return compareTo((Fraction)other) == 0;
 }

 public boolean equals(long n) {
   return compareTo(n) == 0;
 }

 public int hashCode() {
return (int) (numerator_ ^ denominator_);
  }

}
```

A.4.2. ContinuedFractions.java

```
package fractions;

//-------------------------------------------------
// Converts to and from simple continued fractions
// http://www.merriampark.com/fractions.htm#Continued
//-------------------------------------------------

import java.util.Vector;
//import EDU.oswego.cs.dl.util.concurrent.misc.Fraction;

public abstract class ContinuedFractions {
public static Vector to (Fraction f) {
Vector v = new Vector ();
if (f.denominator () == 1) {
v.addElement (new Long (f.numerator ()));
}
else if (f.numerator () == 1) {
v.addElement (new Long (0));
v.addElement (new Long (f.denominator ()));
}
else {
Fraction x = new Fraction (f);
long quotient, remainder;
while (x.denominator () > 1) {
```

```
quotient = x.numerator () / x.denominator ();
remainder = x.numerator () % x.denominator ();
v.addElement (new Long (quotient));
x = new Fraction (x.denominator (), remainder);
if (x.denominator () == 1) {
v.addElement (new Long (x.numerator ()));
}
}
}
return v;
}

public static Fraction from (Vector v) {
if (v.size () == 1) {
return new Fraction (getLong (v, 0), 1);
}
Fraction sum = null;
Fraction f;
long a;
for (int i = v.size () - 1; i >= 1; i--) {
a = getLong (v, i - 1);
if (sum == null) {
 Fraction f1 = new Fraction (a, 1);
 Fraction f2 = new Fraction (1, getLong (v, i));
 sum = f1.plus (f2);
}
else {
sum = sum.inverse ().plus (a);
}
}
return sum;
}

 //-----------------------------------------------
 //  // Utility function to extract long from a Vector
 //    //-----------------------------------------------

 private static long getLong (Vector v, int idx) {
 Long longObj = (Long) v.elementAt (idx);
 return longObj.longValue ();
 }
 public static void main (String[] args) {
```

```
 Fraction fractions[] = {new Fraction (45, 16),
 new Fraction (1, 3),
 new Fraction (2, 1),
 new Fraction (3, 5),
 new Fraction (8, 16)};
 Vector v;
 Fraction f;
 for (int i = 0; i < fractions.length; i++) {
 f = fractions[i];
 v = ContinuedFractions.to (f);
 System.out.println (v.toString ());
 f = ContinuedFractions.from (v);
 System.out.println (f.toString ());
 }
 }
 }
```

A.5. Shor-Algorithmus

A.5.1. ShorAlgorithm.java

```
package shor;
import java.util.Vector;
import components.*;
import fractions.*;

/**
 * Diese Klasse faktorisiert kleine natuerliche Zahlen
 * nach dem Shor-Algorithmus
 * @author Nils Rosemann
*/

public class ShorAlgorithm {
    /*
     * Gatter fuer die Vorbereitungsarbeit
     */
    protected Gate prepare;
    /**
     * Gatter fuer die inverse FourierTransformation
     */
    protected Gate ift;
    /**
```

```
 * Anzahl der Qubits
 */
protected int qubitNumber;
/**
 * Erschafft ein ShorAlgorithm-Objekt fuer qubitNumber Qubits.
 * Dabei werden die konstanten Schaltkreiselemente (Vorbereitung
 * und inverse FourierTransformation) schon konstruiert.
 *
 * @param qubitNumber Anzahl der Qubits
 */
  public ShorAlgorithm(int qubitNumber) {
  this.qubitNumber = qubitNumber;
  IdentityGate id = new IdentityGate((int)Math.pow(2, qubitNumber/2));
  this.prepare = HadamardGate.multipleH(qubitNumber/2).tensor(id);
  this.ift = (new InverseFourierTransformCircuit(
  qubitNumber/2)).getGate().tensor(id);
}
/**
 * Versucht, die Zahl N zu faktorisieren.
 *
 * @param N zu faktorisierende Zahl.
 * Muss durch qubitNumber/2 Bits darstellbar sein!
 * @return nicht-trivialer Faktor von N
 */
public long factorize(int N) {
  if(N%2==0)return 2;      // gerade Zahlen aussortieren
  if(isPrimeOrPower(N)){   // Primzahlen und Potenzen aussortieren
    System.err.println("Primzahl oder Potenz!");
    System.exit(0);
  }
  int measure = 0,x=0;
  long faktor=2;
  // solange noch kein Faktor gefunden, weitermachen
  while(N%faktor!=0){
    // solange noch nicht sinnvolles gemessen, weitermachen
    while(measure==0) {
      x = (int)(N*Math.random()+1); // Zufallszahl
      if(ggT(x, N)>1){ // Glueckstreffer!
        System.out.println(
           "Zufallszahl hat schon gemeinsamen Faktor!");
        return ggT(x, N);
      }
```

```
        // Register vorbereiten
        Register reg = new Register(qubitNumber);
        // Mod. Exponentiation
        ModGate mod = new ModGate(x, N, qubitNumber);
        System.out.println("Gatter bereit");

        prepare.operateOn(reg); // Gatter nach einander anwenden
        mod.operateOn(reg);
        ift.operateOn(reg);

        System.out.println("Gatter angewendet");

        measure = reg.measure()/(
               (int)Math.pow(2, qubitNumber/2)); // Messung

        System.out.println(
               "Zufallszahl: " + x + " Messung: " + measure);
      }
      // Kettenbruchentwicklung
      Vector vec = ContinuedFractions.to(new Fraction(
      measure, (long)Math.pow(2, qubitNumber/2)));

      long r = ((Long)vec.lastElement()).longValue();
      System.out.println("Kettenbruchentwicklung ergibt r="+ r);


      long test;
      test = ggT((long)Math.pow(x, r/2)+1,N);
      if(check(test,N))faktor=test;
      test = ggT((long)Math.pow(x, r/2)-1,N);
      if(check(test,N))faktor=test;
      test = ggT((long)Math.pow(x, r)+1,N);
      if(check(test,N))faktor=test;
      test = ggT((long)Math.pow(x, r)-1,N);
      if(check(test,N))faktor=test;
      measure = 0;
    }

    return faktor; // Faktor
  }

  /**
```

```
 * Euklidischer Algorithmus
 * @param x erste Zahl
 * @param y zweite Zahl
 * @return gemeinsamer Teiler
 */
public static long ggT(long x, long y) {
  if (y == 0)
    return x;
  else
    return ggT(y, x % y);
  }

/**
 * naiver Primzahlen- und Potenztest
 * @param x zu testende Zahl
 * @return true, wenn Zahl Primzahl oder Potenz, sonst
 */
public static boolean isPrimeOrPower(long x) {
  for(int i=2; i<=(int)Math.sqrt(x); i++) {
    if(x%i==0)return false;
    long pot = i;
    while(pot<=x) {
      if(x==pot)return true;
      pot*=i;
    }
  }
  return true;
}
/**
 * Testet, ob x ein nicht-trivialer Faktor von N ist.
 */
public boolean check(long x, int N) {
  return (x!=1 && x!=N && N%x==0);
}
/**
 * Hauptprogramm. Kann mit null, einem oder zwei Argumenten
 * genutzt werden.
 * Null: Die Zahl 15 wird mit 8 Qubits (minimale Anzahl) faktorisiert.
 * Eins: Das Argument wird mit 8 Qubits faktorisiert.
 * Zwei: Das erste Argument wird mit so vielen Qubits faktorisiert,
 * wie das zweite Argument angibt.
 * @param args Argumente (null, eins oder zwei Argumente)
```

```
     */
    public static void main(String[] args) {
      int input, qubits;
      // Anzahl Argumente auswerten
      if(args.length>1)qubits = Integer.parseInt(args[1]);
        else qubits = 8;
      if(args.length>0)input = Integer.parseInt(args[0]);
        else input = 15;

      // Shor-Algorithmus mit Qubit-Anzahl initialisieren
      ShorAlgorithm shor = new ShorAlgorithm(qubits);
      // Faktorisierungsmethode aufrufen.
      long a = shor.factorize(input);
      // Faktor gefunden?
      if(a!=0)System.out.println(
           "Faktorisierung: "+input+" = "+a+" * "+input/a);

        else System.out.println("Faktorisierung fehlgeschlagen!");
      }
}
```

Literatur

[BBBV97] C. H. Bennett, E. Bernstein, G. Brassard, and U. V. Vazirani. Strengths and weaknesses of quantum computing. *SIAM J. Comput.*, 26(5):1510–1523, 1997. *arXive e-print quant-ph/9701001.*

[BBC+95] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. H. Margolus, P. W. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter. Elementary gates for quantum computation. *Phys. Rev. A*, 52(5):3457–3467, 1995. *arXive e-print quant-ph/9503016.*

[BCDP96] D. Beckman, A. N. Chari, Sr. Devabhaktuni, and J. Preskill. Efficient networks for quantum factoring. *Phys. Rev. A*, 54(2):1034–1063, 1996. *arXive e-print quant-ph/9602016.*

[BMP+99] P. O. Boykin, T. Mor, M. Pulver, V. Roychowdhury, and F. Vatan. On universal and fault-tolerant quantum computing. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 486–494, 1999. *arXive e-print quant-ph/9906054.*

[CFH97] D. Cory, A. Fahmy, and T. Havel. Ensemble quantum computing by nuclear magnetic resonance spectroscopy. In *Proc. Nat. Acad. Sci. USA*, volume 94, pages 1634–1639. National Academy of Science USA, 1997.

[CZ95] J. I. Cirac and P. Zoller. Quantum computations with cold trapped ions. *Phys. Rev. Lett.*, 74(20):4091–4094, 1995.

[Deu85] D. Deutsch. Quantum theory, the church-turing principle and the universal quantum computer. *Proc. R. Soc. Lond. A*, 400:97–117, 1985.

[Deu89] D. Deutsch. Quantum computational networks. *Proc. R. Soc. Lond. A*, 425:73–90, 1989.

[Die82] D. Dieks. Communication by epr devices. *Phys. Lett. A*, 92(6):271–272, 1982.

[Div95] D. P. Divincenzo. Quantum computation. *Science*, 270:255–261, 1995.

[Fey82] R. P. Feynman. Simulating physics with computers. *Int. J. Theor. Phys.*, 21(6&7):467–488, 1982.

[Fey85] R. P. Feynman. Quantum mechanical computers. *Optics News*, 11:11–20, 1985.

[Gro97] L. K. Grover. Quantum mechanics helps in searching for a needle in a haystack. *Phys. Rev. Lett.*, 79(2):325–328, 1997. *arXive e-print quant-ph/9706033.*

[HW60] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers, Fourth Edition.* Oxford University Press, London, 1960.

[KMW02] D. Kielpinski, C. R. Monroe, and D. J. Wineland. Architecture for a large-scale ion-trap quantum computer. *Nature*, 417:709–711, 2002.

[LD98] D. Loss and D. P. DiVincenzo. Quantum computation with quantum dots. *Phys. Rev. A*, 57(1):120–126, 1998. *arXive e-print cond-mat/9701055.*

[NC00] M. A. Nielsen and I. L. Chuang. *Quantum computation and quantum information.* Cambridge University Press, Cambridge, 2000.

[NPT99] Y. Nakamura, Yu. A. Pashkin, and J. S. Tsai. Coherent control of macroscopic quantum states in a single-cooper-pair box. *Nature*, 398(6730):786–788, 1999.

[Pit99] O. Pittenger. *An Introduction to Quantum Computing Algorithms.* Birkhäuser Verlag, Boston, 1999.

[Sch95] B. Schumacher. Quantum coding. *Phys. Rev. A*, 51:2738–2747, 1995.

[Sho94] P. W. Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings of the* 35^{th} *Annual Symposium on Foundations of Computer Science*, pages 124–134. Institute of Electrical and Electronic Engineers Computer Society Press, 1994.

[Sim94] D. R. Simon. On the power of quantum computation. In *Proceedings of the* 35^{th} *Annual Symposium on Foundations of Computer Science*, pages 116–123, Los Alamitos, CA, 1994. Institute of Electrical and Electronic Engineers Computer Society Press.

[Sim97] D. R. Simon. On the power of quantum computation. *SIAM J. Comput.*, 26(5):1474–1483, 1997.

[Vin95] D. P. Di Vincenzo. Two-bit gates are universal for quantum computation. *Phys. Rev. A*, 51(2):1015–1022, 1995. *arXive e-print cond-mat/9407022.*

[VSB+01] L. M. K. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang. Experimental realization of shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, 414:883–887, 2001. *arXive e-print quant-ph/0112176.*

Danksagung

Diese Bachelor-Arbeit entstand in der Arbeitsgruppe "Makroskopische Systeme und Quantentheorie" des Fachbereichs Physik an der Universität Osnabrück.
Mein besonderer Dank gilt Herrn Prof. Dr. Heinz-Jürgen Schmidt für die Möglichkeit, ein Thema über Quantencomputer zu bearbeiten. Bei drei Vorlesungen und einem Treffen pro Woche kann man mit ruhigem Gewissen von sehr intensiver (und sehr guter) Betreuung sprechen: Vielen Dank dafür!
Ich danke meinem "Mitstreiter" Uwe Sander für vier Monate erfolgreicher und vor allem humorvoller Zusammenarbeit.
Auch danke ich meinem Zimmergenossen Mirko Brüger, der in Sachen Tee, Theorie und Text immer zur Verfügung stand.
Ein ganz besonderer Dank gilt meinen Eltern, die immer hinter mir stehen.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Bachelor-Arbeit selbstständig verfasst, keine anderen als die angegebenen Hilfmittel verwendet und zuvor noch keine Bachelor-Prüfung der Fachrichtung "Physik" abgelegt habe. Ersteres gilt nicht für Kapitel 2 "Quantencomputer", welches Uwe Sander und ich zusammen ohne fremde Hilfe erstellt haben.

Osnabrück, den 12. Juli 2004

Nils Rosemann