Qubits als Bestandteile von Quantencomputern

Physik und Informatik

Philip Geißler, Alexander von Mach, Joe Schaller

Carl-Zeiss Gymnasium Jena

Außenbetreuer: Dr. Stephan Fritzsche
Fachlehrerin: Kerstin Hertenberger
Seminarfachlehrerin: Dr. Ilona Böttcher

17. September 2017

Inhaltsverzeichnis

Kapitel 1

Einführung Quantencomputer

Die Welt befindet sich im digitalem Zeitalter. Immer und überall hat man Zugriff auf Internet. In jeder Maschine befinden sich Mikrochips oder andere elektronische Rechenwerke. Nach dem Moore'schen Gesetz verdoppelt sich die Anzahl an Transistoren auf einem Chip alle ein bis zwei Jahre. Bisher traf diese Gesetzmäßigkeit immer zu, doch bald wird man bei der Herstellung von Transistoren an die Grenzen des Machbaren stoßen, so dass die Rechenleistung eines Chips bei gleicher Größe nicht weiter erhöht werden kann. Da jedoch die Digitalisierung der Welt weiter fortschreitet, müssen Computer leistungsfähiger werden, ohne dass die Größe des Rechners erheblich zunimmt. Die Lösung dieses Problems könnte der sogenannte Quantencomputer sein. Dieser nutzt Gesetze und Phänomene der Quantenphysik um ein komplexes Problem um ein Vielfaches schneller zu berechnen als ein herkömmlicher Computer.
Schwerpunkt der vorliegenden Seminarfacharbeit sind die Funktionsweise des Quantencomputers und das Verhalten der Qubits. Dieses Verhalten soll auf der Grundlage der Quantenphysik erläutert werden. Es wird geklärt, wie Qubits durch Quantengatter beeinflusst und manipuliert werden und wie diese Erkenntnisse zum Bau des Quantencomputers genutzt werden können. Weiterhin werden Nutzungsmöglichkeiten des Quantencomputers und auch die damit einhergehenden Chancen und Gefahren aufgezeigt.
Als Teil der praktischen Arbeit wird eine Simulation eines Quantencomputers erstellt, mit der das Verhalten von einzelnen, aber auch verschränkten Qubits anschaulich dargestellt wird. Durch Quantengatter können die Zustände der Qubits verändert werden. Die Ergebnisse werden dann als Wahrscheinlichkeitsdiagramm ausgegeben. Zum Ende dieser Seminarfacharbeit wird noch ein Ausblick darauf gegeben, wie sich die weitere Forschung entwickeln wird und ob der Quantencomputer bald im Wohnzimmer stehen wird.

Kapitel 2

Qubits

2.1 Was sind Qubits?

Es ist sinnvoll zuerst zu klären, worum es sich bei einem Qubit handelt. Der Qubit oder auch Quanten-Bit ist ein Objekt zur informatischen Datenverarbeitung, welches quantenmechanische Effekte ausnutzt. Diese Effekte treten nur bei kleinsten Teilchen auf und sind messbar. Dazu gehören zum Beispiel Ionen, Photonen oder Elektronen, welche, dank ihrer quantenmechanischen Eigenschaften, auf verschiedenste Arten manipuliert und zur Datenverarbeitung genutzt werden können, . Als erste Möglichkeit zur Umsetzung von Qubits wird Ionenfalle näher betrachtet. Zuerst erzeugt man ein Vakuum in dem ein Magnetfeld erzeugt wird, um dann ein oder mehrere Ionen, welche zu meist positiv geladen sind, in diesem Magnetfeld gefangen zu halten und in einer Kette aufzureihen. Nun lässt man dieses System auf wenige Kelvin über dem Absoluten Nullpunkt auskühlen, um sicherzustellen, dass alle Ionen in Ihrem Grundzustand sind. Um nun die Manipulation zu veranschaulichen konzentrieren wir uns auf nur ein Ion, da ansonsten durch Effekte der Verschränkung sehr viel komplexere Probleme auftreten. Da wir nun ein Ion im Grundzustand haben und es manipulieren wollen, haben wir nun die Wahl auf welche Art und Weise wir dies tun. Zum einen könnte man ein 2. Magnetfeld anlegen, welches dann den Spin neu ausrichten würde, Jedoch wäre das in einer Kette mit mehreren Ionen fatal, da man so die verschränkten zustände verliert. Die zweite Möglichkeit ist, das Ion mit Lichtimpulsen anzuregen um es in einen bestimmten Anfangszustand zu bringen. Doch auch bei dieser Varianten, welche sich auch bei mehreren Verschränkten Ionen anwenden lässt gibt es ein Problem, und zwar will das Ion nicht besonders lange an der Energie festhalten und in den Grundzustand zurück kehren. Bei steigender Intensität von Lichtimpulsen nimmt hierbei die Zeit Dramatisch ab, jedoch können so auch die Ionen stärker manipuliert werden, aber auch ein stark manipulierten Ion bringt nicht viel wenn es nach weniger als einer milliardstel Sekunde zurück in den Grundzustand fällt, zudem, dass man in diesem kurzen Zeitraum nicht einmal Gatter auf das neu erschaffene Qubit anwenden kann. Nun wird das Ion nur schwach angeregt, damit es nur leicht vom Grundzustand abweicht, jedoch auch viel Zeit zur Manipulation lässt, da es so mehrere Sekunden bis Minuten in diesem Zustand Bleibt. Um nebenher noch einige Technische Sachen anzumerken, ist zu erwähnen, dass durch solcher Ionenfallen schon weit mehr als 10 Ionen miteinander verschränkt wurden, wodurch einerseits sehr komplexe Superpositionszustände, andererseits

aber auch riesige Matrizen entstehen, welche mit der rechen Leistung eines herkömmlichen Computers kaum zu simulieren sind. Einige der neusten Ionenfallen haben es geschafft 14 Ionen aufzureihen, was eine Matrix zur Folge hat welche die Maße $14^2 * 14^2$ hat. Um nun noch auf zwei andere sehr interessante Methoden einzugehen, ist es wichtig ein Vorwissen über Supraleiter zu haben, da diese sehr essenziell sind. Supraleiter entstehen, wenn man ein elektrisch mehr oder minder gut leitfähigen Stoff nimmt und ihr auf eine bestimmte Temperatur herunter kühlt. Sollte man nun eine kritische Temperatur erreichen wird er supraleitend, was bedeutet, dass der elektrische Widerstand dieses Leiters null entspricht. Somit legt man den Baustein für sehr leistungsfähige Elektromagneten, welche in der heutigen Zeit gerne in Kernspintomographen verwendet werden. Und schon hierbei liegt der große Nutzen. Mit solch einem Elektromagneten kann man den Spin von Atomkernen so ausrichten, dass alle in dieselbe Richtung Zeigen. Leider hat dieses Problem einige essenzielle Probleme. Einerseits ist es nicht skalierbar, da es sehr hart ist riesige Supraleitende Elektromagnete zu verkleinern so, dass sie vielleicht eines Tages in eine Tasche passen könnten. Zudem ist das Problem, dass wir bei der Kernspinresonanz nicht nur einen Atomkern ausrichten sondern alle, welche sich im starken Bereich des Feldes befinden. Des Weiteren wird an dieser Technologie zum heutigen Zeitpunkt nicht mehr intensiv geforscht. Die nächste Möglichkeit auf welche ich eingehen wollte ist SQUIDs oder auch supraleitende Quanteninterferenzeinheit. Hierbei haben wir 2 Supraleitende Teile in der Form eines Halbkreises, jedes Teil mit einem Anschluss. Zwischen diesen Supraleiten befindet sich eine Trennschicht, welche als Isolierschicht dient und extrem dünn sein muss, da durch die extreme Kälte erzeugte Cooper-Paare durch diese hindurch wandern können müssen. Nun kann man beim Anlegen eines Weiteren Magnetfeldes die Ströme so beeinflussen, dass es zu einem messbaren Spannungsabfall über dem SQUID kommt, welche wiederum eine Periode verursacht. Diese hat wiederum die gleiche Periode wie das magnetische Flussquantum.

2.2 Grundsätzliches Verhalten von Qubits

2.2.1 Mathematische Grundlagen

Zustände eines einzelnen Qubits

Im rein mathematischen Sinne ist ein Qubit durch ein System mit einem Zustand $|\psi_1\rangle$ als Kombination der Eigenzustände $|0\rangle$ und $|1\rangle$ definiert. Es weist deshalb folgende zwei Eigenschaften auf[1]:

$$|\psi_1\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \;\text{ mit } \alpha, \beta \in \mathbb{C} \tag{2.1}$$

$$\begin{aligned} 1 &= |\alpha|^2 + |\beta|^2 \\ &= \alpha\bar{\alpha} + \beta\bar{\beta} \\ &= (\Re(\alpha) + i\Im(\alpha)) \cdot (\Re(\alpha) - i\Im(\alpha)) + (\Re(\beta) + i\Im(\beta)) \cdot (\Re(\beta) - i\Im(\beta)) \\ &= \Re(\alpha)^2 + \Im(\alpha)^2 + \Re(\beta)^2 + \Im(\beta)^2 \end{aligned} \tag{2.2}$$

[1] Terr, o.J, S.1 [1]

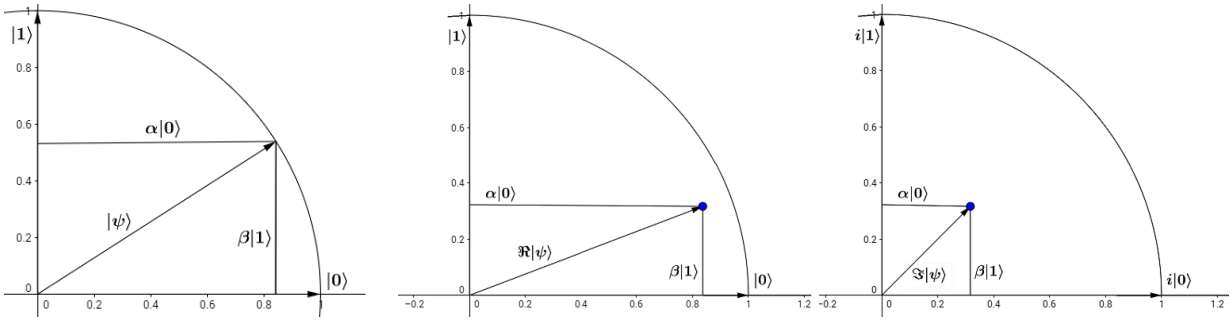
Hierbei geben α und β die sogenannten Amplituden des Ein-Qubit Systems an. Nimmt man von einem der beiden Werte das Betragsquadrat, so erhält man die Wahrscheinlichkeit für das Auftreten des korrespondierenden Grundzustandes. Es besteht eine Eindeutigkeit des Vorganges in Richtung der Wahrscheinlichkeiten, eine Eineindeutigkeit ist jedoch nicht mehr gegeben, denn eine gemessene Wahrscheinlichkeit p für den Grundzustand $|0\rangle$ des Ein-Qubit Systems kann allein im reellen Zahlenbereich vier mögliche Anfangsgesamtzustände besitzen. Für einen komplexen Zahlenbereich erweitert sich die Anzahl der Gesamtzustände ins Unendliche.

$$\begin{aligned} |\alpha|^2 &= p & |\beta|^2 &= 1-p & \alpha,\beta &\in \mathbb{R} \\ \alpha^2 &= p & \beta^2 &= 1-p & & \\ \alpha &= \pm\sqrt{p} & \beta &= \pm\sqrt{1-p} & & \\ \Rightarrow |\psi_1\rangle &= \pm\sqrt{p}\,|0\rangle \pm \sqrt{1-p}\,|1\rangle & & & & \end{aligned}$$

$$\begin{aligned} |\alpha|^2 &= p & |\beta|^2 &= 1-p & \alpha,\beta &\in \mathbb{C} \\ \Re(\alpha)^2 + \Im(\alpha)^2 &= p & \Re(\beta)^2 + \Im(\beta)^2 &= 1-p & & \\ \Rightarrow \alpha &\text{ ist nicht einschränkbar} & \Rightarrow \beta &\text{ ist nicht einschränkbar} & & \end{aligned}$$

Matrixdarstellung des Qubitzustandes

Durch die fehlende Eineindeutigkeit entwickelt sich eine Ungewissheit über den Zustand des Qubits, welchen man nicht durch Messergebnisse ermitteln kann. Man kann ihn jedoch nach einer beliebigen Anzahl von eindeutigen, zustandsverändernden Einwirkungen berechnen. Es bietet sich hierbei an, die Grundzustände als zweidimensionale Vektoren zu betrachten, aus denen sich ein zweidimensional komplexer Zustandsvektor bilden lässt. Will man diesen Vektor darstellen, so benötigt man 4 Koordinatenachsen. Sind α und β jedoch reell, so reicht eine zweidimensionale Darstellung aus. Abbildung 2.1 stellt diesen Sachverhalt für beide Varianten dar.



(a) $|\psi_1\rangle = 0.84\,|0\rangle + 0.55\,|1\rangle$ (b) $|\psi_1\rangle = (0.84 + 0.32i)\,|0\rangle + (0.32 + 0.32i)\,|1\rangle$

Abbildung 2.1: Darstellungsmöglichkeiten abhängig von den Amplituden der Grundvektoren[2]

[2]Zhang et al., Juli 2014 [2]

$$|\mathbb{C}^2| = |\mathbb{R}^4|$$

$$\alpha = a_1 + 0 * i \wedge \beta = a_2 + 0 * i \qquad \Rightarrow |\psi\rangle \in \mathbb{R}^2$$

Falls diese Darstellungsweise genutzt wird, kann man folglich den gesamten Zustand des einzelnen Qubits in eine $2 * 1$ Matrix zusammenfassen, die den gesamten Zustand des Ein-Qubit Systems darstellt.

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \qquad \Rightarrow |\psi_1\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

Erweiterung auf mehrerere Qubits

Da eine Superposition eines Qubits durch die Amplituden aller möglichen Teilzustände dieses Qubits beschrieben wird, sind für eine vollständige Superpositionsbeschreibung dieses Qubits die Amplituden von $|0\rangle$ und $|1\rangle$ nötig. Verallgemeinert auf n Qubits steigt die Anzahl dieser Teilzustände exponentiell, denn n Qubits besitzen 2^n Teilzustände. Somit wird nach der oben genannten Darstellungsweise eine $2^n * 1$ Matrix für das n-Qubit System benötigt. Die erste Zeile der Matrix gibt dabei die Amplitude für den Endzustand $|000\cdots0\rangle$ und die letzte Zeile die Amplitude für den Endzustand $|111\cdots1\rangle$ an. Allgemein kann man den Zustand in der m-ten Zeile durch die binäre Darstellung der Zeilenzahl (m_2) ermitteln.

$$|\psi_1\rangle = \begin{pmatrix} \alpha_1 \\ \beta_1 \end{pmatrix} \qquad |\psi_2\rangle = \begin{pmatrix} \alpha_1 * \alpha_2 \\ \alpha_1 * \beta_2 \\ \beta_1 * \alpha_2 \\ \beta_1 * \beta_2 \end{pmatrix} \qquad |\psi_3\rangle = \begin{pmatrix} \alpha_1 * \alpha_2 * \alpha_3 \\ \alpha_1 * \alpha_2 * \beta_3 \\ \alpha_1 * \beta_2 * \alpha_3 \\ \alpha_1 * \beta_2 * \beta_3 \\ \vdots \end{pmatrix} \qquad \cdots$$

Folglich nimmt auch die Zahl der darzustellenden Dimensionen für das gesamte n-Qubit System zu, denn es werden durch die 2^n komplexen Dimensionen theoretisch $2 * 2^n = 2^{(n+1)}$ Koordinatenachsen benötigt. Eine unrealistische Anzahl für alle $n > 1$, denn ab $n = 2$ bedarf es mindestens 8 Koordinatenachsen.

Kroneckermultiplikation

Um eine Veränderung an einer Quantensuperposition vorzunehmen, benötigt man bei dieser Darstellungsweise eine Matrix, welche dieselbe Breite aufweist wie die Höhe der Quantensuperposition. Sowohl für die Berechnung des anfänglichen Quantensuperpositionszustandes aus den einzelnen Qubits als auch für die Berechnung eines Gesamtgatters aus den einzelnen

später beschriebenen Gattern nutzt man das Kroneckerprodukt[3]. Es ist ein besonderes Produkt zweier beliebiger Matrizen A und B und und enthält alle Permutationen von Produkten der Einträge der multiplizierten Matrizen. Das Kroneckerprodukt ist jedoch nicht kommutativ, weswegen die Gatter in derselben Weise wie die einzelnen Qubits kronecker-multiplizert werden müssen, um ein valides Ergebnis erhalten zu können.

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} A_{a1} \cdot B & A_{a2} \cdot B & \cdots \\ A_{b1} \cdot B & A_{b2} \cdot B & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} = \begin{bmatrix} A_{a1} \cdot B_{a1} & A_{a1} \cdot B_{a2} & \cdots & A_{a2} \cdot B_{a1} & A_{a2} \cdot B_{a2} & \cdots & \cdots \\ A_{a1} \cdot B_{b1} & A_{a1} \cdot B_{b2} & \cdots & A_{a2} \cdot B_{b1} & A_{a2} \cdot B_{b2} & \cdots & \cdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \cdots \\ A_{b1} \cdot B_{a1} & A_{b1} \cdot B_{a2} & \cdots & A_{b2} \cdot B_{a1} & A_{b2} \cdot B_{a2} & \cdots & \cdots \\ A_{b1} \cdot B_{b1} & A_{b1} \cdot B_{b2} & \cdots & A_{b2} \cdot B_{b1} & A_{b2} \cdot B_{b2} & \cdots & \cdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

$$|\psi_n\rangle = q_1 \otimes (q_2 \otimes (\cdots \otimes (q_{n-1} \otimes q_n)))$$
$$Gatter_n = G_{t(1)} \otimes (G_{t(2)} \otimes (\cdots \otimes (G_{t(n-1)} \otimes G_{t(n)})))$$

Matrixmultiplikation

Will man aus den durch Kroneckermultiplikation berechneten Gesamtgattern und der Anfangssuperposition die Ausgabe des Quantencomputers berechnen, dann muss man Matrixmultiplikation anwenden können[4]. Für eine Matrixmultiplikation werden zwei Ausgangsmatrizen mit der Eigenschaft benötigt, dass die Zeilenanzahl der zweiten Matrix gleich der Spaltenanzahl der ersten Matrix ist. Als Ergebnis der Matrixmultiplikation erhält man eine Matrix mit der Zeilenanzahl der ersten und der Spaltenanzahl der zweiten Ausgangsmatrix.

$$\mathbf{A}_{l*m}\mathbf{B}_{m*n} = \mathbf{Result}_{l*n}$$

Ist das Ziel die Berechnung des Endsuperpositionszustandes des Qubitsystems als Folge von zeitlich nacheinander ablaufenden Einwirkungen der einzelnen Gesamtgatter auf den Qubitsystem-Anfangszustand, so muss der Anfangszustand in zeitlich geordneter Reihenfolge mit allen Gesamtgattern multipliziert werden, um den Endsuperpositionszustand zu erhalten.

$$|\psi_{n:res}\rangle = \cdots (Gatter_3 \cdot (Gatter_2 \cdot (Gatter_1 \cdot |\psi_{n:0}\rangle)))$$

Die gestellten Bedingungen werden dabei eingehalten, da die Qubitsuperpositionszustände $2^n * 1$ und die Gatter $2^n * 2^n$ als Größe aufweisen.

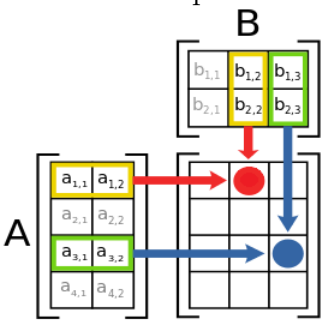
[3]Cappellaro, 2012, S.39 [3]
[4]Institute of Physics, Slovak Academy of Sciences, 2016, S.1 [4]

Allgemein

$$
\begin{aligned}
\mathbf{AB}_{l*n} &= \mathbf{A}_{l*m}\mathbf{B}_{m*n} \\
&= \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ a_{l1} & \cdots & \cdots & a_{lm} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ b_{m1} & \cdots & \cdots & b_{mn} \end{bmatrix} \\
&= \begin{bmatrix} \sum_{k=1}^{m} a_{1k}b_{k1} & \sum_{k=1}^{m} a_{1k}b_{k2} & \cdots & \sum_{k=1}^{m} a_{1k}b_{kn} \\ \sum_{k=1}^{m} a_{2k}b_{k1} & \sum_{k=1}^{m} a_{2k}b_{k2} & \ddots & \vdots \\ \vdots & \ddots & \ddots & \vdots \\ \sum_{k=1}^{m} a_{lk}b_{k1} & \cdots & \cdots & \sum_{k=1}^{m} a_{lk}b_{kn} \end{bmatrix}
\end{aligned}
$$

(a) Mathematische Umsetzung

Beispiel



Rot : $a_{1,1} \cdot b_{1,2} + a_{1,2} \cdot b_{2,2}$
Blau: $a_{3,1} \cdot b_{1,3} + a_{3,2} \cdot b_{2,3}$

(b) Vorgehensverdeutlichung[5]

In der mathematischen Darstellung erschließt sich daraus ein Problem. Es werden große Matrizen erstellt, welche dann aufwendig mit einem Qubitzustand multipliziert werden müssen. Durch das exponentielle Komplexitätswachstum ist die Berechnung des Endsuperpositionszustandes des Qubitsystems daher schon bei moderaten Systemgrößen selbst mit Computerrechenassistenz nahezu unmöglich. Deswegen werden Quantencomputer entwickelt, die sich die Eigenschaften von Teilchen zu Nutze machen, die in der Lage sind, diese Veränderungen physisch ausführen zu können.[6]

Verschränkte Qubits

Eine der komplexesten und hervorstechendsten Eigenschaften von n-Qubit Systemen ist die Verschränkbarkeit der Qubits des Systems[7]. Darunter versteht man die Möglichkeit, einen Endsuperpositionszustand zu erzeugen, welcher nicht allein durch theoretische Amplituden der korrespondierenden Grundzustände der Qubits beschrieben werden kann. Dieser Zustand hat deshalb die Eigenschaft, aus keinem Kroneckerprodukt n spezifischer $|\psi_1\rangle$ Matrizen berechnet werden zu können.

$$|\psi_n\rangle \neq q_1 \otimes (q_2 \otimes (\cdots \otimes (q_{n-1} \otimes q_n)))$$

Somit erklärt die Verschränkung das exponentielle Datenwachstum der Superpositionszustände eines n-Qubit Systems. Denn wenn nicht alle Zustände eines Systems durch die einzelnen Teilsysteme beschrieben werden können, dann müssen Daten über alle möglichen Permutationszustände der einzelnen Qubitamplituden des Systems gespeichert werden können. Dies entspricht exakt der Wirkung des Kroneckerproduktes von n Qubits.

[5] User:Bilou, 2010 [5]
[6] siehe Kapitel 3.2.1: Funktionsweise von Quantencomputern
[7] Wengenmayr, Juli 2012, S.2 [6]

$$\underbrace{2\left\{\begin{array}{ccccc}\begin{pmatrix}\alpha_1\\ \beta_1\end{pmatrix} & \cdot & \begin{pmatrix}\alpha_2\\ \beta_2\end{pmatrix} & \cdots & \begin{pmatrix}\alpha_n\\ \beta_n\end{pmatrix}\\ \downarrow & & \downarrow & \cdots & \downarrow \\ — & & — & \cdots & —\end{array}\right.}_{\mathrm{n}} = \begin{pmatrix}\alpha_1\\ \beta_1\end{pmatrix} \otimes \left(\begin{pmatrix}\alpha_2\\ \beta_2\end{pmatrix} \otimes \left(\cdots \otimes \begin{pmatrix}\alpha_n\\ \beta_n\end{pmatrix}\right)\right) \triangleq 2^n$$

2.2.2 Physikalische Grundlagen

Welle Teilchen Dualismus

Um nun von der mathematischen Theorie in die Physik zurück zu kommen, muss zuvor einiges gesagt werden, so zum Beispiel, dass Quantensysteme an sich nicht wie herkömmliche Objekte der Physik agieren. Daher unterscheidet man heutzutage in der Physik in den Makro- und Mikrokosmus, wobei Makrokosmus die Welt wie wir sie kennen beschreibt und Mikrokosmus die Welt kleinster Teilchen, was alles von Lichtteilchen über Elektronen bis hin zu Atomkernen und ganzen Atomen einschließt. Während alle Objekte des Makrokosmus eine bestimmte Position in Zeit und Raum haben, kann dies von den Mikroskopischen Teilchen nicht behauptet werden. So kommen wir zu dem ersten Quantenmechanischen Problem und zwar dem Welle Teilchen Dualismus, welcher besagt, dass alle fundamentalen Partikel sowohl Teilchen als auch Welle sein können. Das bekannteste Beispiel dafür ist das Verhalten von Licht bei dem man diese einerseits als Strahlen, andererseits aber auch Wellen betrachten kann so sieht man beim Beschuss einen Photodetektors mit einzelnen Lichtimpulsen feststellen, das sich nach und nach an verschiedenen Stellen helle punkte ergeben, welche nebenher ein weiteres quantenmechanischen Phänomen aufzeigen welches zu einem späteren Punkt erklärt wird[8].

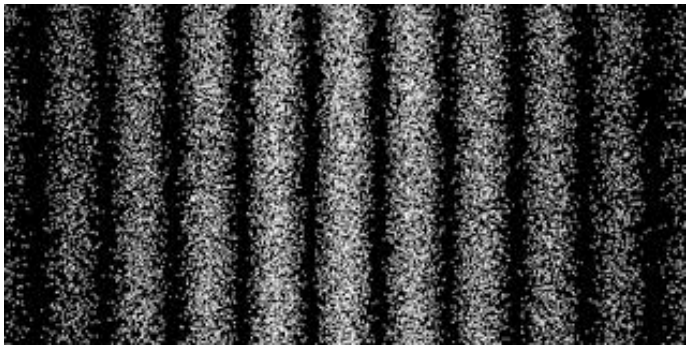


Abbildung 2.3: Ergebnis eines Doppelschlitzexperimentes

[8]Sexl, o. J. [7]

Das Wellenverhalten eines Lichtteilchens hingegen scheint nicht so offensichtlich zu sein, da Es gegensätzlich erscheint das ein Teilchen sich auch wie eine Welle verhält. Jedoch wird durch Versuche, wie dem Doppelspaltversuch klar das sich Licht auch wie eine Welle verhält in dem man Lichtteilchen an stellen beobachten kann an denen laut Strahlentheorie keine sein sollten.

Heisenbergsche Unschärferelation

Und an dieser Stelle kommen wir zu einem weiteren quantenmechanischen Phänomen, welches schon absurder klingt als das eben genannte. Es ist nicht vorherzusehen, an welcher Stelle ein Teilchen zu einem bestimmten späteren Zeitpunkt ist, wenn man weiß wie Momentum und Position des Teilchens zum jetzigen Zeitpunkt sind. Dies Liegt an der Heisenbergschen Unschärferelation, welche im spezifischen besagt, dass man nie zwei gleichwertige Eigenschaften eines Quantenobjekts zur selben Zeit wissen kann, deswegen Unschärfe. So ist es zwar möglich zwei Messungen schnell hintereinander jedoch nicht gleichzeitig zu machen unter der Bedingung, dass unterschiedliche Eigenschaften gemessen werden. Ein bekanntes und zu Anfang genanntes Beispiel dafür ist, das Für ein Teilchen nicht Ort und Impuls zur selben Zeit bekannt sein können. Am besten erkennt man dies anhand von Lichtteilchen. Will man nun den Ort eines Lichtteilchens wissen stellt man sich folgende Frage: "Was weiß ich?". Und stellt dann fest, dass man Die Geschwindigkeit weiß. Dies hat nun jedoch zur Folge, dass man nicht die Position bestimmen kann, was zum Teil auch wieder mit dem Wellenverhalten zu tun hat. So ist also die Position Eines Lichtteilchens immer unbekannt bis zu dem Zeitpunkt an dem die Position gemessen wird und das Teilchen verschwindet. Nun um dies zu testen kann man wieder das Doppelspaltexperiment heranziehen um gleich beide der genannten Theorien zu testen. Wie schon im ersten Experiment erwähnt kann man durch die Welleneigenschaften Lichtteilchen an Stellen beobachten an denen normalerweise keine sein sollten, jedoch kann man nach Beendigung des Experimentes ein eindeutiges Interferenzmuster erkennen, welches nun die 1. Theorie bestätigt, um nun aber die 2. Theorie zu testen muss man einen Schritt zurück gehen und betrachten, in welcher Weise die Lichtteilchen den Detektor erreichen. Hierbei kann man jedoch feststellen das beim Beschuss mit einem einzelnen Lichtimpuls dieser immer an einer anderen Stelle landet, wobei die Gebiete der Interferenzverstärkung Gebiete mit höherer Wahrscheinlichkeit darstellen und die „dunklen" Zonen, Gebiete mit niedriger Wahrscheinlichkeit. So kann man also allein aus dem Versuchs Aufbau nicht vorhersagen, wo jedes einzelne Lichtteilchen am Ende sein wird, sondern nur das Resultat einer großen Menge an Lichtteilchen auf einem Schirm. [9]

Kollaps der Wellenfunktion

Was uns auch schon zum nächsten Phänomen der Quantenmechanik bringt, und zwar dem Kollaps der Wellenfunktion. Wie wir bereits erfahren haben können wir nicht vorhersagen wo jedes einzelne Teilchen sein wird sondern nur deren Gesamtheit. Und dies Liegt wiederum an dem Wellenverhalten. Man stelle sich nun also einen Graphen vor mit einer Raum- und einer Zeitachse nun fügen wir ein Teilchen hinzu, welches jedoch nicht als dieses sondern als Wahrscheinlichkeitswelle dargestellt wird mit einem Maximum an der der Stelle, an welcher

[9] Heisenberg, o. J. [8]

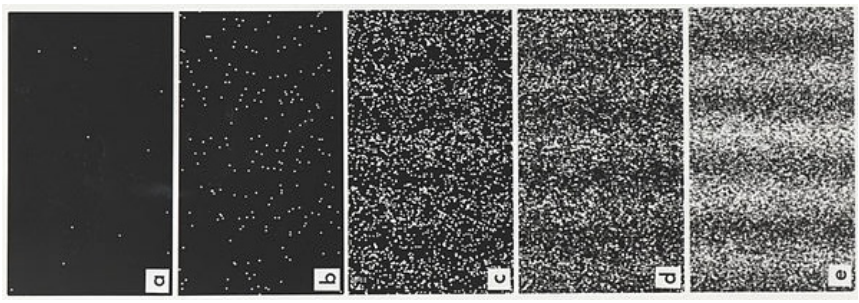


Abbildung 2.4: Aufbau des Interferenzmusters aus einzelnen Messpunkten

das Teilchen gerade gemessen wurde. Der Rest der Kurve kann durch eine Linie beschrieben werden, welche mit dem Abstandsbetrag zur 2. Potenz abfällt. Lassen wir nun Zeit vergehen sieht man, dass zwar immer noch ein Maximum am Punkt des gemessenen Teilchens vorliegt jedoch jetzt die Kurve wesentlich flacher verläuft. In Folge dessen wird es immer wahrscheinlicher das Teilchen an einen ganz anderen Ort vorzufinden als bei der ersten Messung. Daraus können wir also folgern, dass das Teilchen zu jedem Zeitpunkt an dem es nicht gemessen wird nur eine Wolke an Wahrscheinlichkeiten ist und wir erst erkennen können, welche dieser Möglichkeiten nun Wahr wird wenn wir es beobachten.

Felder

Diesem ist hinzuzufügen, dass wir für viele fundamentale Teilchen ein eigenes Feld vorliegt so haben wir für Elektronen ein Elektronenfeld für Gluonen ein Gluonenfeld usw. Sollte nun ein Teilchen des entsprechenden Feldes dieses durchqueren entsteht eine Art Loch, welches sich um das Teilchen erstreckt. Um dies deutlicher zu machen kann man Einsteins Interpretation von Gravitation heranziehen, welche besagt, dass sich der Raum um Ansammlungen von Energie krümmt und so eine Art Loch entsteht, in welches andere Objekte Hinein rollen können. Im Falle von Elektronenfeldern kann man sich diese Löcher eher als Wahrscheinlichkeit für das Erscheinen eines Teilchens bei einer Beobachtung vorstellen. So haben wir auch um jedes Atom eine mehr oder weniger ovale Einbuchtung im Elektronenfeld.

Tunneleffekt

Auch aus diesem Sachverhalt und der Eigenschaft, dass jedes Teilchen auch eine Welle ist lässt sich eine weitere Ungewöhnlichkeit aufdecken. Und zwar stellen wir uns vor wir haben einen Ball und werfen diesen gegen eine Wand. Im Normalfall kommt der Ball immer wieder zurück gesprungen. Jedoch in der Quantenwelt ist dies nicht immer der Fall also nehmen wir statt des Balles ein Elektron, welches ein fundamentales Teilchen und damit ein Quantenobjekt ist. Werfen wir nun dieses Elektron gegen eine entsprechende Barriere wird dieses auch wie in der klassischen Physik zurückkommen jedoch nicht immer. In manchen Fällen wird das Elektron auf der einen Seite der Barriere verschwinden und auf der anderen Seite wieder auftauchen, so als ob es direkt durch die Wand hindurch geworfen wurde. Doch lässt sich auch dieses Phänomen bekannt als Tunneleffekt mit den bisherigen Erkenntnissen erklären. So wissen wir, dass das Elektron in seinem eigenen Feld eine Art Loch erzeugt welches zeigt das in dieser Region das Elektron sein muss. An der Stelle der Barriere muss jetzt jedoch die tiefe dieses Loches null betragen, da nicht zwei Objekte am gleichen Ort sein können. Jedoch auf der anderen Seite der Barriere Hindert das Feld nichts daran das theoretische Wahrscheinlichkeitsloch weiterzuführen. Also ist das Elektron auch mit einer bestimmten Wahrscheinlichkeit dort. So kann es passieren, dass nun das Teilchen anscheinend durch sofortige Teleportation auf der anderen Seite gelandet ist jedoch lässt sich dieses Argument leicht auflösen wenn man sich das Teilchen wiederum nicht als Teilchen sondern als Wahrscheinlichkeitswolke, denn in diesem Fall war das Teilchen ja von Anfang an, an keinem bestimmten Ort und kann sich deswegen auch nicht „Teleportiert“ haben. Für den Beobachter hat es nur den Schein als würde es dies tun.[10]

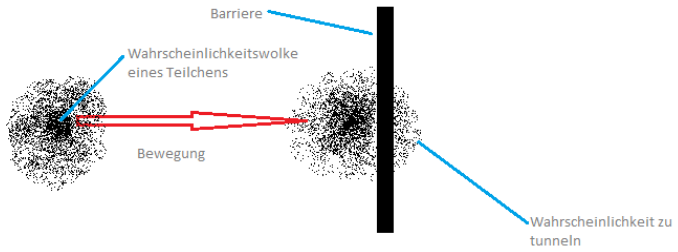


Abbildung 2.5: Zeigt auf das "Teilchhen", dass auch auf der anderen Seite erscheinen kann

Spin

Der Spin ist eine der Eigenschaften von Atomkernen und fundamentalen Teilchen. Der Begriff des Spins ist taucht beispielsweise im Krankenhaus bei der Kernspintomographie auf. Jedes mikroskopische Objekt, welches nur aus einem Teilchen besteht, hat einen Eigenspin. Dazu gehören Gluonen, Quarks und Leptonen. Aber auch größere und zusammengesetzte Teilchen besitzen einen Spin, welcher sich aus dem Spin der einzelnen Bestandteile ergibt. So müssen zum Beispiel bei einem Proton, die Spins der einzelnen Quarks zusammengezählt werden, um den Gesamtspin zu ermitteln. Mit steigender Anzahl von Teilchen, wie in einem Atom

[10]Milq, o. J. [9]

oder gar einem Molekül, wird dies, aufgrund zunehmender Entfernung zum mikroskopischen Kosmos, zunehmend komplizierter. Weiterhin jedoch verhält sich der Spin wie eine Drehung im Sinne der klassischen Physik und so hat auch hier jedes Teilchen, was einen Spin besitzt, einen Drehimpuls. Der Spin kann durch bestimmte Krafteinwirkungen seine Rotationsachse verändern. Dabei wird auch hier bei einer größeren Rotation der Drehachse mehr Energie benötigt. Hinzuzugeben ist, dass ein Teilchen sowohl mit als auch gegen den Uhrzeigersinn rotieren kann, was im Endeffekt eine Auswirkung auf den Verlauf der Rotationsachse hat.

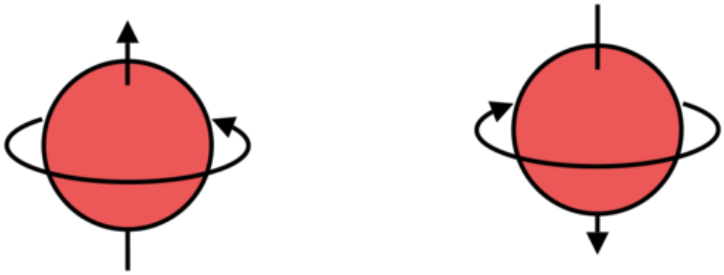


Abbildung 2.6: Zeigt zwei Teilchen mit entgegengesetztem Spin

Polarisierung und Wellen

Sogenannte Polarisationsfilter werden beispielsweise in 3D-Brillen oder an Fensterscheiben von Autos benutzt.Diese dienen polarisieren und filtern von Lichtwellen, was eine wichtige Rolle in der technisierten Welt spielt. Was ist eine Polarisation und was unterscheidet polarisiertes Licht von nicht polarisiertem? Jedes Photon oder jeder Lichtimpuls ist auf eine bestimmte Weise polarisiert. Dabei gibt es verschiedene Arten: Zum einen die lineare Polarisierung, welche im Modell verdeutlicht, dass die Lichtwelle, solang sie keinen äußeren Einflüssen unterliegt, auf einer bestimmten Achse schwingt. Anders ist dies bei der zirkularen oder kreisförmigen Polarisation. Hierbei ist die Auslenkung der Welle abhängig von Ort und Zeit, wo hingegen die Lineare Polarisation von keiner dieser beiden Größen abhängig ist. [11]

[11] Prior, 2009 [10]

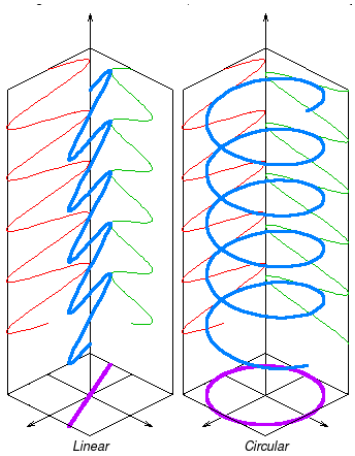


Abbildung 2.7: linear polarisiert(links)/zirkular polarisiert(rechts)

Die elliptische Polarisation ist noch komplexer da hier auf die Rotation der Polarisationsachse noch eine Verschiebung zur Z-Achse folgt. Vereinfacht ist, aus der Sicht aus der Perspektive eines Beobachters, auf der Z-Achse ein Oval, jedoch entsteht Rotationskreis bei einer Draufsicht, ganz ähnlich wie bei der kreisförmigen Polarisation.

2.2.3 Technische Grundlagen

In vielen Fällen ist es sehr viel einfacher eine Theorie zu einem Sachverhalt aufzustellen, als tatsächlich eine Umsetzung anzubieten. Dies gilt vor allem für Quantencomputer, welche in der Theorie seit vielen Jahren erdacht und erweitert werden, jedoch erst seit kurzer Zeit in der Realität umgesetzt werden können. Hierbei liegt es aber nicht am fehlenden Interesse an einer solchen Technologie sondern an hohen Kosten und mangelndem Fortschritt. Viele der Technologien, welche notwendig sind um solche Systeme umzusetzen, sind erst seit wenigen Jahrzehnten bekannt. So zum Beispiel Supraleiter oder bestimmte Messgeräte zur Ausrichtung von Spin oder der Ladung kleinster Teilchen.
Da es, wie bereits in den physikalischen Grundlagen[12] erwähnt wurde, viele verschiedene Methoden ein Qubit zu erzeugen gibt und entstehen damit auch viele technische Möglichkeiten diese umzusetzen. Jedoch geben Institutionen, wie Google oder IBM, welche an solchen neuen Technologien arbeiten, nur selten ihren aktuellen Forschungsstand an die Öffentlichkeit weiter. Dies bedeutet, dass sich in vielen Fällen auf die Theorie gestützt werden muss um eventuelle technische Umsetzungen zu erfassen.

[12] siehe Kapitel 2.1.2: Physikalische Grundlagen

Ionenfallen

Ionenfallen sind die wohl bekannteste und am stärksten etablierte Methode eine große Anzahl verschränkter Qubits zu erschaffen. Hierbei wird durch stromdurchflossene Leiter ein magnetisches Feld erschaffen, welches die Ionen in einer Linie hält. Durch das gleichschalten des Stromes synchronisieren sich die, sich im Grundzustand befindlichen, Ionen und durch Anregung eines Photons können diese in einen bestimmten Quantenzustand gebracht werden. In diesem Falle entsteht durch Photonenemission ein gezielter Lichtimpuls, welcher zur Anregung eines beliebigen Ions verwendet wird, um es in einen anderen Zustand zu versetzen.

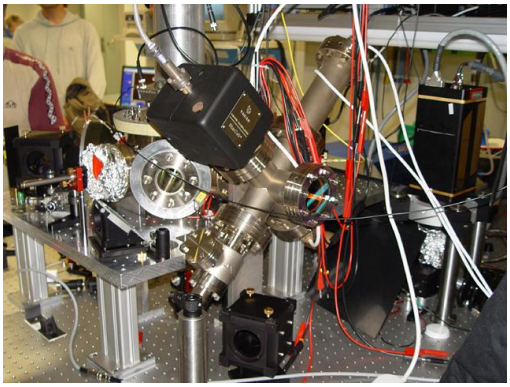


Abbildung 2.8: Ionenfalle als Laboraufbau

Magnetresonanz

Wie man am Beispiel des ibmqx2, welcher von IBM entwickelt und zum Teil für die Öffentlichkeit zur Verfügung gestellt wurde, sehen kann, gibt es auch Unternehmen, welche ihren Fortschritt mit der Gesellschaft teilen wollen. Er funktioniert mit Hilfe von Magnetresonanz: Ein Atomkern wird in einem Magnetfeld festgehalten und durch einen Lichtimpuls im Mikrowellenbereich angeregt. Wie bereits in den Physikalischen Grundlagen erwähnt wurde, zerfällt dieser Zustand nach kurzer Zeit wieder. Im Falle des ibmqx2 beträgt diese Zerfallszeit 83 Nanosekunden, in welchen die Gatter angewandt und die Messungen vollzogen werden müssen. Der "Prozessor"des Quantencomputers hat dabei eine Betriebstemperatur von 16 Millikelvin, um die Spulen zu jedem Zeitpunkt Supraleitend zu erhalten.

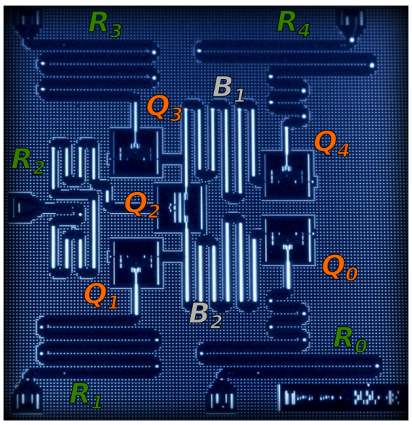


Abbildung 2.9: IBM Quantencomputer(Vorlage für das Programm

Annäherungsversuche

Am Beispiel von Google sieht man, dass es auch ganz andere Methoden gibt, die Leistungsfähigkeit von Rechenmaschinen zu steigern. Vielleicht hat man in den Medien bereits von Googles Quantencomputer gehört, was jedoch dahinter steht wissen die wenigsten. In der Realität ähnelt der Rechner, welcher von der Firma D-WAVE hergestellt wird, eher einem ganz gewöhnlichen Prozessor eines herkömmlichen PC's, als einem Quantencomputer. In der Tat wird auch ein beinahe ganz normaler Prozessorkern in den D-WAVE eingebaut. Ein entscheidender Unterschied ist jedoch, das der Prozessor auf 1 Kelvin herunter gekühlt wird und somit in einen Supraleitenden Zustand versetzt wird. Mit dieser Methode kann man die Leistung eines Prozessors ins millionenfache steigern da nun die Elektronen ungehindert die Halbleiter passieren können. Doch dabei ist der Nachteil, dass sich die Steigerung der Leistung immer noch Linear und nicht exponentiell wie bei einem wahren Quantencomputer verhält.[13]

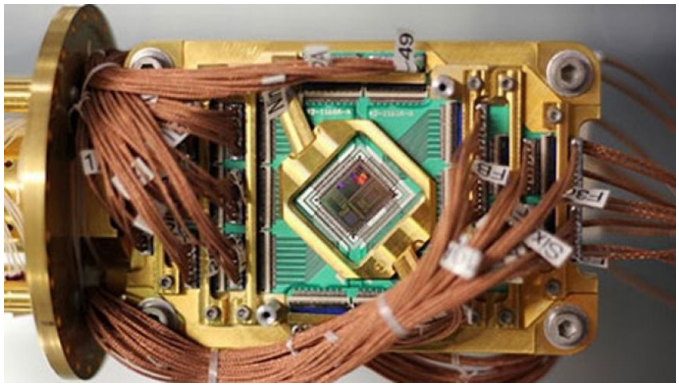


Abbildung 2.10: IBM Quantencomputer(Vorlage für das Programm)

[13]D-Wave Systems, o.J. [11]

2.3 Quantengatter

Um mit einem Quantencomputer und den Qubits arbeiten zu können, benötigt man sogenannte Quantengatter. Quantengatter wirken sich, im Gegensatz zu herkömmlichen logischen Gattern, auf die verschiedenen digitalen Superpositionszustände der Qubits aus. Quantengatter können daher durch mathematische Operatoren, wie Matrizen, beschrieben werden. Um Quantengatter besser betrachten zu können sollte, von Beginn an, zwischen realen und idealen Quantencomputern unterschieden werden. Der Unterschied hierbei ist, dass in einem idealen System die nächste Operation auf der Grundlage der Voraussage der letzten Operation gemacht werden kann. In einem realen System ist dies nicht möglich, da ohne Erhalt der Quantenzustände deren Zustand verfällt und der Qubit versucht, in einen Grundzustand zurückzukehren. Dieser Prozess geschieht weiterhin nicht in absehbarer Zeit sondern zufällig, was bei Experimenten anhand der starken Abweichungen der Zustände zur Vorhersage deutlich wird.

2.3.1 Negationsgatter

Not Gatter

Es gibt zwei Möglichkeiten, um die Idee eines Not-Gatters zu interpretieren. Die einfachere Interpretation ist die eines Tausch-Gatters, welches seine negierende Wirkung in den Amplituden der Grundzustände des Qubits sieht. Dieser Tausch wird durch eine sogenannte „Exchange Matrix“, eine antidiagonale quadratische Matrix der Größe 2 mit Einsen als einzige Einträge ungleich 0 umgesetzt. Diese Matrix ist auch als X-Pauli-Gatter bekannt[14].

$$|\psi_1\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \qquad\qquad Tausch\,(|\psi_1\rangle) = \beta\,|0\rangle + \alpha\,|1\rangle$$

$$X = Tausch = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

Etwas komplexer ist die Interpretation des Not-Gatters als Anti-Gatter mit der Zielsetzung einer Zustandsamplitudenveränderung der Qubitgrundzustände zu einem senkrechten Qubitvektor in der zweidimensional komplexen Qubitdarstellungsweise. Dies hat aufgrund der senkrechten Grundvektoren $|0\rangle$ und $|1\rangle$ eine negierende Wirkung auf die Amplituden des Qubits.

$$|\psi_1\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \qquad\qquad Anti\,(|\psi_1\rangle) = |\psi_1^{\perp}\rangle = \alpha^*\,|0\rangle - \beta^*\,|1\rangle$$

Es ist allerdings aufgrund komplexer Gründe außerhalb des Verständnisbereiches der Autoren selbst theoretisch unmöglich, ein Anti-Gatter zu verwirklichen, da bewiesenermaßen kein Gatter existieren kann, welches diese Aufgabe perfekt erfüllt.[15]

[14] siehe Kapitel 2.3.2: Pauligatter

[15] Shumovsky, Rupasov, 06.12.2012, S.67 [12]

C-Not Gatter

Ein C-Not Gatter oder Control-Not Gatter hat mit einer Einschränkung dieselbe Wirkung wie das X-Pauli Gatter. Es wirkt nur auf das Zielqubit, falls sich ein spezifiziertes Kontrollqubit im Zustand $|1\rangle$ befindet.

$$q_{control} = |0\rangle \Rightarrow CNot_{target} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \mathbf{I} \quad q_{control} = |1\rangle \Rightarrow CNot_{target} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \mathbf{X}$$

Wendet man ein C-Not Gatter auf ein n-Qubit System an, so hat man n^2 Möglichkeiten für die Position des C-Not Gatters abhängig vom Kontrollqubit und vom Zielqubit. Damit streckt sich das C-Not Gatter theoretisch über 2 Qubits, praktisch muss die Größe durch die Gesamtgatterberechnung mittels Kroeneckermultiplikation[16] jedoch mindestens die Anzahl der Qubits sein, über die sich das C-Not Gatter erstreckt. Außerdem hat jede dieser Möglichkeiten eine andere Wirkung auf das System und muss deshalb zwangsläufig eine eigene Matrixdarstellung besitzen.

$$CN_{(Gesamtqubitanzahl,Kontrollqubit,Zielqubit)}$$

$$CN_{(2,1,0)} = \begin{bmatrix} 1&0&0&0 \\ 0&0&0&1 \\ 0&0&1&0 \\ 0&1&0&0 \end{bmatrix} \qquad CN_{(2,0,1)} = \begin{bmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&1 \\ 0&0&1&0 \end{bmatrix} \qquad CN_{(3,2,0)} = \begin{bmatrix} 1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1 \\ 0&0&0&0&1&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&0 \\ 0&0&0&1&0&0&0&0 \end{bmatrix}$$

Abbildung 2.11: Beispiele einiger ausgewählter C-Not Gattermatrizen

Die Herleitung der C-Not Gattermatrix ist aufwendig. Die intuitivste Herangehensweise für die Herleitung der Gattermatrix eines C-Not Gatters über m Qubits ist der Beginn mit einer quadratischen Nullmatrix der Größe 2^m. Dann weist man jeder Zeile und jeder Spalte den korrespondierenden Teilzustand zu, geht nacheinander alle Zeilen und damit Ausgangsteilzustände durch und setzt eine Eins in der Spalte, die den Zustand darstellt, zu dem der Ausgangsteilzustand verändert wurde.

$$CNot_{(m,v1,v2)} = \begin{bmatrix} 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \begin{matrix} \\ \Rightarrow \\ \\ \end{matrix} \begin{matrix} |\cdots 00\rangle \\ |\cdots 01\rangle \\ |\cdots 10\rangle \\ \vdots \end{matrix} \begin{matrix} |\cdots 00\rangle \; |\cdots 01\rangle \; |\cdots 10\rangle \cdots \\ \begin{bmatrix} 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \end{matrix} \Rightarrow \begin{bmatrix} 1 & 0 & 0 & \cdots \\ 0 & 0 & 1 & \cdots \\ 0 & 1 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

[16] siehe Kapitel 2.2.1 Mathematische Grundlagen

CC-Not Gatter

Ein CC-Not Gatter oder Toffoli Gatter[17] ist eine Variante eines kontrollierten Not-Gatters mit zwei Kontrollqubits, im Unterschied zum C-Not Gatter. Es müssen sich beide Kontrollqubits im Zustand $|1\rangle$ befinden, damit ein X-Gatter auf das Zielqubit wirkt. Die Größe hängt wieder von der Anzahl der Qubits ab, die das CC-Not Gatter überspannt. Das mittlere CCNot Qubit ist dabei irrelevant für die Arraygröße, aber nicht für die Gattermatrix, welche nach dem gleichen Schema wie das C-Not Gatter erstellt werden kann.

C_n-Not Gatter

Theoretisch kann die Vorgehensweise auf beliebig viele Kontrollqubits ausgedehnt werden, praktisch werden größere C_n-Not Gatter jedoch wegen der wachsenden Komplexität und Verlangsamung[18] von Qubitsystemen nicht für Qubitalgorithmen verwendet.

2.3.2 Identitygatter

Das simpelste Quantengatter ist das Identity-Gatter. Dabei handelt es sich, im Gegensatz zu allen anderen Gattern, um keine Operation die den Zustand des Qubits aktiv verändert, sondern nur um eine Anweisung zu warten und damit den Zustand des Quantenbits langsam verfallen zu lassen. Dies ist jedoch nur in einem realen Quantencomputer nützlich, da ideale Systeme nicht zerfallen und damit kein Zweck für ein solches Gatter besteht. Hinzuzufügen ist, dass dies vor, während, aber auch nach anderen Operationen angewendet werden kann.

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

2.3.3 Pauligatter

Für jedes Gatter existiert eine Matrix, welche eine Operation auf die Zustandsmatrix des Qubits darstellt, welche auf einen Qubit angewendet einen vom Anfangszustand verschiedenen Zustand erzeugt (Identity-Gatter ausgenommen). Die simpelsten dieser Gatter sind die Pauli-X,Y und Z-Gatter, welche alle samt einfache Qubitoperationen sind. Das erste Gatter, das X-Pauli-Gatter, oder auch Not-Gatter genannt, ist das erste der Pauli Gatter und verursacht eine Negierung der Amplituden der Grundzustände. Zudem wird die Transformation durch die unten stehende Matrix beschrieben. Bei der Anwendung eines X-Gatters werden alle orthogonal aufeinander stehenden Achsen, bei einer Spiegelung an der X-Achse, einer vierdimensionalen Rotation unterzogen, was den Körper im Ganzen spiegelt.

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

[17] Benannt nach Tomaso Toffoli, „ECE“ Forschungsprofessor der „Boston University“
[18] Gross, Flammia, Eisert, 2009, Abstract [13]

Das zweite Pauli-Gatter ist das Y-Gatter. Es sorgt für eine Negation der Phasen von X und Z und führt damit einen sogenannten Bit- und Phaseflip durch. Hierbei wird wieder die Amplitude der Grundzustände vertauscht. Man kann dieses Gatter als eine Kombination aus dem X- und Z-Gatter verstehen, da beide zustände negiert werden.

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

Das dritte und letzte Pauli Gatter stellt das Pauli-Z-Gatter dar. Bei Anwendung auf den Grundzustand kann man keine Veränderung des Messergebnisses feststellen, da es sich nur um einen Phaseflip handelt, welcher das gegebene Qubit nicht beeinflusst. Dies hängt damit zusammen, das bei einer Messung des Qubitzustandes immer in Z-Richtung gemessen wird. Dies erklärt auch warum die Spiegelung an der Z-Achse keinen Einfluss auf die Z-Messung hat.

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

2.3.4 Hadamardgatter

Im Gegensatz zu den Pauli-Gattern ist das Hadamard-Gatter auf Mehr-Qubit-Systeme anwendbar, da die Matrix nicht statisch, sondern rekursiv auf mehrdimensionale Systeme anwendbar ist. Das Hadamard Gatter ist auch das erste Gatter, welches Superpositionszustände erzeugen. Auch die beiden nächsten Gatter, S und Sgd, können Superpositionen herstellen, was aber noch näher erläutert wird. Bei diesem Gatter findet eine Phasenverschiebung von 90 Grad statt, welche zum Beispiel die Amplitude des Y-Zustandes negieren kann. Eine zusätzliche Eigenschaft des Hadamardgatters ist, dass bei hintereinander Schaltung zweier Hadamard-Gatter der Zustand vor der ersten Transformation vorliegt.

$$H = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

2.3.5 Phasegatter

Das S-Gatter erzeugt durch eine simple viertel Drehung oder eine Rotation um die z-Achse. Dabei passiert es, dass Y-Vektor gleich X-Vektor und X der negierten Amplitude von Y entspricht. Wie schon erwähnt, generiert es zudem einen Superpositionszustand.
Das Sdg-Gatter verhält sich ähnlich wie das S- Gatter nur, dass nun der X- gleich dem Y-Vektor ist und Y nun der negierten Amplitude von X entspricht.

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} Sdg = \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix}$$

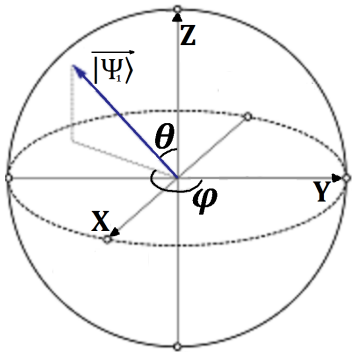
2.4 Messungen von Qubits

2.4.1 Standardmessung auf $|0\rangle$ / $|1\rangle$ Basis

Bei der Standardmessung auf $|0\rangle$ / $|1\rangle$ Basis geht es vor allem darum, die durch den Quantenalgorithmus erreichten, Ergebnisse, in einer auswertbaren Weise, auszugeben. $|0\rangle$ beschreibt hierbei den Grundzustand, welchen man immer erhält, wenn keine Operation angewendet wurde. Durch eine Verwendung eines Gatters kann dieser nun beliebig manipuliert werden, so dass als Ergebnis, eine Reihe von Ergebnissen für $|0\rangle$ und $|1\rangle$ entsteht. Zum Schluss wird die Amplitudenmatrix des Endsuperpositionszustandes, eines oder mehrerer Qubits, zurück gegeben. Die Zustände dieser Qubits können zudem auch komplex sein, da ein Qubit, neben den Standartbasisvektoren, eine Phase besitzt. Um nun diese Superpositionen darzustellen, wird nun ein Koeffizient beigesetzt[19], wie zum Beispiel $\alpha|0\rangle + \beta|1\rangle$. Dies beschreibt den allgemeinen Zustand eines Qubits, wobei die Koeffizienten positiv, negativ und komplex sein können. Wird nun $|\alpha|^2 + |\beta|^2$ quadriert, ergibt sich die Wahrscheinlichkeit für die Messung des Zustandes $|0\rangle$ oder $|0\rangle$. Dies lässt sich auch auf Mehr-Qubit-Systeme übertragen, wobei sich,für jedes zusätzliche Qubit, die Anzahl von möglichen Zuständen verdoppelt.

2.4.2 Blochkugelmessung

Speziell für Visualisierungen von einzelnen Qubits wird eine besondere Darstellungsweise benutzt: die Blochkugel[20].



$$\overrightarrow{Z_{max}} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = |0\rangle \qquad \overrightarrow{Z_{min}} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} = |1\rangle$$

$$\overrightarrow{Y_{max}} = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} = \frac{|0\rangle + i|1\rangle}{\sqrt{2}} \qquad \overrightarrow{Y_{min}} = \begin{pmatrix} 0 \\ -1 \\ 0 \end{pmatrix} = \frac{|0\rangle - i|1\rangle}{\sqrt{2}}$$

$$\overrightarrow{X_{max}} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \qquad \overrightarrow{X_{min}} = \begin{pmatrix} -1 \\ 0 \\ 0 \end{pmatrix} = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

$$\overrightarrow{XY_{\phi}} = \begin{pmatrix} sin(\phi) \\ cos(\phi) \\ 0 \end{pmatrix} = \frac{|0\rangle + e^{i\phi}|1\rangle}{\sqrt{2}}$$

Abbildung 2.12: Blochkugelschema und bestimmte Blochvektoren

Die Blochkugel ist eine Einheitskugel, welche dazu benutzt wird, alle Zustände eines Qubits der Form $\alpha|0\rangle + \beta|1\rangle$ als Blochvektor der Länge 1 mithilfe von Kugelkoordinaten $(1, \theta, \phi)$ auf ihr darzustellen. Verschränkte Qubits können auch einzeln dargestellt werden, befinden sich dabei jedoch in der Einheitskugel.

[19] siehe Kapitel 2.1.2: Physikalische Grundlagen
[20] Terr, o.J, S.1 [14]

$$
\begin{aligned}
|\psi_1\rangle_{Standard} &= \alpha\,|0\rangle + \beta\,|1\rangle && \text{(normale Schreibweise)}\\
|\psi_1\rangle_{Bloch} &= cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\phi}\cdot sin\left(\frac{\theta}{2}\right)|1\rangle && \text{(Blochvektordefiniton)}\\
&= cos\left(\frac{\theta}{2}\right)|0\rangle + (cos(\phi) + i\cdot sin(\phi))\cdot sin\left(\frac{\theta}{2}\right)|1\rangle \quad 0<\theta<\pi\ ,\ 0<\phi<2\pi\\
&\Rightarrow \alpha\in\mathbb{R}\wedge\beta\in\mathbb{C}
\end{aligned}
$$

Um $|\psi_1\rangle$ in Blochvektorform darstellen zu können, muss α jedoch reell sein. Es kann allerdings jede Zustandsmatrix eines einzelnen Qubits so verändert werden, dass eine weiterhin normierte Zustandsmatrix entsteht, welche denselben Qubitzustand wie die Ausgangsmatrix beschreibt, gleich normiert ist, aber ein reelles α beinhaltet.

$$
\begin{pmatrix}\alpha\\ \beta\end{pmatrix} \equiv c\begin{pmatrix}\alpha\\ \beta\end{pmatrix} = \begin{pmatrix}c\alpha\\ c\beta\end{pmatrix} \qquad c := \frac{\bar{\alpha}}{|\alpha|}
$$

$$
\Rightarrow \begin{pmatrix}\alpha\\ \beta\end{pmatrix} \equiv \begin{pmatrix}\frac{\alpha\cdot\bar{\alpha}}{|\alpha|}\\ \frac{\beta\cdot\bar{\alpha}}{|\alpha|}\end{pmatrix} \qquad \frac{\alpha\cdot\bar{\alpha}}{|\alpha|} = \frac{\Re(\alpha)^2+\Im(\alpha)^2}{\sqrt{\Re(\alpha)^2+\Im(\alpha)^2}} = \sqrt{\Re(\alpha)^2+\Im(\alpha)^2}\in\mathbb{R}
$$

$$
\begin{aligned}
|\alpha| &= \sqrt{\Re(\alpha)^2+\Im(\alpha)^2} = |c\alpha|\\
\Rightarrow |\beta| &= |c\beta|
\end{aligned}
$$

Sollen beide Zustandsamplituden des in der Blochkugel dargestellten Qubits aus einem Blochvektor berechnet werden, dann berechnet man die Real- und Imaginärteile von α und β einzeln durch trigonometrische Funktionen und setzt diese dann zusammen.

$$
\alpha\,|0\rangle + \beta\,|1\rangle = cos\left(\tfrac{\theta}{2}\right)|0\rangle + e^{i\phi}\cdot sin\left(\tfrac{\theta}{2}\right)|1\rangle \tag{2.3}
$$

$$
\begin{aligned}
&\Rightarrow \Re(\alpha) = \alpha = cos\left(\frac{\theta}{2}\right) && \Rightarrow \Im(\alpha) = 0\\
&\Rightarrow \Re(\beta) = cos(\phi)\cdot sin\left(\frac{\theta}{2}\right) && \Rightarrow \Im(\beta) = sin(\phi)\cdot sin\left(\frac{\theta}{2}\right)
\end{aligned}
$$

Ist eine Umrechnung in die andere Richtung, von den Zustandsamplituden zum Blochvektor, gefragt, nutzt man die umgestellten Gleichungen, um eine Lösung zu erhalten.

$$
\alpha = cos\left(\tfrac{\theta}{2}\right) \tag{2.4}
$$

$$
\Re(\beta) = cos(\phi)\cdot \sin\left(\tfrac{\theta}{2}\right) \qquad \Im(\beta) = sin(\phi)\cdot sin\left(\tfrac{\theta}{2}\right) \tag{2.5}
$$

$$\Rightarrow \frac{\theta}{2} = cos^{-1}(\alpha) \qquad \Rightarrow \theta = 2 \cdot cos^{-1}(\alpha)$$

$$\Rightarrow cos(\phi) = \frac{\Re(\beta)}{sin\left(\frac{\theta}{2}\right)} = \frac{\Re(\beta)}{\sqrt{1-\alpha^2}} \qquad \Rightarrow \phi = cos^{-1}\left(\frac{\Re(\beta)}{\sqrt{1-\alpha^2}}\right)$$

$$\Rightarrow sin(\phi) = \frac{\Im(\beta)}{sin\left(\frac{\theta}{2}\right)} = \frac{\Im(\beta)}{\sqrt{1-\alpha^2}} \qquad \Rightarrow \phi = sin^{-1}\left(\frac{\Im(\beta)}{\sqrt{1-\alpha^2}}\right)$$

$$\text{falls } \Im(\beta) < 0 \Rightarrow \phi_{richtig} = 2\pi - \phi_{berechnet}$$

Wird letztendlich noch der Blochvektor im kartesischen Koordinatensystem mit den gleichen Koordinatenachsen der Blochkugel benötigt, lassen sich die Skalare der einzelnen Grundvektoren für den Ergebnisvektor mithilfe der Umrechnungsgleichungen vom Kugelkoordinatensystem ins kartesische Koordinatensystem berechnen.

$$\overrightarrow{|\psi_1\rangle}_{polar} = \overrightarrow{v(1,\theta,\phi)} \Longrightarrow \overrightarrow{|\psi_1\rangle}_{kartesian} = \overrightarrow{v(x,y,z)}$$

$$x = sin(\theta) \cdot cos(\phi)$$
$$y = sin(\theta) \cdot sin(\phi)$$
$$z = cos(\theta)$$

Kapitel 3

Quantencomputer

Als Quantencomputer bezeichnet man eine Rechenmaschine, welche Gesetze und Phänomene der Quantenmechanik nutzt, um einen Geschwindigkeitsvorteil gegenüber einem herkömmlichen Computer, welcher auf die Möglichkeiten der klassischen Physik begrenzt ist, zu erzielen. Nachdem die mathematischen und physikalischen Grundlagen des Qubits, sowie die Umsetzung der Quantengatter zur Manipulation dieser Qubits erklärt wurden, wird nun der Quantencomputer selbst näher betrachtet. Zuerst wird näher auf den Aufbau eingegangen und danach werden die Unterschiede zum herkömmlichen Computer herausgearbeitet.

3.1 Aufbau eines Quantencomputers

Der Aufbau eines Quantencomputers ist abhängig von der Art der Realisierung der Qubits und daher ist es schwierig konkrete Aussagen über eine allgemeine Bauweise zu formulieren. Daher werden einige Ansätze in den voran gegangenen Abschnitten über Möglichkeiten zur Umsetzung von Qubits näher betrachtet[1].
Je nach Art der Realisierung der Qubits sind dann unterschiedliche Apparaturen von Nöten um zum einen die Qubits zu implementieren und zum anderen die Quantengatter, welche der Manipulierung der Zustände dienen, zu erzeugen. Weiterhin variieren die Geräte zur Messung des Qubitzustandes.

3.2 Besonderheiten des Quantencomputers

3.2.1 Funktionsweise des herkömmlichen Computers

Um die Funktionsweise eines Computers zu verstehen muss zuerst betrachtet werden, was unter einem Computer verstanden wird. [2] Unter diesem Begriff werden alle Recheneinheiten gefasst, die mechanisch oder elektronisch, mittels programmierbarer Rechenvorschriften Daten verarbeiten. Somit fallen nicht nur PC in diese Definition, sondern auch Smartphones, Mikrochips mit Mikroprozessoren und andere Geräte die auf elektrischer Basis Daten

[1] siehe Kapitel 2.1.1: Theoretische Qubitarten
[2] Schanze , November 2016 [16]

verarbeiten. Das erste Gerät dieser Art war der 1941 von Konrad Zuse gebaute Z3. Dieser war der erste Digitalrechner weltweit. Die Vorgänger des Z3, also die Zuse Z1 und die Z2, waren mechanisch realisiert worden, während der Z3 mit elektrischen Relais gebaut wurde. Des Weiteren besaß der Z3 bereits viele Merkmale heutiger Rechner. Zahlen wurden bereits binär codiert und der Z3 war bereits in der Lage, Gleitkommazahlen zu berechnen. Weiterhin besaß er Ein- und Ausgabegeräte und Mikroprogramme, mit denen, so weit dies leistungstechnisch möglich war, parallele Operationen ausgeführt werden konnten. Konrad Zuse war damit einer der größten Vordenker für die 1945 vom österreichischen Mathematiker John von Neumann, veröffentlichte Arbeit zum Thema des universellen Aufbaus eines Computers. Heute werden alle Computer nach dieser sogenannten Von-Neumann-Architektur [3] aufgebaut.

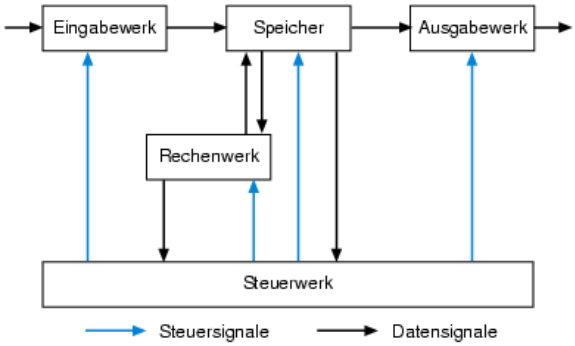


Abbildung 3.1: Computerarchitektur nach John von Neumann

Nach Neumanns Theorie besteht ein universeller Computer aus 5 Baugruppen, dem Steuerwerk, dem Rechenwerk, einem Speicher sowie Eingabewerk und Ausgabewerk, welche alle miteinander interagieren. Die durch das Eingabewerk erfassten Daten werden im Speicher abgelegt und mit dem Rechenwerk ausgetauscht, welches verschiedene Rechenoperationen auf die Daten anwendet. Die Daten werden im Speicher ablegt. Danach werden die Daten mit Hilfe des Ausgabewerkes ausgegeben. Der ganze Prozess wird vom Steuerwerk überwacht und gesteuert, indem es den eingegebenen Algorithmus interpretiert und die anderen Komponenten danach ansteuert. Neben dem Aufbau traf John von Neumann auch Aussagen über die Arbeitsweise des Rechners. So müssen die Zahlen im Rechner binär codiert werden und die Struktur des Rechners muss unabhängig vom zu lösenden Problem sein (Prinzip des Universalrechners). Laut Neumann werden Daten und Programm im selben Speicher abgelegt, welcher aus fortlaufend nummerierten, gleich großen Zellen besteht. Über diese Speicheradresse erfolgt der Zugriff auf die Daten durch das Steuerwerk.
Bevor auf die konkrete Umsetzung eingegangen wird, soll auf die Umsetzung der binär codierten Zahlen im Computer eingegangen werden. Natürliche Zahlen müssen, um sie elektronisch verarbeiten zu können, binär codiert werden. Es gibt verschiedene Möglichkeiten, eine Dezimalzahl in eine Binärzahl umzuwandeln, wobei die besondere Aufmerksamkeit den Gleitkommazahlen gebührt. (In Endfassung näher betrachten) Der Kern aller Systeme bildet die Darstellung einzelner Stellen der Binärzahl durch Bits dargestellt werden, welche aneinandergereiht die Binärzahl ergeben. Bits sind Zweizustandssysteme, die die Zustände 1

[3] Wiener, März 2016 [16]

oder 0 haben können. Im Computer werden diese Bits durch die Zustände des elektrischen Stroms auf den Leiterbahnen (0 bei keinem Stromfluss und 1 bei Stromfluss), durch die Ladung der Kondensatoren im Speicher oder durch den momentanen elektrischen Widerstand eines Transistors ausgedrückt. Mittels logischer Operationen, welche mit Hilfe von Gattern realisiert werden, können die Zustände gezielt manipuliert werden, sodass es möglich ist mit den Binärzahlen Berechnungen auszuführen.
Nach der Betrachtung des prinzipiellen Aufbaus und der Handhabung der Zahlen eines herkömmlichen Computers, wird nun die Technische Umsetzung und die genauere Funktionsweise betrachtet. Die einzelnen Werke werden technisch durch die verschiedenen Bauteile eines Computers realisiert: Rechen- und Steuerwerk werden im Prozessor untergebracht, der Speicher wird durch den RAM, also den Arbeitsspeicher, und die Festplatte zur langfristigen Datensicherung realisiert, während Tastatur und Bildschirm Ein- und Ausgabewerk darstellen.
Der Arbeitsspeicher, oder auch RAM, besteht aus sehr vielen kleinen Kondensatoren, welche die Zustände eines Bits (1 oder 0) annehmen können, indem sie entweder geladen oder ungeladen sind.[4] Da sich die Kondensatoren nach einiger Zeit von allein entladen, müssen diese regelmäßig in gewissen Zeitabständen, sogenannten Auffrischungszyklen, nachgeladen werden. Durch die Kopplung der Kondensatoren mit Transistoren ist es möglich den Zustand eines Kondensators zu verändern oder wiederherzustellen. Durch die Anordnung der Transistoren in Form einer Matrix ist ein Zugriff über Zeilen- und Spaltennummer möglich.

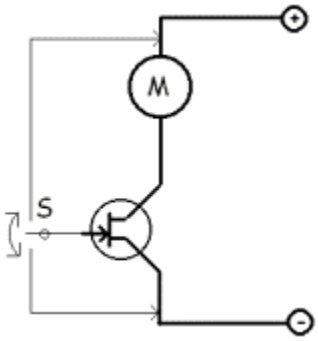
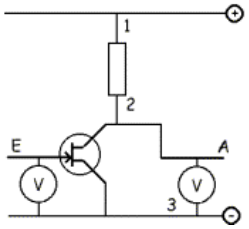


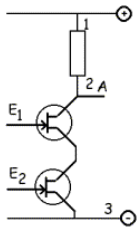
Abbildung 3.2: Schaltplan eines Schalttransistors

Transistoren werden nicht nur im flüchtigen Speicher verwendet sondern sie stellen den Kern der heutigen Digitalrechner dar. Wenn man Transistoren auf bestimmte Art koppelt kann man die grundlegenden logischen Gatter realisieren. Durch die Verknüpfung der Grundgatter, also der NOT-Schaltung, der NAND-Schaltung und der NOR-Schaltung kann man weitere logische Gatter, wie das AND-Gatter oder das OR-Gatter formen.

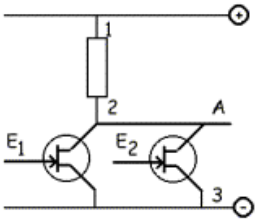
[4]User: Jeff, November 2012 [17]



(a) Schaltplan eines NOT-Gatters



(b) Schaltplan eines NAND-Gatters



(c) Schaltplan eines NOR-Gatters

Abbildung 3.3: Schaltpläne der grundlegenden logischen Gattern

Transistoren sind somit ein wesentlicher Bestandteil der digitalen Verarbeitung. Um die Leistung eines Rechners zu steigern, müssen mehr Transistoren verbaut werden. Das Moore 'sche Gesetz trifft Aussagen über diese Entwicklung: Es besagt, dass sich die Anzahl der Transistoren pro Flächeneinheit alle 12 bis 24 Monate verdoppelt. Bisher konnte dies immer erfüllt werden, aber die Computerindustrie wird bald an physikalische Grenzen gelangen.

3.2.2 Funktionsweise des Quantencomputers

Bisher gibt es keine festen Aufbaustrukturen für Quantencomputer, da je nach Qubitart unterschiedliche Komponenten benötigt werden, ist die exakte Funktionsweise nicht zu verallgemeinern. Es gibt jedoch bei jedem Quantencomputer bestimmte Eigenschaften, die ‚unabhängig von der Qubitumsetzung, gegeben sein müssen, um die Definition eines Quantenrechners zu Erfüllen.
Hierzu gehören das zu Nutze machen von Quantenmechanischen Effekten zur Berechnung von Problemen. Dabei ist der Kern des Quantencomputers der Qubit. Ein Qubit besitzt sowohl zwei Basiszustände, als auch unendlich viele Mischzustände. Aus dieser und weiteren Eigenschaften wie Superposition oder Verschränkung mehrerer Qubits ergibt sich eine

bestimmte Verhaltensweise eines von Qubits [5]. Ein Quantencomputer ist ein System, in welchem sich mehrere Qubits befinden. Der Quantencomputer kann die Zustände dieser Qubits verändern und messen. Die Veränderung der Zustände wird mit Quantengattern[6], dem quantenmechanischem Äquivalent zu herkömmlichen logischen Gattern realisiert. Ein Quantencomputer benötigt weiterhin spezielle, für quantenmechanische Berechnungen geeignete, Arbeitsanweisungen, da ein Quantenrechner mit den herkömmlichen Programmen nicht sinnvoll und schnell arbeiten kann, weil diese Anweisungen keine Superposition und Verschränkung zulassen würden. Die sogenannten Quantenalgorithmen, wie beispielsweise der Grover-Algorithmus[7], sind solche speziellen Arbeitsschrittfolgen für Quantenrechner. Diese Algorithmen arbeiten auf Quantencomputern erheblich schneller,als ein herkömmlicher Computer mit einem klassischen, binär arbeitenden Algorithmus. Ein Quantencomputer arbeitet also im Grunde ähnlich wie ein klassischer Rechner, da auch er eine Datenstruktur zur Darstellung von Zuständen besitzt, mit Hilfe von Gattern diese Datenstrukturen beeinflusst und ein Ergebnis in Form eines gemessenen Zustandes zurückgibt. Diese Ähnlichkeiten resultieren aus der Definition einer Rechenmaschine und werden im nächsten Abschnitt näher betrachtet.

3.2.3 Vergleich der beiden Systeme

Der größte Unterschied zwischen Computer und Quantenrechner liegt in der Umsetzung der Bits. Während der normale Bit nur 2 feste Zustände annehmen kann, kann der Qubit unendlich viele Zustände zwischen 1 und 0 annehmen.Dies aus dem Phänomen der Superposition [8] resultieren. Der Quantenrechner arbeitet mit Wahrscheinlichkeiten die erst der Gatter-Schaltung durch Messung zu einem konkreten Zustand werden. Der normale Computer hingegen arbeitet zu jedem Zeitpunkt mit konkreten Zuständen und Werten. Die nächste Verschiedenheit ist die Umsetzung der Gatter: Beim normalem Computer hat man einige Grundgatter die gekoppelt andere Grundgatter ergeben. Beim Quantencomputer gibt es ausschließlich spezialisierte Gatter die nicht gleichzeitig und gekoppelt angewendet werden können, sondern nur nacheinander [9]. Des Weiteren gibt es für den Quantencomputer keinen standardisierten Bauplan, da der Aufbau abhängig von der Umsetzung der Qubits ist [10]. Da herkömmlicher Computer und Quantencomputer unterschiedlich arbeiten und sich die Funktionsweise so enorm unterscheidet ist es schwierig weitere geeignete Vergleichskriterien aufzuzeigen. Als Gemeinsamkeit kann jedoch die Aufgabe der beiden Systeme genannt werden. Sowohl Computer als auch Quantenrechner dienen der digitalen Datenverarbeitung.

[5] siehe Kapitel 2: Qubits
[6] siehe Kapitel 2.3: Quantengatter
[7] siehe Kapitel 3.4.1: Nutzen für Wissenschaften
[8] siehe Kapitel 2.2: Grundsätzliches Verhalten von Qubits
[9] siehe Kapitel 2.3: Quantengatter
[10] siehe Kapitel 2.1.1: Theoretische Qubitarten

3.3 Heutiger Forschungsstand

Wie in vorangegangenen Abschnitten bereits erwähnt, befindet sich die Forschung am Quantencomputer noch in der Grundlagenforschung. Bisher ist es nicht möglich einen Quantencomputer außerhalb zu bauen, weil sich die Qubits bisher schlecht manipulieren lassen, da die Qubits sehr stark von äußeren Störungen beeinflusst werden. Diese Probleme müssen noch gelöst werden ehe man den Quantencomputer kommerziell nutzen kann. Des Weiteren gibt es noch nicht die perfekte Art Qubits zu realisieren. Viele Verfahren sind aufgrund mangelnder Manipulierbarkeit oder Flüchtigkeit der Qubits ungeeignet, um diese quantenmechanische Zweizustandssysteme in einem Quantencomputer zu verwenden. Es wird wohl noch einige Jahrzehnte dauern bis sich der Quantencomputer etabliert.
Jedoch behaupten einige Computerhersteller bald einen marktfähigen Quantenrechner präsentieren zu können. So behauptete D-Wave Systems schon 2013 einen echten Quantencomputer gebaut zu haben. Jedoch ließ der Beweis bis 2015 auf sich warten, aber selbst zu diesem Zeitpunkt wurde nur ein spezieller Testfall präsentiert. Daher sind viele Experten auf diesem Gebiet nicht überzeugt. Der deutsche Physiker Matthias Troyer hatte die Chance den D-Wave zu testen (Endfassung näher betrachten) und konnte bestätigen, dass dieser Quantenmechanische Effekte zum Rechnen nutzt. Jedoch konnte er wiederlegen, dass der D-Wave einen erheblichen Geschwindigkeitsvorteil gegenüber herkömmlichen Superrechnern aufweist.[11] Aber auch der Computerhersteller IBM ist der Meinung einen Quantencomputer zur kommerziellen Nutzung auf den Markt bringen zu können und hat daraufhin die „Quantum Experience Platform“, einen in der IBM-Cloud betriebenen Quantenprozessor, der Öffentlichkeit zugänglich gemacht. Jeder hat die Möglichkeit, von zu Hause aus, eigene Experimente mit dem Quantenprozessor durchzuführen. So kann der Nutzer verschiedene Quantengatter auf 5 Qubits anwenden und am Ende sein Resultat betrachten.[12]

3.4 Nutzungsmöglichkeiten des Quantencomputers

Wird der Quantencomputer einen ebenso großen Nutzen haben und unser alltägliches Leben so verändern, wie es der digitale Universalrechner im 20. Jahrhundert tat? Oder ist der Quantenrechner nur für einige Spezialanwendungen geeignet? Viele Unternehmen und Institute halten sehr viel von der Idee des Quantenrechners und investieren viel Zeit und Geld in die Forschung. Doch mit welchem tatsächlichen Nutzen ist zu rechnen? Viele Organisationen, Institute und Unternehmen sind am Quantencomputer, aufgrund des theoretischen Geschwindigkeitsvorteils bei der Berechnung von Problemen, interessiert und investieren viel Geld in die Forschung.

[11] Meier, März 2015 [18]
[12] IBM, o. J. [19]

3.4.1 Nutzen für Wissenschaften

Shor-Algorithmus

Der Shor-Algorithmus wurde 1994 von Peter Shor veröffentlicht und entstammt dem mathematischen Teilgebiet der Zahlentheorie und nutzt Effekte der Quantenmechanik. Der Algorithmus dient der Berechnung, nichttrivialer Teiler von Zahlen. Damit zählt der Shor-Algorithmus zu den Faktorisierungsverfahren.[13] Der Shor-Algorithmus wurde zwar schon mit einem Quantencomputer geteste, jedoch ist der Shor-Algorithmus bisher nicht sinnvoll anwendbar, weil er massiven Technischen Einschränkungen unterliegt. Das liegt daran, dass man für eine Zahl **n** mindestens **log n** Qubits benötigt.[14]

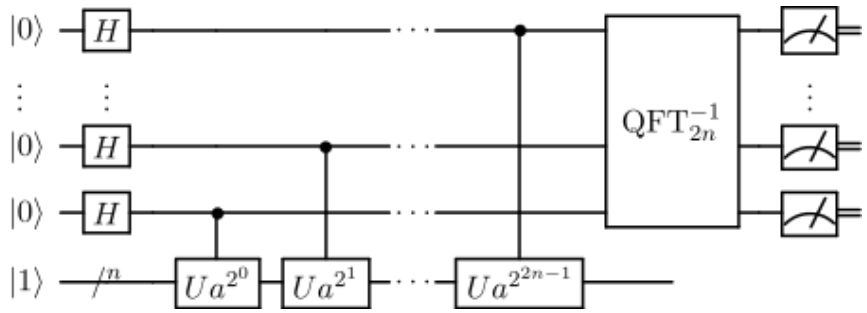


Abbildung 3.4: Schematische Darstellung des Shor-Algorithmus

Grover-Algorithmus

Der Groover-Algorithmus wurde 1996 von Lov Grover veröffentlicht und ist bisher der einzige Algorithmus der auf einem Quantencomputer schneller läuft als auf einem klassischen Computer. Dieser Algorithmus dient der Suche in einer unsortierten Datenbank mit **N** Elementen in

$$O\left(\sqrt{N}\right)$$

und mit einem Speicherbedarf von:

$$O\left(logN\right)$$

Wie alle Quantenmechanischen Algorithmen liefert der Grover-Algorithmus nur mit einer gewissen Wahrscheinlichkeit das richtige Ergebnis. Diese Wahrscheinlichkeit steigt mit zunehmenden N, sodass sich eine Wahrscheinlichkeit von 1 ergibt wenn N ins unendliche geht. Aus dem ‚in den vorangegangenen Abschnitten erläuterten Verhalten[15], geht die Arbeitsweise des Grover-Algorithmus hervor. Weiterhin werden einige Operationen benötigt: Das Hadamard-Gatter[16], das Orakel und den Rotationsoperator. Das Orakel und der Rotationsoperator erzeugen eine Phasenverschiebung.

[13] Paul, Zoppke, 2002 [20]
[14] Truhn, o. J. [21]
[15] siehe Kapitel 2.2: Grundsätzliches Verhalten von Qubits
[16] siehe Kapitel 2.3.3: Hadamardgatter

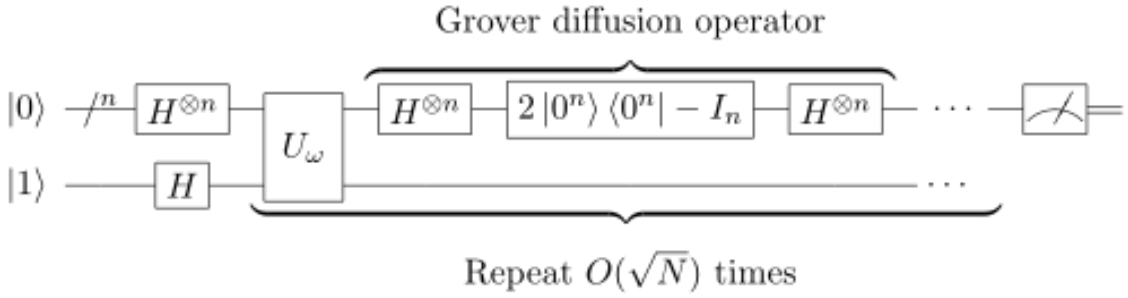


Abbildung 3.5: Schematische Darstellung des Grover-Algorithmus

Der Grover-Algorithmus arbeitet dann nach folgendem Ablaufplan: Der Anfangszustand wird mit —0¿ präpariert, dann wird durch die Hadamard-Transformation eine gleichmäßige Superposition erzeugt. Daraufhin folgt eine iterative Schleife von Operationen: Die Anwendung des Orakels, eine Hadamard-Transformation, eine Rotation, eine weitere Hadamard-Anwendung und das Erhöhen der von i, welches Anfang 0 ist. Nach dem Durchlauf der Schleife erhält man das Ergebnis als Wahrscheinlichkeit[17].

3.4.2 Nutzen für Geheimdienste

Der US-amerikanische Geheimdienst NSA ist besonders interessiert an einem Quantencomputer und investiert aus diesem Grund viele Millionen Dollar in ein geheimes Forschungsprojekt. Ziel des Projekts „Durchbrechen harter Ziele“ ist es, jene Verschlüsselungsverfahren auszuhebeln, durch die die meisten Daten im Internet gesichert sind.[18] Einige, inzwischen veraltete Verfahren konnten mit dem Fortschritt der Entwicklung des Computers und der zunehmenden Leistung bereits entschlüsselt werden, jedoch gibt es immer bessere Verfahren die ein herkömmlicher Computer nicht mehr durchbrechen kann. Das Verschlüsselungsverfahren RSA, welches nach den Erfindern Rivest, Shamir und Adleman benannt ist, ist für einen herkömmlichen Computer nicht in akzeptabler Zeit zu knacken. RSA arbeitet mit zwei Schlüsseln. Zum einen mit dem öffentlichen Schlüssel, welcher dazu dient die Nachrichten zu Verschlüsseln und andererseits dem privatem Schlüssel, welcher geheim gehalten wird, der dazu dient die Daten zu entschlüsseln. Es ist zwar möglich die ursprüngliche Information der verschlüsselten Bits zu berechnen, indem man den privaten Schlüssel aus dem öffentlichen Schlüssel ermittelt, aber das würde viel zu viel Zeit brauchen. Die NSA setzt darauf, dass der Quantencomputer so viel schneller als ein herkömmlicher Superrechner arbeitet, so dass Daten die durch Verfahren wie RSA verschlüsselt wurden, in kürzester Zeit entschlüsselt werden können und die NSA so noch mehr Menschen überwachen kann.

[17] Schubotz, 2008 [22]
[18] Meier, März 2015 [18]

3.4.3 Nutzen für das alltägliche Leben der Menschen

Neben den Geheimdiensten und Forschungsinstituten sind auch andere große Unternehmen an einem Quantencomputer interessiert. Die NASA beispielsweise möchte den Quantencomputer nutzen, um Bilder des Weltraumteleskops Kepler nach bisher nicht entdeckten Planeten zu durchsuchen. Weiterhin soll der Quantencomputer die Planung von Raumfahrtmissionen erleichtern und den herkömmlichen Rechner ablösen. Diese benötigen manchmal Wochen um bei Missionen das optimale Ressourcenmanagement zu berechnen. Die NASA setzt große Erwartungen in den Quantencomputer, der sehr schnell und zuverlässig sogenannte Optimierungsprobleme lösen soll.[19] Aber auch Google hat das Ziel, die herkömmlichen Supercomputer in den Großrechenzentren durch Quantenrechner zu ersetzen und so die Internetsuche zu beschleunigen und weiter zu verbessern.

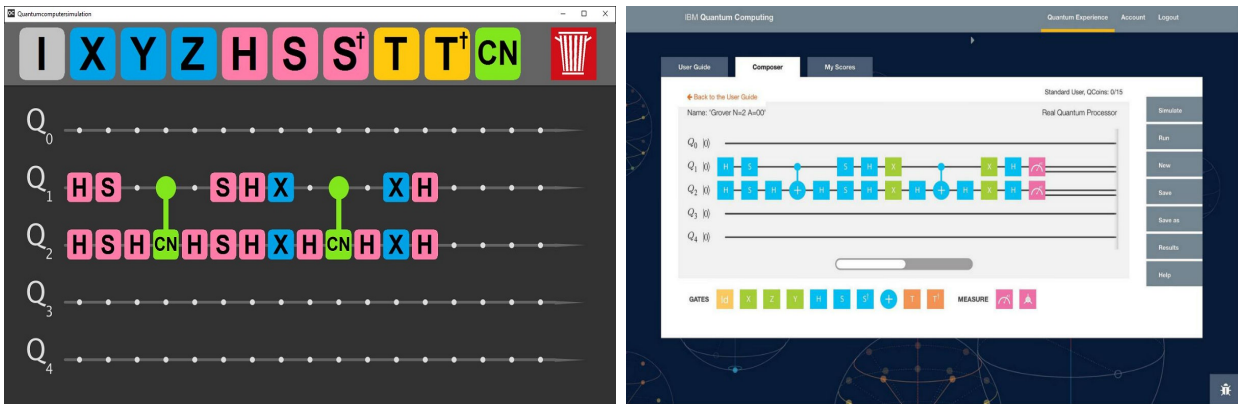
[19] Meier, März 2015 [18]

Kapitel 4

Quantencomputersimulation

4.1 Grundidee

Da jede bisherige Auseinandersetzung mit dem Konzept des Quantencomputers sehr theoretisch verlief, wurde die Simulation als praktischer Überblick über gegenwärtige und Ausblick auf künftige Fortschritte der Quantencomputerforschung verwendet. Die Inspiration für die Umsetzung dieses Zieles entstand durch das „Quantum-Experience“ Projekt des Unternehmens IBM[1]. Als eines der gegenwärtig führenden IT-Unternehmen investiert und forscht es in jede Richtung der Computertechnologie und ist unter anderem durch die Watson KI bekannt. 2016 stellte IBM ein weiteres Forschungsprojekt online: einen Quantencomputer. Jeder Nutzer kann über einen sogenannten Composer Gatter auf einen Gesamtqubitzustand anwenden. Der Composer ist hierbei die Schnittstelle zwischen dem Nutzer und dem Algorithmus und damit ein Teil der graphischen Nutzeroberfläche. Man besitzt dabei zwei Möglichkeiten zur Interaktion. Entweder man arbeitet mit einem echten Gesamtqubitzustand, welcher bei ca. 0.015 Kelvin parallel im „Thomas J. Watson Research Center“ erzeugt wird, oder man entscheidet sich für die Arbeit mit einer rein durch Software definierten Simulation.



(a) Simulationsprogramm dieser Seminarfacharbeit (b) IBM Quantum Experience Composer[1]

Abbildung 4.1: Graphische Benutzeroberflächen beider Quantencomputersimulationen

[1]IBM Q Experience, 2016 [23]

Der Computer ist in der Größe des Projektes, der Kommunikation zwischen Qubits und der Wiedergabe eines vollständig richtigen Ergebnisses aufgrund von Ungenauigkeiten und Grenzen der physikalischen Welt eingeschränkt, während die Simulation mathematisch richtige – und damit rein theoretische – Ergebnisse liefert und IBM's Quantencomputer nicht benötigt. Es war ein Ziel der Arbeit, eine eigene Simulation zu entwerfen und zu implementieren sowie die GUI in den Hauptpunkten wie Gatterdarstellung und allgemeiner Verwendung möglichst nah an IBM's „Quantum-Experience“ anzulehnen wie in der vorherigen Abbildung sieht. Dabei wurde kein Quelltext übernommen, denn der benutzte Code und die verwendeten Bilder des eigenen Programmes sind alle selbst erstellt. Die starken Ähnlichkeiten der Layouts sind auch aufgrund von Notationskonventionen nötig, da besondere Gatter spezielle Namen und Abkürzungen besitzen, welche benutzt werden sollten, um die GUI allgemein verständlich zu halten. Ausserdem ist die Darstellung von Gattern auf einem sogenannten „Wirediagramm“ allgemein üblich. Ein Wirediagramm teilt die möglichen Gatterplatzierungen auf N Qubits und M Zeitabschnitte auf. Die Gatter im Wirediagramm werden dann nacheinander auf die korrespondierenden Qubits angewendet. In den obigen Bildern wirken also zuerst zwei Hadamard-Gatter auf Qubit eins und zwei, dann zwei S-Gatter und so weiter. Für das Programmieren des Quantencomputersimulators wurde aufgrund des umfassenden Vorwissens durch den Schullehrplan die Programmierbasissprache Java[2] verwendet. Processing[3], eine quelloffene (open source) Entwicklungsumgebung und Programmiersprache mit Schwerpunkten auf Animation und Grafik, wurde zusätzlich für einen einfacheren Visualisierungscode eingebunden.

4.2 Programmstruktur und Zustandsberechnung

Java ist eine objektorientierte Programmiersprache, weswegen Objektorientierung zur Übersichtlichkeit und Funktionstüchtigkeit von in Java geschriebenen Programmen ab gewissen Größen von elementarer Bedeutung ist. Objektorientierte Programmierung baut ein Programm als Zusammenspiel einzelner Objekte auf, welche eigene Daten (Attribute) und einen ausführbaren Code (Methoden) besitzen und durch Klassen mithilfe spezieller Ausgangsvariablen instanziiert werden. Nimmt man Qubits als Beispiel, dann stellt der Vorgang der Erstellung eines neuen Qubits die abstrakte Klasse dar. Möchte man nun ein neues Qubit, also ein Objekt der Klasse, erzeugen, dann gibt man als Ausgangsvariablen den Startzustand des Qubits an, damit durch den Erstellungsvorgang ein neues Qubit erzeugt werden kann. Der Vorgang ist jedoch nicht nur einmalig nutzbar, da durch neue Ausgangsvariablen weiterhin neue Qubits erzeugt werden können und parallel existieren. Betrachtet man nun die gesamten Ausgangsqubits, dann besitzen sie eine festgelegte Anzahl und einen festgelegten Zustand und können mit Gattern interagieren. Dies sind die Attribute und Methoden des Objektes „Qubits“. Außerdem besteht es aus den einzelnen Qubits, welche wiederum als Objekte mit eigenen Attributen und Methoden fungieren, die als einzelne Attribute der Qubitgesamtheit gelten und darin gebunden sind. Die Simulation funktioniert auf diese Weise mit 7 teilweise ineinander verschachtelten Klassen, welche durch das unten aufgeführte Klassendiagramm modelliert werden.

[2]Oracle, Java SE Development Kit 8, 2014 [24]
[3]Ben Fry, Casey Reas, Processing 3.0, 2015 [25]

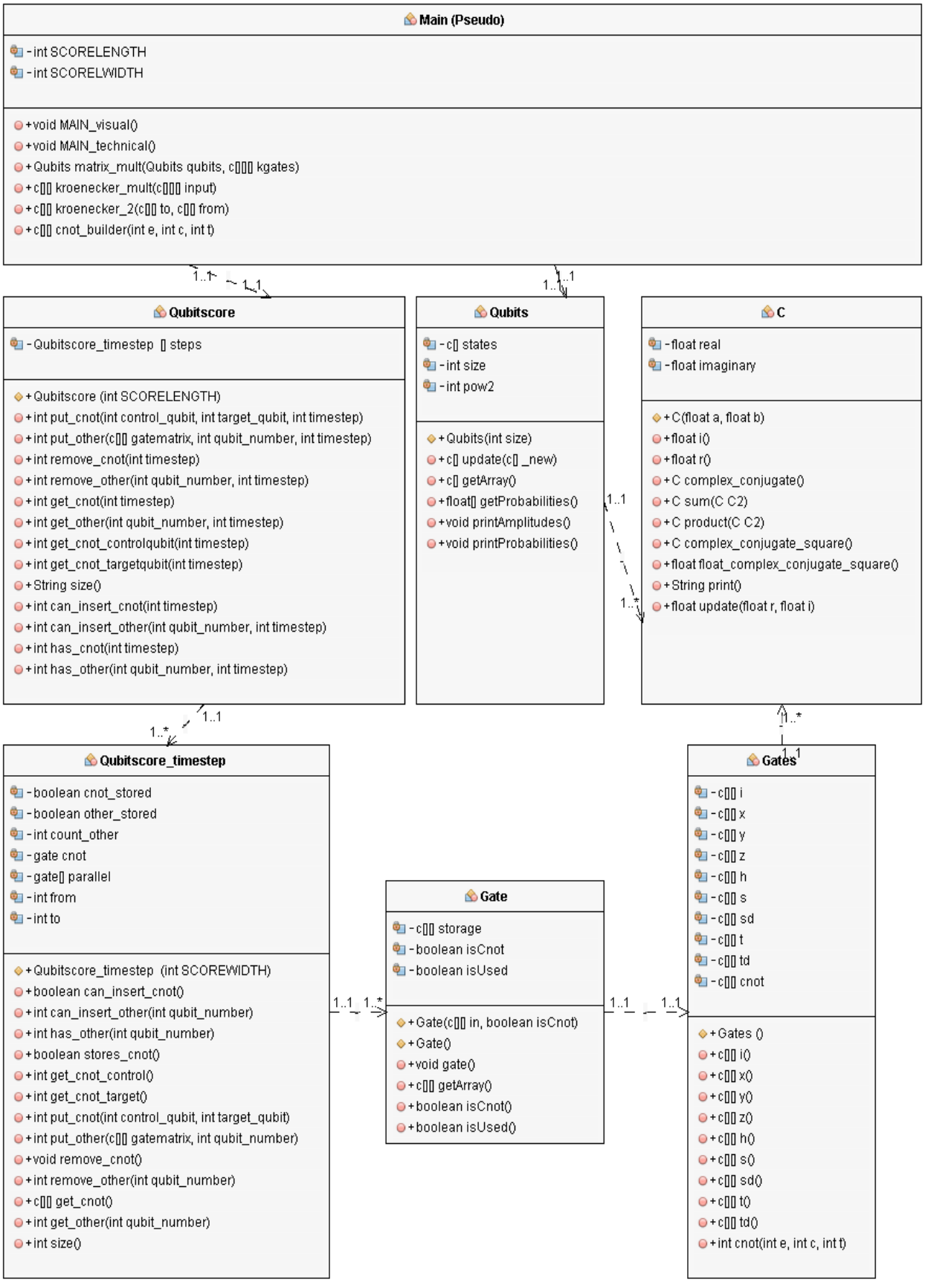


Abbildung 4.2: Klassendiagramm der Quantencomputersimulation

4.2.1 C

Die „C“-Klasse ist als Darstellung von komplexen Zahlen die Basis der Quantencomputersimulation, da Qubits universell nur durch 2 komplexe Zahlen modelliert werden können. Diese sind in Java nicht standardmäßig implementiert, weswegen selbst eine Klasse programmiert werden musste. Jedes Objekt der Klasse besitzt als Attribute je eine Fließkommazahl zur Beschreibung des Realteils r und des Imaginärteils i, um eine komplexe Zahl der Form $r + i\sqrt{-1}$ zu simulieren. Berechnungen können durch bekannte mathematische Gleichungen im komplexen Raum durchgeführt werden[4].

$$\Re(r + i\sqrt{-1}) = r \qquad \textit{Realteil} \qquad (4.1)$$

$$\Im(r + i\sqrt{-1}) = i \qquad \textit{Imaginärteil} \qquad (4.2)$$

$$(a + b\sqrt{-1}) + (r + i\sqrt{-1}) = ((a + r) + (b + i)\sqrt{-1}) \qquad \textit{Summe} \qquad (4.3)$$

$$(a + b\sqrt{-1}) \cdot (r + i\sqrt{-1}) = ((ar - bi) + (ai + br)\sqrt{-1}) \qquad \textit{Produkt} \qquad (4.4)$$

$$\overline{r + i\sqrt{-1}} = r - i\sqrt{-1} \qquad \textit{komplexes Konjugat} \qquad (4.5)$$

$$|r + i\sqrt{-1}|^2 = (r^2 + i^2) + 0\sqrt{-1} = r^2 + i^2 \qquad \textit{Absolutes Quadrat} \qquad (4.6)$$

$$r := a,\ i := b,\ r + i\sqrt{-1} = a + b\sqrt{-1} \qquad \textit{Zuweisung} \qquad (4.7)$$

Hierbei ist $\sqrt{-1}$ als imaginäre Einheit für die Implementation nicht nötig, da keine komplexen Zahlen verwendet, sondern nur via Neuzuweisungen oder Ausgaben von Attributen des „C“-Objektes durch Methoden simuliert werden.

$$(r, i).r := r \qquad (r, i).i := i \qquad (4.8)$$

$$(r, i).c\_conjugate := (r, -i) \qquad (r, i).update(a, b) := (a, b) \qquad (4.9)$$

$$(r, i).sum(a, b) := (a + r, b + i) \qquad (r, i).product(a, b) := (ar - bi, ai + br) \qquad (4.10)$$

$$(r, i).cc\_square := (r^2 + i^2, 0) \qquad (r, i).float\_cc\_square := r^2 + i^2 \qquad (4.11)$$

4.2.2 Gates

Die „Gates“-Klasse ist als Sammel-Speicher der einzelnen Gatter die Basis der „Gate“-Klasse, da sie die einzelnen Gatter frei zur Verfügung stellt. Hierbei benutzt sie „C“-Objekte, da jedes Gatter durch komplexe Zahlen beschrieben wird[5], und speichert jedes dieser Gatter als zweidimensionales Array.

$$\text{Bsp: } \mathbf{S} = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} = \begin{bmatrix} 1 + 0i & 0 + 0i \\ 0 + 0i & 0 + i \end{bmatrix} \Longrightarrow \left[\Big[(1, 0), (0, 0) \Big], \Big[(0, 0), (0, 1) \Big] \right] \qquad (4.12)$$

Das C-Not Gatter nimmt wieder eine Sonderrolle ein, da es nicht vorher definierbar ist, sondern durch 3 Parameter bestimmt wird, welche sich bei jedem C-Not Gatter erst während der Platzierung ergeben. Deswegen wird es durch eine eigene Methode neu generiert, welche wegen ihrer Komplexität in die Main ausgelagert wurde.

[4] Weisstein, 2003, S.1 [26]

[5] siehe Kapitel 2.3 Quantengatter

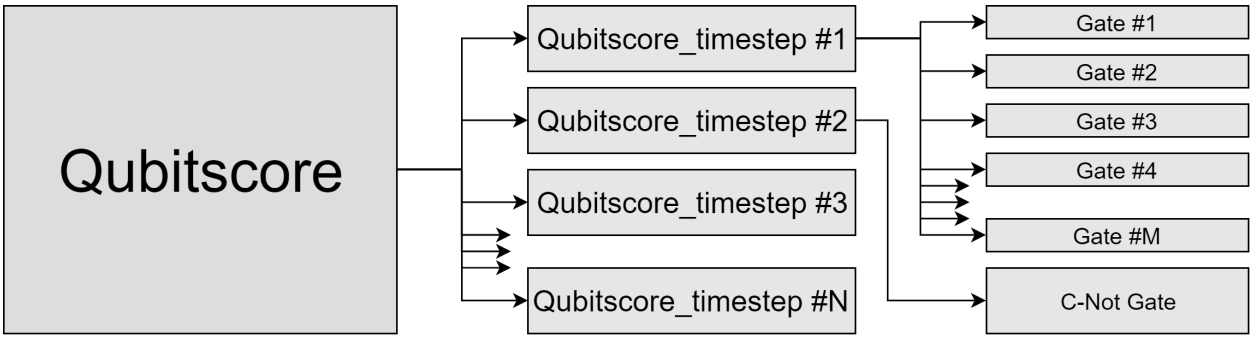


Abbildung 4.3: Hierarchie des Qubitscores

4.2.3 Gate

Die „Gate“-Klasse speichert genau ein Gatter aus der „Gates“-Klasse und kann dieses Gatter ausgeben. Es unterscheidet zusätzlich noch zwischen 2 Kategorien von Gattern, dem C-Not Gatter und allen anderen Gattern. Dies geschieht, da diese beiden Arten von Gattern unterschiedliche Größen aufweisen. Das C-Not Gatter besteht aus Arrays der Länge 2^N, während alle anderen Gatter aus Arrays der Länge 2 aufgebaut sind. Man benötigt also N normale Gatter, um ein Gatter derselben Größe zu erreichen, weswegen das C-Not Gatter extra betrachtet werden muss.

4.2.4 Qubitscore_timestep & Qubitscore

Jedes Objekt der „Qubitscore_timestep“-Klasse stellt einen Zeitabschnitt im Wirediagramm dar. Es unterteilt sich in N Gatter, welche jeweils mit einem Qubit korrespondieren. Während der Ausgabeberechnung führt es eine Multiplikation mit dem Objekt der „Qubits“-Klasse aus. Um dies umzusetzen, benötigt es eine Matrix P der Größe 2^N und muss deswegen entweder ein C-Not Gatter Q_{C-Not} oder bis zu N andere Gatter Q_x enthalten, um nach einer Kroneckermultiplikation die richtige Größe aufzuweisen.

$$P = \begin{cases} Q_{C-Not} & \\ \prod_{n=1}^{N} Q_n & \Big| \; Q_n = \begin{cases} Q_n & \textit{for P.has\_other(n)==true} \\ \mathbf{I} & \textit{for P.has\_other(n)==false} \end{cases} \\ \text{Kronecker} & \end{cases} \quad \textit{for P.cnot\_stored()==true} \tag{4.13}$$

Das Objekt der „Qubitscore“-Klasse stellt das gesamte Wirediagramm dar, ist hauptsächlich zum Zusammenfassen der einzelnen Qubitscore_timesteps gedacht und leitet „Getter“ und „Setter“[6] des Codes zum Erstellen der visuellen Repräsentation auf diese weiter wie man in Abbildung 4.3 sehen kann.

4.2.5 Qubits

Die „Qubits“-Klasse verwendet 2^N komplexe Objekte, um die N benötigten Qubits mathematisch korrekt darzustellen. Diese Liste stellt das Kroneckerprodukt aus allen einzel-

[6]Methoden zum Entgegennehmen und Verändern von Objektattributen

nen Qubitmatrizen dar und enthält je einen Eintrag für das Produkt der Elemente jeder möglichen Permutation von N der $2N$ Elemente α_1 bis α_N und β_1 bis β_N.[7] Beginnt die Berechnung des vom Nutzer erstellten Programms, dann wird nacheinander jedes der M Gesamtgatter mit dem Kroneckerprodukt der Qubits, also dem Gesamtqubitzustand multipliziert, um am Ende die Ausgabedaten zu erhalten. Beide Arten der Matrixmultiplikation wurden allerdings aufgrund ihrer Komplexität nicht objektintern geregelt, sondern wieder in die Main als Methoden ausgelagert. Das „Qubits"-Objekt ist die passive Eingabe, auf die andere Objekte einwirken, wodurch es schließlich zur Ausgabe wird. Danach werden die einzelnen Wahrscheinlichkeiten als absolutes Quadrat der einzelnen Teilsuperpositionszustände an die Main übergeben und optional sowohl die Amplituden als auch die Wahrscheinlichkeiten ebendieser auf der Konsole ausgegeben.

4.2.6 Main

Ein Teil der „Main"-Klasse stellt die Übersicht über alle anderen Objekte jeder Klasse dar, regelt den groben Ablauf des Programmes und sorgt für die Berechnung des Ausgabesuperpositionszustandes. Der andere Teil beherbergt die ausgelagerten Methoden.

```
float[] state_calculator() {
  // Dem Qubitscore werden alle Gatterpositionen und Gatterarten
     uebergeben.
  Qubitscore QS = Gatterspeicher.getData();
  // Das Kroneckerprodukt der einzelnen Qubitscore_timesteps wird
     berechnet.
  c[][][] after_kronecker = new c[M[(int) pow(2, N)][(int) pow(2, N)];
  for (int i=0; i<M; i++) {
        // Spezialfall: C-Not Gatter
        // In diesem Fall kann man einfach kopieren.
    if (QS.has_cnot(i) == true) {
      after_kronecker[i] = QS.get_cnot(i);
    }
    // Normalfall: gewöhnliche Gatter
    // In diesem Fall muss man das Kroneckerprodukt berechen.
    else {
      c[][][] normal_gates = new c[N][][];
      for (int j=0; j<normal_gates.length; j++) {
        normal_gates[j] = QS.get_other(j, i);
      }
      after_kronecker[i] = kronecker_mult(normal_gates);
    }
  }
  // Das Matrixprodukt aus Qubits und Qubitscore_timesteps wird
     berechnet.
  Qubits after_product = new Qubits(SCOREWIDTH);
  after_product = matrix_mult(qubits, after_kronecker);
  // Die Endwahrscheinlichkeiten werden zurückgegeben.
  return after_product.getProbabilities();
}
```

[7] siehe Kapitel 2.2.1 Mathematische Grundlagen

Kroneckerprodukt

Die Methode zum Berechnen des Kroneckerprodukts ist für ihre Aufgabe zweigeteilt. Der erste Teil empfängt eine Liste von Matrizen, teilt diese auf und übergibt sie nacheinander immer wieder dem zweiten Teil. Dieser iteriert dann über beide Arrays und schreibt die Produkte der einzelnen Arrayelemente in das Ausgabearray. Das Ausgabearray wird durch den ersten Teil mit einer weiteren Matrix neu übergeben, bis alle vorhandenen Matrizen genutzt wurden.

```
// Kroneckerprodukt von 2 Matrizen
c[][] kronecker_2(c[][] to, c[][] from) {
  // Instanziierung des Ausgabearrays
  c[][] output = new c[to.length*from.length]
      [to[0].length*from[0].length];
  // Iteration über Array "to"
  for (int i=0; i<to.length; i++){
  for (int j=0; j<to[0].length; j++){
    // Iteration über Array "from"
    for (int k=0; k<from.length; k++){
    for (int l=0; l<from[0].length; l++){
      // Berechnung der Koordinaten des
          einzelnen Ergebnisses im
          Ausgabearray dortiges Eintragen
      int y = i*from.length + k;
      int x = j*from[0].length + l;
      out[y][x]=to[i][j].product(from[k][l]);
  }}}}
  // Ausgabe des Ausgabearrays
  return out;
}
```

```
// Kroneckerprodukt von n Matrizen
c[][] kronecker_mult(c[][][] input) {
  // Sonderfall: 2 Matrizen
  if (input.length==2) {
    // Matrizenübergabe an kronecker_2
        Methode und Rückgabe des Ergebnisses
    return kronecker_2(input[0], input[1]);
  // Normalfall: mehrere Matrizen
  } else {
    // Sequenzielle Matrizenübergabe an
        kronecker_2 Methode und
        Wiederverwendung der Rückgabe
    c[][] first = input[0];
    c[][][] rest = new c[input.length-1][][];
    for (int i=1; i<input.length; i++) {
      rest[i-1] = input[i];
    }
    // Ausgabe des Endergebnisses
    return kronecker_2(first, kronecker_mult(
        rest));
}}
```

Matrixprodukt

Die Methode zum Berechnen des Matrixproduktes funktioniert analog zu der des Kroneckerproduktes. Sie erstellt das Ausgabearray jedoch basierend auf den Größen der Eingabearrays und bildet für jedes Element die Summe aus den einzelnen benötigten Produkten, um sie in das Ausgabearray zu schreiben.

```
//Matrixprodukt von 2 Matrizen
Qubits matrix_mult_2(Qubits qb, c[][] kgate) {
  // Instanziierung der zu multiplizierenden Matrizen
  c[] q = qb.getArray();
  c[] out = new c[q.length];
  // Iteration über das Ausgabearray und Summierung der
       einzelnen Produkte
  for (int m=0; m<kgate.length; m++) {
    c partSum = new c(0, 0);
    for (int n=0; n<kgate[0].length; n++) {
      partSum = partSum.sum(q[n].product(kgate[m][n]));
    }
    // Eintragen der jeweiligen Produktsummen
    out[m] = partSum;
  }
  // Ausgabe des upgedateten "Qubits" Objekts
  qb.update(out);
  return qb;
}
```

```
// Matrixprodukt von n Matrizen
Qubits matrix_mult
    (Qubits qb, c[][][] kgates) {
  // Sequenzielle Matrizenübergabe
      an matrix_mult_2 Methode und
      Wiederverwendung der Rückgabe
  for(int i=0;i<kgates.length;i++){
    qb=matrix_mult_2(qb,kgates[i]);
  }
  // Ausgabe des "Qubits" Objektes
      nach Matrixmultiplikation mit
      jedem Kroneckergatter
  return qb;
}
```

C-Not Gatterberechnung

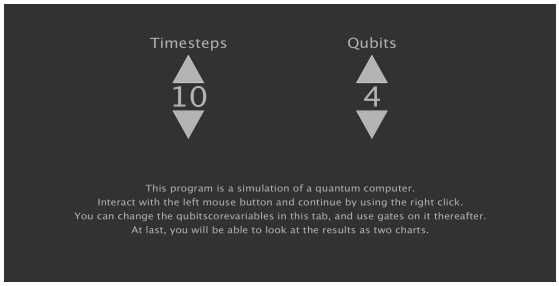
Die Methode zum Erstellen des C-Not Gatters kann durch einen Vergleichsalgorithmus von einer normalen Qubitzustandsliste und dieser nach Anwendung der Veränderung durch Kontrollqubit und Zielqubit das gesuchte Gatter bestimmen, indem sie untersucht, welcher Zustand wohin übergeht.

```
// Pseudocode
c[][] cnot_builder(N, Kontrollqubit, Zielqubit) {
  // Erstellen zweier Arrays mit Zustaenden abhaengig vom C-Not Gatter
  int[][] davor = new int[2^N][N];      // Zustaende davor
  int[][] danach = new int[2^N][N];     // Zustaende danach
  // Mit dem Stellenwertsystem und den binaeren Zahlen lassen sich Qubitzustandsarten
      darstellen: |0110> => 0110 => 6
  // Durch diese Vorgehensweise werden alle Qubitzustandsarten im "davor" Array
      aufgelistet
  for (zaehler = 0 bis 2^N) {
    int binaer_zaehler = binary(zaehler);
    for (j = 0 bis 2^N) {
      davor[i][j] = binaer_zaehler%10;
      binaer_zaehler = binaer_zaehler/10;
  }}
  // "danach" analog zu "davor", aber falls der Zustand des Kontrollqubits 1 ist, wird
      dar Zielqubitzustand invertiert
  for (i = 0 bis 2^N) {
    danach[i] = davor[i];
    if (davor[i][Kontrollqubit]==1) {
      if      (danach[i][Zielqubit]==1) danach[i][Zielqubit]=0;
      else if (danach[i][Zielqubit]==0) danach[i][Zielqubit]=1;
  }}
  // Vergleichsarray, welches parallel die Position jeder Qubitzustandsart im "davor"
      und "danach" Array speichert
  int[][] aenderung = new int[2^N][2];
  for (position_1,position_2 = 0 bis 2^N) {
      if (davor[position1] = danach[position_2]) {
        aenderung[position_1][0]=position_1;
        aenderung[position_1][1]=position_2;
  }}
  // Erstellung des Basisarrays für das C-Not Gatter der Größe 2^N
  c[][] output = new c[2^N][2^N];
  for (i,j = 0 bis 2^N) {
      output[i][j] = new c(0, 0);
  }
  // Einfügen der Einsen an den richtigen Stellen mithilfe des "aenderung" Arrays
  for (i = 0 bis 2^N) {
    output[ aenderung[i][0] ][ aenderung[i][1] ] = 1;
  }
  // Ausgabe des fertigen C-Not Gatters
  return output;
}
```

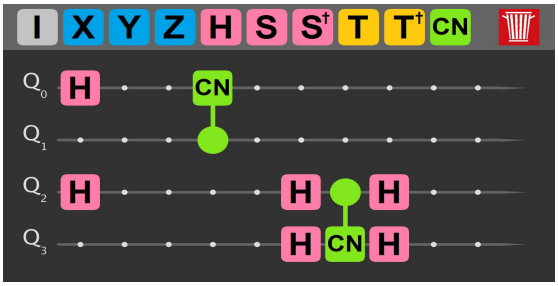
4.3 GUI

Die GUI stellt die Nutzeroberfläche der Quantencomputersimulation dar und sollte möglichst intuitiv zu bedienen sein. Aus diesem Grund wurde eine rein auf der Maus als Eingabegerät basierende Oberfläche implementiert, mit welcher man über Click oder Click & Drop interagieren kann. Das Interagieren erfolgt mit der linken Maustaste, das Bestätigen mit der rechten Maustaste und das Beenden über die Escape-Taste. Die GUI ist in zwei Teile geteilt: vor der Berechnung des Endsuperpositionszustandes mit Fokus auf Eingabe und Manipulation des Qubitscores und danach mit Fokus auf klarer Informationsdarstellung.

4.3.1 Eingabe



(a) Auswahl der Qubit- und Zeitabschnittsanzahl

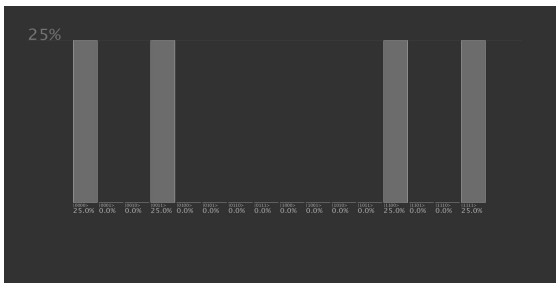


(b) Quantumscore Nutzeroberfläche

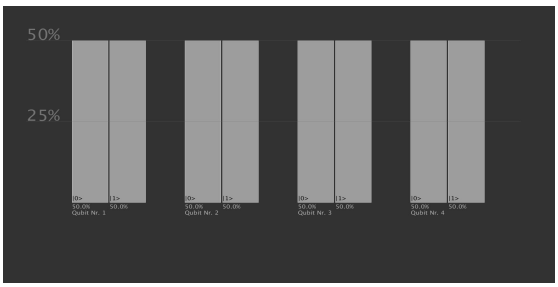
Abbildung 4.4: Graphische Benutzeroberflächen des Eingabeteils

Der Eingabeteil ist wiederum in zwei Unterkategorien gegliedert. Das erste Element ist der Auswahlbildschirm, auf dem man 1 bis 20 Zeitabschnitten und 2 bis 5 parallele Qubits einstellen kann. Wurde dies gemacht, dann wird das zweite Element angezeigt, ein Qubitscore mit der gewählten Anzahl an Zeitabschnitten und Qubits. Auf diesem kann man nun seine Gatter frei platzieren. Die einzige Einschränkung stellt die Regel dar, dass nur ein C-Not Gatter oder N andere Gatter gleichzeitig einen Zeitabschnitt belegen dürfen. Bestätigt man nun seine Eingabe, dann wird der Endsuperpositionszustand berechnet und man gelangt in den Ausgabeteil.

4.3.2 Ausgabe



(a) Wahrscheinlichkeit jedes Qubitzustandes



(b) Zustandswahrscheinlichkeit von jedem Qubit

Abbildung 4.5: Graphische Benutzeroberflächen des Ausgabeteils

Auch der nach der Berechnung angezeigte Ausgabeteil besteht aus zwei Unterkategorien: einem Diagramm zur Verdeutlichung der Wahrscheinlichkeit von jedem möglichen erreichbaren Qubitzustand und einem Diagramm zur Verdeutlichung der Zustandswahrscheinlichkeit von jedem Qubit. Wendet man also die oben gezeigten Gatter auf die Qubits an, so sieht man, dass nur $|0000\rangle, |0011\rangle, |1100\rangle, |1111\rangle$ als Zustände mit den gleichen Wahrscheinlichkeiten von 25% auftreten können. Außerdem hat jedes Qubit eine Wahrscheinlichkeit von 50%, selbst im Zustand $|0\rangle$ oder $|1\rangle$ aufzutreten.

Kapitel 5

Zusammenfassung und Ausblick

In der vorangegangen Arbeit sind die wichtigsten Fakten und Ideen zur Forschung an Quantencomputern als neue Generation von Rechnern erläutert worden. Es wurde gezeigt, wie man Qubits theoretisch verhalten, wie sie sich realisieren und beeinflussen lassen und welche Physikalische und technische Voraussetzungen gegeben sein müssen um mit den Qubits zu arbeiten. Das theoretische mathematische Verhalten wurde durch eine Simulation von einigen Qubits näher betrachtet und visualisiert dargestellt. Es wurden weiterhin Ideen zur Konstruktion und für den Bau eines Quantencomputers erklärt und die Unterschiede zum herkömmlichen Rechner aufgezeigt. Des Weiteren wurde über verschiedene Nutzungsmöglichkeiten nachgedacht und einige Probleme, wie der Shor- und der Grover- Algorithmus, erläutert, für die der Quantenrechner erwiesen besser geeignet ist, als normale Digitalrechner.
Doch wie im Kapitel über den heutigen Forschungsstand erläutert, steckt die Forschung am Quantencomputer noch in der Anfangsphase der Grundlagenforschung. Wie also könnte sich die Forschung weiter entwickeln und was sind dabei mögliche Ergebnisse?
Es ist schwierig Aussagen über eine Entwicklung einer Forschung zu treffen, da es in der Wissenschaft immer starke Sprünge und große technische Fortschritte von ungeahntem Ausmaß gibt. Man kann aber definitiv sagen, dass in den nächsten Jahren oder Jahrzehnten die Forschung am Quantencomputer weiter betrieben wird und Ergebnisse liefert, welche den Weg zum funktionierenden Quantencomputer bereiten.

Im Laufe unserer Seminarfacharbeit konnten wir viel über die, interessante Thematik des Quantencomputings lernen und es ist uns gelungen eine funktionierende Simulation eines Quantensystems zu entwickeln. In der Zeit des Seminarfachs konnten wir viele wichtige Erfahrungen, für die spätere Studienzeit und den damit einhergehenden wissenschaftlichen Arbeiten, sammeln.

Glossar

Amplitude: Die Amplitude ist ein Begriff aus der Mathematik und Physik zur Beschreibung von Schwingungen und steht für den maximalen Ausschlag der Funktion.

Gesamtgatter: Das Gesamtgatter ist das Ergebnis der Kroneckermultiplikation aller Gatter eines Zeitabschnittes. Sie wirken nacheinander auf die Gesamtheit der Qubits ein, um einen Ausgabezustand zu ereichen.

GUI: Die GUI ist das „graphical user interface“ oder zu deutsch die „graphische Benutzeroberfläche“. Sie dient zur visuellen Darstellung von Programminhalten und einer intuitiveren Interaktion des Benutzers mit dem Programm.

Komplexe Zahlen: Komplexe Zahlen sind der um imaginäre Einheiten erweiterte reelle Zahlenbereich. Eine imaginäre Einheiten ist als die Wurzel von -1 definiert.

RSA: Das RSA-Verfahren ist ein heutzutage übliches Verfahren zur Verschlüsselung von Daten.

Qubit: Der Qubit, oder Quantenbit, ist das quantenmechanische Äquivalent zum Bit. Ein Qubit dient der Verarbeitung von Daten und ist in der Lage Superpositionszustände und Verschränkungszustände anzunehmen.

Wirediagram: Das Wirediagram ist, in diesem Zusammenhang, ein Schaltbild zur Veranschaulichung der Position von Quantengattern.

Abbildungsverzeichnis

Literaturverzeichnis

[1] Terr, David. "Qubit."From MathWorld–A Wolfram Web Resource, created by Eric W. Weisstein.
http://mathworld.wolfram.com/Qubit.html, 14.04.2017

[2] Rui Zhang, Zhiteng Wang, Hongjun Zhang, Schematic diagram of the qubit probability amplitude, ResearchGate, Juli 2014,
https://www.researchgate.net/figure/287429217_fig3_Schematic-diagram-of-the-qubit-probability-amplitude, 17.04.2017

[3] Paola Cappellaro, Composites systems and Entanglement, MIT open courseware, 2012,
https://ocw.mit.edu/courses/nuclear-engineering/22-51-quantum-theory-of-radiation-interactions-fall-2012/lecture-notes/MIT22_51F12_Ch6.pdf 17.06.2017

[4] Research Center for Quantum Information, Institute of Physics, Slovak Academy of Sciences, Basic concepts in quantum computation, Quantiki, Januar 2016,
https://www.quantiki.org/wiki/basic-concepts-quantum-computation, 17.04.2017

[5] User:Bilou, Schematic depiction of the matrix product AB of two matrices A and B, Wikipedia, Oktober 2010,
https://en.wikipedia.org/wiki/Matrix_multiplication/media/File:Matrix_multiplication_diagram_2.svg, 16.04.2017

[6] Roland Wengenmayr, Superrechner für Spezialanwendungen, Zeit Online, Juli 2012,
http://www.zeit.de/digital/internet/2012-07/quantencomputer-medikamente-materialforschung, 20.04.2017

[7] R. Sexl, Materie in Raum und Zeit, LEIFI, ohne Jahr,
https://www.leifiphysik.de/quantenphysik/quantenobjekt-elektron/welle-teilchen-dualismus, 17.6.2017

[8] Werner Heisenbeg, LEIFI, ohne Jahr, https://www.leifiphysik.de/quantenphysik/quantenobjekt-elektron/unschaerferelation, 17.6.2017

[9] Tunneleffekt, milq, ohne Jahr, http://www.milq-physik.de/11._Tunneleffekt, 23.5.2017

[10] Mira Prior, Polarisation elektromagnetischer Wellen, LP, 2009, https://lp.uni-goettingen.de/get/text/5356, 16.6.2017

[11] D-Wave Systems, The Quantum Computer, D-Wave, ohne Jahr, https://www.dwavesys.com/d-wave-two-system, 23.5.2017

[12] Alexander S. Shumovsky, Valery I. Rupasov, Quantum Communication and Information Technologies, Springer Science & Business Media, 06.12.2012, goo.gl/zuBEmF, 14.04.2017

[13] D.Gross, S.T.Flammia, J.Eisert, Most Quantum States Are Too Entangled To Be Useful As Computational Resources, Phys. Rev. Lett., Mai 2009, https://journals.aps.org/prl/abstract/10.1103/PhysRevLett.102.190501, 20.04.2017

[14] Terr, David. "Bloch Sphere."From MathWorld–A Wolfram Web Resource, created by Eric W. Weisstein. http://mathworld.wolfram.com/BlochSphere.html, 05.05.2017

[15] Robert Schanze, Wie funktioniert ein Computer? – Erklärung für Laien & Profis, Giga, November 2016, http://www.giga.de/downloads/windows-10/specials/wie-funktioniert-ein-computer-erklaerung-fuer-laien-profis/, 20.4.2017

[16] Jan Wiener, John von Neumann und die Von-Neumann-Architektur,*/lehre, März 2016 http://www.lehre.jan-wieners.de/wisem15/bit-i/john-von-neumann-und-die-von-neumann-architektur/, 20.4.2017

[17] User: Jeff, Der Arbeitsspeicher (RAM oder PC-Speicher), CCM, November 2012, http://de.ccm.net/contents/273-der-arbeitsspeicher-ram-oder-pc-speicher, 20.4.2017

[18] Christian J. Meier, Die kurze Geschichte des Quantencomputers, 1. Auflage, Heise Zeitschriften Verlag, März 2015, 17.6.2017

[19] IBM, https://www.research.ibm.com/ibm-q/, 24.4.2017

[20] Christian Paul & Till Zoppke, Shor's Algorithm - Faktorisierung großer Zahlen mit einem Quantencomputer, FU-Berlin, 2002, http://page.mi.fu-berlin.de/alt/vorlesungen/sem02/shors-algorithm.pdf, 20.7.2017

[21] Daniel Truhn, Einführung in den Shor-Algorithmus, ohne Jahr, https://www.cond-mat.de/teaching/QCsem/truhnpaper.pdf, 20.7.2017

[22] Moritz Schubotz, Seminarvortrag zum Thema Grover-Algorithmus, 2008, https://www.itp.tu-berlin.de/fileadmin/a3233/upload/AG-Brandes/Seminar-SS-08/moritz.pdf, 15.7.2017

[23] IBM Q experience, Mai 2016, https://quantumexperience.ng.bluemix.net/qstage/, 06.05.2017

[24] Oracle, Java SE Development Kit 8, März 2014, http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html, 12.07.2017

[25] Ben Fry, Casey Reas, Processing 3.0, 2015,
https://processing.org/, 12.07.2017

[26] Weisstein, Eric W. "Complex Number."From MathWorld–A Wolfram Web Resource.
http://mathworld.wolfram.com/ComplexNumber.html, 10.07.2017

Anhang

Quelltext der Quantencomputersimulation:

```
//// Main_visual Tab:
// safe 6
// global variables for the size of the window
int CURR_WIDTH;
int CURR_HEIGHT;
// global variables for the type of visualisation
// If GRAPHTYPE is 0, then it swows a bar graph of the probabilites of all possible
    quantum states.
// If GRAPHTYPE is 1, then it swows a bar graph of every Qubits probability to be 0 or
    1.
int GRAPHTYPE = 0;
// If DISPLAYTYPE is 0, then it shows the the menu for changing the timesteps and the
    qubitcount.
// If DISPLAYTYPE is 1, then it shows the the editor for the Qubitscore.
// If DISPLAYTYPE is 2, then it shows the the charts from GRAPHTYPE.
int DISPLAYTYPE = 0; // starts as 1, gets set to 0 when the user hits ok and gets set to
     two after the save gets updated
// The Editor class gets initialized (parameters are given when the Editor is used after
     the setup)
Editor e;
// Probabilitys for every quamtum position
float[] endoutput;

/**
 * Intitalizes the window with a hardcoded standard size and makes it possible to change
     the parameters.
 * Initializes the mp3 player.
 * It sets the application-icon and the name of the application, too.
 * Needs to be used in processing.
 **/
void setup() {
  //size(900, 600);
  fullScreen();
  minim = new Minim(this);
  player = minim.loadFile("sounds/click.wav");
  surface.setResizable(true);
  PImage titlebaricon = loadImage("icons/icon.png");
  surface.setIcon(titlebaricon);
  surface.setTitle("Quantumcomputersimulation");
  load_gate_images();
}

/**
 * This method is the standard update method of processing.
 * It triggers 60times per second if possible.
 * Depending on the typevariables, it's showing differend parts of the programm-GUI.
 * Methods, which only need to be activated once, are outsourced to the mouse_clicked()
    method.
 **/
void draw() {
  //System.out.println(DISPLAYTYPE);
```

```
  window_resized_update();
  // Setup
  if (DISPLAYTYPE == 0) drawSetup();
  // Editor
  if (DISPLAYTYPE == 1) {
    key_press_check();
    e.editorDraw();
    e.check_for_action();
  }
  // Graphs
  if (DISPLAYTYPE == 2) {
    if (GRAPHTYPE == 0) drawBarchart(endoutput);
    if (GRAPHTYPE == 1) drawQubitchart(endoutput);
  }
  check_if_too_small();
}

//// Main_technical Tab:
import java.util.Arrays;

// The program is totaly ok with changing theese values als long as the scorewidth is
    bigger than 1 !!!
// But the visualisation depends on these values and will go haywire when you change
    them, so only change the values when you use the program internally
int SCORELENGTH = 10;  // Timesteps
int SCOREWIDTH = 3;   // Qubitcount
// initializes main classes
Qubits q;
Gates g = new Gates();

/**
 * This Method needs a Qubitscore as an input and uses it on an new Instance of Qubits.
 * It uses SCOREWIDTH and SCOREHEIGHT as global parameters to determine the length and
     breadth of its action on the Qubits
 * It returns the probability for every quantum position
 **/
float[] state_calculator() {

  // infuses the score with all the needed gates
  Qubitscore qscore = e.getData();

  // calculates the kroeneckerproduct of every timestep and saves it
  c[][][] after_kroenecker = new c[SCORELENGTH][(int) pow(2, SCOREWIDTH)][(int) pow(2,
      SCOREWIDTH)];
  for (int i=0; i<SCORELENGTH; i++) {
    // special case of a C-Not gate
    // can just copy
    if (qscore.has_cnot(i) == true) {
      after_kroenecker[i] = qscore.get_cnot(i);
    }
    // normal case of an arrangement of gates
    // can copy the kroeneckerproduct
    else {
      c[][][] normal_gates = new c[SCOREWIDTH][][];
      for (int j=0; j<normal_gates.length; j++) {
        normal_gates[j] = qscore.get_other(j, i);
      }
      after_kroenecker[i] = kroenecker_mult(normal_gates);
    }
  }
  // calculates the matrixproduct of the Qubitstate and all of the matrices
  Qubits after_product = new Qubits(SCOREWIDTH);
  after_product = matrix_mult(q, after_kroenecker);

  // outputs the probabilities of the qubitpositions
  //after_product.printProbabilities();

  //returns the endresult
```

```
  return after_product.getProbabilities();
}

//// c_not_method Tab:
/**
 * Calculates a C-Not gate whitch is as big as the kroeneckerproduct of all the possible
     gates in one timeinstant
 * e controls the size, because it gives the number of qubits in the system
 * c and t describe the important qubits
 * only usefull for c!=t and e>c and e>t
 * IMPORTANT: c and t count from 0
 * TODO: fix the reverse code issue (not important)
 **/
c[][] cnot_builder(int e, int c, int t) {
  // entanglementsize, controlqubit, targetqubit

  // code hack
  // wrote code for other direction, so I need to reverse the qubits to obtain the right
      C-Not gate
  t=e-t-1;
  c=e-c-1;
  // builds two Arrays with the positions before and after the CNOT
  int p = (int) pow(2, e);     //positionscount
  int[][] sp = new int[p][e];  //superposition
  int[][] spa = new int[p][e]; //superposition after cnot

  // creates all the positions (binary count up)
  for (int i=0; i<p; i++) {
    int b = Integer.parseInt(binary(i));
    for (int j=0; j<e; j++) {
      sp[i][j] = b%10;
      b=b/10;
    }
  }

  // creates all the positions after the C-Not gate
  // same as the binary count up, but when the nuber at the magnitude of c is one, then
      the number at the magnitude of t will get reversed
  for (int i=0; i<p; i++) {
    spa[i] = sp[i].clone();
    if (sp[i][c]==1) {
      if (spa[i][t]==1) spa[i][t]=0;
      else if (spa[i][t]==0) spa[i][t]=1;
    }
  }

  // Creates and fills a third p by 2 array, which contains the position of every type
      of qubitposition in the sp array on the left column
  //  and the corresponding position of the same qubitposition in the spa array in the
      right column.
  // left collum in order!
  int[][] ones = new int[p][2];
  for (int i=0; i<p; i++) {
    for (int j=0; j<p; j++) {
      if (Arrays.equals(sp[i], spa[j])) {
        ones[i][0]=i; // redundant because of the orderedness of the left collumn, but
            good for better understanding
        ones[i][1]=j; // the position of the corresponding qubitposition
      }
    }
  }

  // builds the output CNOT matrix with the size of 2 to the power of e squared
  c[][] output = new c[p][p];
  for (int i=0; i<p; i++) {
    for (int j=0; j<p; j++) {
      output[i][j] = new c(0, 0);
    }
```

```
  }

  // sets the ones based on the "ones" array
  for (int i=0; i<p; i++) {
    output[ ones[i][0] ][ ones[i][1] ].update(1, 0);
  }

  return output;
}

//// complex_class Tab:
/**
 * Creates a complex number of the form a+bi
 **/
class c {
  float real;
  float imaginary;

  // constructs two numbers
  // the first takes the part of the real number, the other of the imaginary number
  c(float a, float b) {
    this.real=a;
    this.imaginary=b;
  }

  // returns the imaginary part of the complex number
  float i() {
    return imaginary;
  }

  // returns the real part of the complex number
  float r() {
    return real;
  }

  // returns the complex conjugate of the complex number
  c conjugate() {
    return new c(real, -imaginary);
  }

  // returns the sum of this comlex number and a given complex number
  c sum(c p) {
    float real_part = real+p.r();
    float imaginary_part = imaginary+p.i();
    return new c(real_part, imaginary_part);
  }

  // returns the product of this comlex number and a given complex number
  c product(c p) {
    float real_part = real*p.r() - imaginary*p.i();
    float imaginary_part = real*p.i() + imaginary*p.r();
    return new c(real_part, imaginary_part);
  }

  // returns the product of this complex number and the comlex conjugate of it
  // unused
  c complex_conjugate_square() {
    return this.product(this.conjugate());
  }

  // returns the real part of the product of this complex number and the comlex
      conjugate of it
  // the return value is also the probability
  float float_complex_conjugate_square() {
    return this.complex_conjugate_square().r();
  }

  // prints this complex number
```

```
  String print() {
    return real+"+"+imaginary+"i ";
  }

  // sets the complex number on request
  // unused
  void update(float a, float b) {
    real=a;
    imaginary=b;
  }
}

//// gate_class Tab:
/**
 * contains all the gates
 * creation of the CNOT is outsourced
 * instead cnot gives an array back that tells the system that a C-Not gate is needed
 **/
class Gates {
  float v = 1.0/sqrt(2);
  c[][] i;
  c[][] x;
  c[][] y;
  c[][] z;
  c[][] h;
  c[][] s;
  c[][] sd;
  c[][] t;
  c[][] td;
  c[][] cnot;

  // Constructs every gate
  Gates() {
    //Identity
    this.i = new c[][] {
      {new c(1, 0), new c(0, 0)},
      {new c(0, 0), new c(1, 0)}};
    //Bitflip
    this.x = new c[][] {
      {new c(0, 0), new c(1, 0)},
      {new c(1, 0), new c(0, 0)}};
    // Phase&Bitflip
    this.y = new c[][] {
      {new c(0, 0), new c(0, -1)},
      {new c(0, 1), new c(0, 0)}};
    // Phaseflip
    this.z = new c[][] {
      {new c(1, 0), new c(0, 0)},
      {new c(0, 0), new c(-1, 0)}};
    // Hadamard
    this.h = new c[][] {
      {new c(v, 0), new c(v, 0)},
      {new c(v, 0), new c(-v, 0)}};
    // Pauli
    this.s = new c[][] {
      {new c(1, 0), new c(0, 0)},
      {new c(0, 0), new c(0, 1)}};
    // Pauli dagger
    this.sd = new c[][] {
      {new c(1, 0), new c(0, 0)},
      {new c(0, 0), new c(0, -1)}};
    // Troffioli
    this.t = new c[][] {
      {new c(1, 0), new c(0, 0)},
      {new c(0, 0), new c(v, v)}};
    // Troffioli dagger
    this.td = new c[][] {
      {new c(1, 0), new c(0, 0)},
```

```
      {new c(0, 0), new c(-v, -v)}};
  }

  // all of theese return their gate counterpart
  c[][] i() {
    return i;
  }
  c[][] x() {
    return x;
  }
  c[][] y() {
    return y;
  }
  c[][] z() {
    return z;
  }
  c[][] h() {
    return h;
  }
  c[][] s() {
    return s;
  }
  c[][] sd() {
    return sd;
  }
  c[][] t() {
    return t;
  }
  c[][] td() {
    return td;
  }
  c[][] cnot(int e, int c, int t) {
    return cnot_builder(e, c, t);
  }
}

/**
 * saves a single quantum gate
 * e.g.: gate x = new gate(g.cnot(SCOREWIDTH, 0, 2), true); saves a specific C-Not gate.
 * Also saves if the gate is a CNOT gate.
 **/
class gate {
  c[][] storage;
  boolean isCnot;
  boolean isUsed;

  // Double Costructor
  // You may only call this method with gate(*GATESCLASS*.*GATE*,true/false);. e.g: gate
      (g.t,false);
  gate(c[][] in, boolean isCnot) {
    this.storage = in;
    this.isCnot = isCnot;
    this.isUsed = true;
  }
  // If no parameter is given, gate{} is marked as unused
  gate() {
    this.storage = g.i();
    this.isUsed = false;
  }
  // Theese methods return the saved data in the class-instance
  c[][] getArray() {
    return storage;
  }
  boolean cnot() {
    return isCnot;
  }
  boolean isUsed() {
    return isUsed;
```

```
  }
}

//// kroenecker_mult_method Tab: (richtiger Name: Kronecker)
/**
 * family of 2 methods to calculate a kroeneckerproduct of arbitrary many matrices
 * kroenecker_mult parts the p*q*r matrix to p q*r marices and gives two of them or one
     and the return of the kroenecker_2 to kroenecker_2
 * kroenecker_2 multiplies 2 arbitrary 2d matrices the kroenecker way
 **/
// main method
// convertes the one big 3d Array to a 2d and a smaller 3d Array
// if it produces two 2d Arrays the endcase is reached
c[][] kroenecker_mult(c[][][] input) {
  if (input.length==2) {
    return kroenecker_2(input[0], input[1]);
  } else {
    c[][] first = input[0];
    c[][][] rest = new c[input.length-1][][];
    for (int i=1; i<input.length; i++) {
      rest[i-1] = input[i];
    }
    return kroenecker_2(first, kroenecker_mult(rest));
  }
}

// calculating method
// calculates the kroeneckerproduct of two 2d matrices of arbitrary sizes
c[][] kroenecker_2(c[][] to, c[][] from) {
  int w_to = to[0].length;
  int h_to = to.length;
  int w_from = from[0].length;
  int h_from = from.length;

  c[][] output = new c[h_to*h_from][w_to*w_from];

  for (int i=0; i<h_to; i++) {     // Iterates over first Array
    for (int j=0; j<w_to; j++) {  //
      for (int k=0; k<h_from; k++) {     // Iterates over second Array
        for (int l=0; l<w_from; l++) {  //
          int y = i*h_from + k;
          int x = j*w_from + l;

          output[y][x] = to[i][j].product(from[k][l]);
        }
      }
    }
  }
  return output;
}

//// matrix_mult_method Tab:
/**
 * family of 2 methods to calculate a matrixproduct of arbitrary many reglemented
     matrices with an 1d matrix
 * matrix_mult parts the p*q*r matrix to p q*r marices and gives two of one and the
     return of the last matrix_mult_2 to matrix_mult_2
 * matrix_mult_2 multiplies a reglemented 2d matrix with an 1d matrix
 **/
// main method
// convertes the one big 3d Array to many 2d arrays
// then it multiplies all of them with the qubitstate
Qubits matrix_mult(Qubits qubits, c[][][] kgates) {
  for (int i=0; i<kgates.length; i++) {
    qubits = matrix_mult_2(qubits, kgates[i]);
  }
  return qubits;
}
```

```
// calculating method
// calculates the matrixproduct of two 2d matrices of reglemented sizes
Qubits matrix_mult_2(Qubits qubits, c[][] kgate) {
  c[] q = qubits.getArray();
  c[] out = new c[q.length];
  // tests if the matrices are multiplicateable
  if (q.length == kgate[0].length) {
    int l = 1;                          // width of first matrix
    int m = kgate.length;               // height of the second array
    int n = kgate[0].length;            // width of the second array and height of the
        first array

    for (int _l=0; _l<l; _l++) {        // redundant, but for better understanding
      for (int _m=0; _m<m; _m++) {
        c partSum = new c(0, 0);
        for (int _n=0; _n<n; _n++) {
          partSum = partSum.sum(q[_n].product(kgate[_m][_n]));
        }
        out[_m] = partSum;
      }
    }

    qubits.update(out);
    return qubits;
  } else {
    System.out.println("Error in matrix multiplication! The given matrices are not
        multiplicateable!");
    return qubits;
  }
}

//// qubit_class Tab:
/**
 * Creates the qubitstartstate and contains all the qubitstates that get calculated
 * works only for positive sizes
 **/
class Qubits {
  c[] states;
  int size;
  int pow2;

  // constructs a qubitstate with every qubit in the |0> position
  Qubits(int size) {
    this.size=size;
    pow2 = (int) pow(2, size);
    states = new c[pow2];
    for (int i=0; i<pow2; i++) {
      states[i] = new c(0, 0);
    }
    states[0].update(1, 0);
  }

  // sets the Qubitstate on request
  void update(c[] a) {
    states=a;
  }

  // returns the qubitstate on request
  // unused
  c[] getArray() {
    return states;
  }

  // returns the qubitstate with every entry as the complex conjugate product of the
      original entry
  float[] getProbabilities() {
    float[] output = new float[pow2];
```

```
    for (int i=0; i<pow2; i++) {
      output[i] = states[i].float_complex_conjugate_square();
    }
    return output;
  }

  // prints the qubitstateamplitude
  // unused
  void printAmplitudes() {
    for (int i=0; i<pow2; i++) {
      System.out.println(states[i].print());
    }
  }

  // prints the qubitstate with every entry as the complex conjugate product of the
      original entry to console
  // IMPORTANT: it rounds the probabilities to 3 decimal places to get rid of
      99.999999999% for example
  void printProbabilities() {
    float[][] output = new float[pow2][2];
    for (int i=0; i<pow2; i++) {
      output[i][0] = states[i].float_complex_conjugate_square();
      output[i][1] = Integer.parseInt(binary(i));

      // code for filling in the zeroes before the first one and converting the float to
          an int
      int fill0 = (int) pow(10, size);
      int with1 = fill0+(int) output[i][1];
      String withS1 = Integer.toString(with1);
      String no1 = withS1.substring(1);

      // code for rounding a number to 3 decimal places
      float times100000 = output[i][0] * 100000;
      int round100000 = round(times100000);
      float roundNormal = round100000 / 1000.0;

      System.out.println("|"+no1+"> : "+roundNormal+"%");
    }
  }
}

//// qubitscore_class Tab:
/**
 * This Class stores the Qubitscore_timesteps, whitch make up the whole Qubitscore.
 * It is used for determining the endqubitposition.
 **/
class Qubitscore {
  // Saves every QS_timestep instances in one Array.
  // separation because of the different requirements of the C-Not and normal gates
  Qubitscore_timestep[] steps;

  // The constructor creates instances of Qubitscore_timestep in every part of the array
      .
  Qubitscore() {
    steps = new Qubitscore_timestep[SCORELENGTH];
    for (int i=0; i<steps.length; i++) {
      steps[i] = new Qubitscore_timestep();
    }
  }

  // Theese methods get, put and remove gates.
  // It is outsourced to Qubitscore_timestep.
  void put_other(c[][] gatematrix, int qubit_number, int timestep) {
    steps[timestep].put_other(gatematrix, qubit_number);
  }
  void put_cnot(int control_qubit, int target_qubit, int timestep) {
    steps[timestep].put_cnot(control_qubit, target_qubit);
  }
```

```
  void remove_other(int qubit_number, int timestep) {
    steps[timestep].remove_other(qubit_number);
  }
  void remove_cnot(int timestep) {
    steps[timestep].remove_cnot();
  }
  c[][] get_other(int qubit_number, int timestep) {
    return steps[timestep].get_other(qubit_number);
  }
  c[][] get_cnot(int timestep) {
    return steps[timestep].get_cnot();
  }
  int get_cnot_controlqubit(int timestep) {
    return steps[timestep].get_cnot_control();
  }
  int get_cnot_targetqubit(int timestep) {
    return steps[timestep].get_cnot_target();
  }
  String size() {
    return steps.length + " x " + steps[0].size();
  }

  // Theese methods determin if a gate is placed at specific coordinates.
  // It is outsourced to Qubitscore_timestep.
  boolean can_insert_cnot(int timestep) {
    return steps[timestep].can_insert_cnot();
  }
  boolean can_insert_other(int qubit_number, int timestep) {
    return steps[timestep].can_insert_other(qubit_number);
  }
  boolean has_cnot(int timestep) {
    return steps[timestep].stores_cnot();
  }
  boolean has_other(int qubit_number, int timestep) {
    return steps[timestep].has_other(qubit_number);
  }
}

/**
 * This Class stores gates, which make up the whole Qubitscore_timestep.
 * It differentiates between an array of normal gates, or a single C-Not gate.
 **/
class Qubitscore_timestep {
  // Stats of the possibility of saving a C-Not gate or other gates
  boolean cnot_stored = false;
  boolean other_stored = false;
  int count_other = 0;
  // reserved spaces for either one of them
  gate[] parallel = new gate[SCOREWIDTH];
  gate cnot;
  int from;
  int to;

  // The constructor creates instances of a placeholdergate in every part of the array.
  Qubitscore_timestep() {
    for (int i=0; i<parallel.length; i++) {
      parallel[i] = new gate();
    }
  }

  // Theese methods determine, which type of gate can be saved or is stored.
  // returns the possibility of saving a C-Not gate or another gate in this timestep
  boolean can_insert_cnot() {
    if (this.stores_cnot()==false) return !other_stored;
    else return false;
  }
  boolean can_insert_other(int qubit_number) {
    if (cnot_stored == true) {
```

```
      return false;
    } else {
      return (!parallel[qubit_number].isUsed());
    }
  }
  boolean has_other(int qubit_number) {
    if (cnot_stored == true) {
      return false;
    } else {
      return (parallel[qubit_number].isUsed());
    }
  }
  boolean stores_cnot() {
    return cnot_stored;
  }
  int get_cnot_control() {
    return from;
  }
  int get_cnot_target() {
    return to;
  }

  // theese methods put gates in this storage and detrmine the type of this storage
  void put_other(c[][] gatematrix, int qubit_number) {
    if (cnot_stored == false) {
      other_stored = true;
      parallel[qubit_number] = new gate(gatematrix, false);
      count_other++;
    } else System.out.println("Putting a normal gate in this storage is not possible, it
        already contains a C-Not gate");
  }
  void put_cnot(int control_qubit, int target_qubit) {
    if (other_stored == false) {
      cnot_stored = true;
      cnot = new gate(g.cnot(SCOREWIDTH, control_qubit, target_qubit), true);
      from = control_qubit;
      to = target_qubit;
    } else System.out.println("Putting a C-Not gate in this storage is not possible, it
       already contains a normal gate");
  }

  // theese methods remove gates from the storage
  void remove_other(int qubit_number) {
    if (other_stored == true) {
      if (count_other == 1) other_stored = false;
      parallel[qubit_number] = new gate();
      count_other--;
    } else System.out.println("You can't remove a nonexisting normal gate. (Or this
       contains a C-Not gate.)");
  }
  void remove_cnot() {
    if (cnot_stored == true) {
      cnot_stored = false;
      cnot = new gate();
      from = -1;
      to = -1;
    } else System.out.println("You can't remove a nonexisting C-Not gate. (Or this
       contains a normal gate.)");
  }

  // theese methods get gates from the storage
  c[][] get_other(int qubit_number) {
    if (cnot_stored == false) {
      return parallel[qubit_number].getArray();
    } else {
      System.out.println("error");
      return new c[0][0];
    }
```

```
  }
  c[][] get_cnot() {
    if (cnot_stored == true) {
      return cnot.getArray();
    } else {
      System.out.println("error");
      return new c[0][0];
    }
  }
  int size() {
    return parallel.length;
  }
}

//// visual_charts Tab:
/**
 * draws a barchart with all the poosible qubitpositions
 * writes the positions under the bars
 **/
void drawBarchart(float[] in) {
  // sets all the needed constants
  background(50);
  strokeWeight(1);
  int horizontal_space = CURR_WIDTH/8;
  int text_space = CURR_HEIGHT/6;
  int vertical_space = CURR_HEIGHT/8;
  translate(0, CURR_HEIGHT);                                  // set the coordinate sytem
      to the bottom left corner
  translate(horizontal_space, -text_space-vertical_space);  // translate the canvas
      according to the speces
  int max_x = CURR_WIDTH-2*horizontal_space;
  int max_y = CURR_HEIGHT-text_space-2*vertical_space; // maximum absolute y
  int count = in.length;
  float thickness = 1.0*max_x/count;

  // searches the biggest probability
  float max_prob = 0;
  for (int i=0; i<count; i++) {
    if (max_prob < in[i]) max_prob = in[i];
  }

  // draws the barchart
  stroke(200);
  pushMatrix();
  for (int i=0; i<count; i++) {
    float bar_height = in[i] * max_y/max_prob;
    fill(map(in[i], 0, 1, 60, 255));
    rect(0, 0, thickness-5, -bar_height);
    fill(180);
    textSize(thickness/6);
    text("|"+binary(i, SCOREWIDTH)+">", 0, thickness*1.1/6);
    textSize(thickness/3.5);
    text(round(in[i]*100000)/1000.0+"%", 0, thickness*1.1/2.4);
    translate(thickness, 0);
  }
  popMatrix();

  // draws the percentagelines
  stroke(108, 100);
  fill(222, 100);
  textAlign(LEFT);
  textSize(CURR_WIDTH/32);
  line(-horizontal_space*2/3, -max_y/(max_prob), max_x+horizontal_space/2, -max_y/(
      max_prob));                  // 100%
  text("100%", -horizontal_space*2/3, -max_y/(max_prob)-3);
  line(-horizontal_space*2/3, -max_y/(max_prob*1.33333), max_x+horizontal_space/2, -
      max_y/(max_prob*1.33333)); // 75%
  text("75%", -horizontal_space*2/3, -max_y/(max_prob*1.33333)-3);
```

```
  line(-horizontal_space*2/3, -max_y/(max_prob*2.0), max_x+horizontal_space/2, -max_y/(
      max_prob*2.0));          // 50%
  text("50%", -horizontal_space*2/3, -max_y/(max_prob*2.0)-3);
  line(-horizontal_space*2/3, -max_y/(max_prob*4.0), max_x+horizontal_space/2, -max_y/(
      max_prob*4.0));          // 25%
  text("25%", -horizontal_space*2/3, -max_y/(max_prob*4.0)-3);
}

/**
 * draws as many twotype charts as there are qubits
 * every twotype chart visualises the chance of the two possible measurements of the
     qubit
 **/
void drawQubitchart(float[] in) {
  // sets all the needed constants
  background(50);
  int horizontal_space = CURR_WIDTH/8;
  int text_space = CURR_HEIGHT/6;
  int vertical_space = CURR_HEIGHT/8;
  translate(0, CURR_HEIGHT);                                    // set the coordinate sytem
      to the bottom left corner
  translate(horizontal_space, -text_space-vertical_space);  // translate the canvas
      according to the speces
  int max_x = CURR_WIDTH-2*horizontal_space;
  int max_y = CURR_HEIGHT-text_space-2*vertical_space; // maximum absolute y
  float number_of_bars = SCOREWIDTH*3-1;
  float bar_thickness = max_x/number_of_bars;
  float space_thickness = max_x/number_of_bars;

  // calculates the probabilities
  float[] qubit_prob = new float[SCOREWIDTH];
  for (int i=0; i<SCOREWIDTH; i++) {
    float prob_sum = 0;
    for (int j=0; j<in.length; j++) {
      String s = binary(j, SCOREWIDTH);
      if (s.charAt(i)=='0') {
        prob_sum = prob_sum + in[j];
      }
    }
    qubit_prob[i] = prob_sum;
  }

  // searches the biggest probability
  float max_prob = 0;
  for (int i=0; i<SCOREWIDTH; i++) {
    if (max_prob < qubit_prob[i]) max_prob = qubit_prob[i];
    if (max_prob < 1-qubit_prob[i]) max_prob = 1-qubit_prob[i];
  }

  // draws the barchart
  pushMatrix();
  stroke(200);
  textSize(bar_thickness/6);
  for (int i=0; i<SCOREWIDTH; i++) {
    float bar_height = qubit_prob[i] * max_y/max_prob;
    fill(map(qubit_prob[i], 0, 1, 60, 255));
    rect(0, 0, bar_thickness-5, -bar_height);
    fill(180);
    text("Qubit Nr. "+(i+1), 0, bar_thickness*1.1/3);
    text(round(qubit_prob[i]*100000)/1000.0+"%", 0, bar_thickness*1.1/6);
    fill(0);
    text("|0>", 0, -8);
    translate(bar_thickness, 0);
    bar_height = (1-qubit_prob[i]) * max_y/max_prob;
    fill(map(1-qubit_prob[i], 0, 1, 60, 255));
    rect(0, 0, bar_thickness-5, -bar_height);
    fill(180);
    text(round((1-qubit_prob[i])*100000)/1000.0+"%", 0, bar_thickness*1.1/6);
```

```
    fill(0);
    text("|1>", 0, -8);
    translate(bar_thickness, 0);
    translate(space_thickness, 0);
  }
  popMatrix();

  // draws the percentagelines
  stroke(108, 100);
  fill(222, 100);
  textAlign(LEFT);
  textSize(CURR_WIDTH/32);
  line(-horizontal_space*2/3, -max_y/(max_prob), max_x+horizontal_space/2, -max_y/(
      max_prob));                  // 100%
  text("100%", -horizontal_space*2/3, -max_y/(max_prob)-3);
  line(-horizontal_space*2/3, -max_y/(max_prob*1.33333), max_x+horizontal_space/2, -
      max_y/(max_prob*1.33333)); // 75%
  text("75%", -horizontal_space*2/3, -max_y/(max_prob*1.33333)-3);
  line(-horizontal_space*2/3, -max_y/(max_prob*2.0), max_x+horizontal_space/2, -max_y/(
      max_prob*2.0));          // 50%
  text("50%", -horizontal_space*2/3, -max_y/(max_prob*2.0)-3);
  line(-horizontal_space*2/3, -max_y/(max_prob*4.0), max_x+horizontal_space/2, -max_y/(
      max_prob*4.0));          // 25%
  text("25%", -horizontal_space*2/3, -max_y/(max_prob*4.0)-3);
}

//// visual_editor Tab:
/**
 * This is the Editor class.
 * It stores every GUI-change to the Qubitscore(model) in the Qubitscore(classinstance).
 * It enables the user of the programm to insert code on a mouse based interface instead
     of changing the code.
 **/
class Editor {
  Qubitscore qs;
  int[][] gatepics = new int[SCOREWIDTH][SCORELENGTH]; // -1 means not used
  boolean hasgate = false;
  // 0:Identity i, 1:Bitflip x, 2:Bit&Phaseflip y, 3:Phaseflip z, 4:Hadamard h, 5:Pauli
      s, 6:Pauli_dagger sd, 7:Troffioli t, 8:Troffioli_dagger td, 9:cnot
  int gatetype;
  // saves if a C-Not control coordinates are set, but not their target coordinates.
  int CNOT_timestep;
  int CNOT_qubit;
  boolean CNOT_half_placed = false;

  // The constructor of editor creates a new Qubitscore.
  Editor() {
    qs = new Qubitscore();
    for (int i=0; i<SCOREWIDTH; i++) {
      for (int j=0; j<SCORELENGTH; j++) {
        gatepics[i][j]=-1;
      }
    }
  }

  // getData returns the saved Qubitscore.
  Qubitscore getData() {
    return qs;
  }

  // This method returns the state of the has_gate varibale, which describes if a gate
      is currently picked .
  boolean get_picked_up() {
    return hasgate;
  }

  // This method returns the state of the gatetype varibale, which describes what a gate
      is currently picked up
```

```
int get_picked_type() {
  return gatetype;
}

// editorDraw is used to draw a GUI.
void editorDraw() {
  float gateimage_height = CURR_HEIGHT/6;
  float gateimage_width = CURR_WIDTH/12;
  float lesser_const = min(gateimage_height, gateimage_width);
  background(50);
  stroke(0);
  strokeWeight(1);
  imageMode(CENTER);
  fill(100);
  rect(-1, -1, CURR_WIDTH+1, gateimage_height+1);
  pushMatrix();
  translate(gateimage_width/2.0, gateimage_height/2);
  translate(gateimage_width/4.0, 0);
  for (int i=0; i<11; i++) {
    if (i==10) translate(gateimage_width/2.0, 0); // one extra for trashexclusion
    image(pictures[i], 0, 0, 0.9*lesser_const, 0.9*lesser_const);
    translate(gateimage_width, 0);
  }
  popMatrix();
  float quscore_height = (CURR_HEIGHT - gateimage_height)*9.0/10.0;
  float quscore_width = CURR_WIDTH*4.0/5.0;
  float edge_gap = CURR_WIDTH/10.0;
  float line_gap = quscore_height/SCOREWIDTH;
  float point_gap = quscore_width/SCORELENGTH;
  float lesser_const2 = min(line_gap,point_gap);

  pushMatrix();
  translate(CURR_WIDTH/10.0, gateimage_height+quscore_height/18.0); // translation in
      the quscorearea
  translate(0, line_gap/2); // translation for an easier loop for the lines
  for (int i=0; i<SCOREWIDTH; i++) {
    noStroke();
    fill(100);
    triangle(quscore_width, line_gap/40.0+1, quscore_width, -line_gap/40.0,
        quscore_width+edge_gap/2, 0);
    stroke(100);
    strokeWeight(line_gap/20.0);
    line(0, 0, quscore_width, 0); // -edge_gap/2
    fill(222);
    textAlign(RIGHT, BASELINE);
    textSize(lesser_const/2.0);
    text("Q", -edge_gap/3.0, 0);
    textAlign(LEFT, TOP);
    textSize(lesser_const/4.0);
    text(i, -edge_gap/3.0, 0);
    pushMatrix();
    translate(point_gap/2, 0); // translation for an easier loop for the points
    for (int j=0; j<SCORELENGTH; j++) {
      strokeWeight(line_gap/10);
      stroke(222);
      point(0, 0);
      translate(point_gap, 0);
    }
    popMatrix();
    translate(0, line_gap);
  }
  popMatrix();
  // Draw a gate if it is in hand.
  imageMode(CENTER);
  // checks for normal gates
  if (hasgate==true && gatetype != 9) {
    image(pictures[gatetype], mouseX, mouseY, 0.5*lesser_const, 0.5*lesser_const);
  }
```

```
    // checks for CNot gates
    else if(hasgate==true && gatetype == 9){
      if(CNOT_half_placed==false) image(pictures[gatetype], mouseX, mouseY, 0.5*
          lesser_const, 0.5*lesser_const);
      else{
        float imX = point_gap*(CNOT_timestep+0.5)+CURR_WIDTH/10.0;
        float imY = line_gap*(CNOT_qubit+0.5)+gateimage_height+quscore_height/18.0;
        stroke(130, 231, 29); strokeWeight(max(line_gap/10, 0.9*lesser_const2)*0.8);
            point(imX, mouseY); // draws the targeting C-Not-gate part
        strokeWeight(0.1*lesser_const); line(imX, imY, imX, mouseY); // draws the line
            connecting both parts
        image(pictures[9], imX, imY, 0.9*lesser_const2, 0.9*lesser_const2); // draws the
             controling C-Not-gate part
      }
    }
    // Draw already placed gates
    pushMatrix();
    translate(CURR_WIDTH/10.0, gateimage_height+quscore_height/18.0); // translation in
        the quscorearea
    translate(point_gap/2, line_gap/2); // translation for an easier placement of the
        gates
    // checks for normal gates
    for (int i=0; i<SCOREWIDTH; i++) {
      for (int j=0; j<SCORELENGTH; j++) {
        if (qs.has_other(i, j)==true) {
          image(pictures[gatepics[i][j]], point_gap*j, line_gap*i, 0.9*lesser_const2,
              0.9*lesser_const2);
        }
      }
    }
    // checks for C-Not gates
    // 9=Control, 11=Target
    for (int j=0; j<SCORELENGTH; j++) {
      if (qs.has_cnot(j)==true) {
        int control=0;
        int target=0;
        for (int i=0; i<SCOREWIDTH; i++) {
          if (gatepics[i][j]==11) target=i;
          if (gatepics[i][j]==9) control=i;
        }
        stroke(130, 231, 29);
        strokeWeight(max(line_gap/10, 0.7*lesser_const2));
        point(point_gap*j, line_gap*target);
        strokeWeight(0.1*lesser_const);
        line(point_gap*j, line_gap*target, point_gap*j, line_gap*control);
        image(pictures[9], point_gap*j, line_gap*control, 0.9*lesser_const2, 0.9*
            lesser_const2);
      }
    }
    popMatrix();
  }

  void check_for_action() {
    float gate_width = CURR_WIDTH/12.0;
    float gateimage_height = CURR_HEIGHT/6;
    float quscore_height = (CURR_HEIGHT - gateimage_height)*9.0/10.0;
    // if you press your mouse in the gatemenu to pick up a gate
    if (mouseY < CURR_HEIGHT/6.0 && mouseX>gate_width/4.0 && mouseX<gate_width*10.25 &&
        mousePressed==true && hasgate==false) {
      float temp_x = mouseX - gate_width/4.0;
      int gate_type = floor(temp_x/gate_width);
      hasgate = true;
      gatetype = gate_type;
    }
    // if you press your mouse over the trash button
    if (mouseY < CURR_HEIGHT/6.0 && mouseX>gate_width*10.75 && mouseX<gate_width*11.75
        && mousePressed==true && hasgate==true) {
      hasgate = false;
```

```
      gatetype = -1;
    }
    // if you press your mouse with a normal gate over the qubitscore
    if (mouseX > CURR_WIDTH/10.0 && mouseX < CURR_WIDTH*9.0/10.0 && mouseY >
        gateimage_height+quscore_height/18.0 && mouseY < gateimage_height+quscore_height
        *19.0/18.0 && mousePressed==true) {
      // maps mousecoordinates in the qubitscore to normed values boetween 0 and 1
      // map(cuurent value, lower current bound, upper current bound, lower target bound
          , upper target bound)
      float temp_x_rel = map(mouseX, CURR_WIDTH/10.0, CURR_WIDTH*9.0/10.0, 0, 1);
      float temp_y_rel = map(mouseY, gateimage_height+quscore_height/18.0,
          gateimage_height+quscore_height*19.0/18.0, 0, 1);
      int curr_timestep = floor(temp_x_rel*SCORELENGTH);
      int curr_qubit = floor(temp_y_rel*SCOREWIDTH);
      // sets borders, so only exact presses count
      boolean exact_enough = true;
      if ((temp_x_rel*SCORELENGTH)%1<0.25 || (temp_x_rel*SCORELENGTH)%1>0.75 || (
          temp_y_rel*SCOREWIDTH)%1<0.25 || (temp_y_rel*SCOREWIDTH)%1>0.75) exact_enough
          = false;
      // this checks for the placing of normal gates and stops picking gates up directly
           after they have been placed
      if (click==true) {
        boolean tempp = false; // is used to stop deplacing a gate directly after
            placing it
        if (exact_enough == true && hasgate==true && gatetype!=9) {
          if (qs.can_insert_other(curr_qubit, curr_timestep)==true) {
            qs.put_other(gatenumbers_to_gates(gatetype), curr_qubit, curr_timestep);
            gatepics[curr_qubit][curr_timestep] = gatetype;
            hasgate = false;
            gatetype = -1;
            tempp = true;
          }
        }
        // this checks for the deplacing of normal gates
        if (exact_enough == true && hasgate==false && tempp==false) {
          if (qs.has_other(curr_qubit, curr_timestep)) {
            qs.remove_other(curr_qubit, curr_timestep);
            gatetype = gatepics[curr_qubit][curr_timestep];
            gatepics[curr_qubit][curr_timestep] = -1;
            hasgate = true;
          }
        }
        // this checks for the placing of C-Not gates
        if (exact_enough == true && hasgate==true && gatetype==9) {
          if (qs.can_insert_cnot(curr_timestep)==true) {
            tempp = false;
            // first C-Not part
            if (CNOT_half_placed==false) {
              CNOT_timestep = curr_timestep;
              CNOT_qubit = curr_qubit;
              CNOT_half_placed = true;
              tempp = true;
            }
            // second C-Not part
            if (tempp==false && CNOT_half_placed==true && curr_timestep==CNOT_timestep
                && curr_qubit!=CNOT_qubit) {
              gatepics[CNOT_qubit][CNOT_timestep] = 9;
              gatepics[curr_qubit][CNOT_timestep] = 11;
              qs.put_cnot(CNOT_qubit, curr_qubit, CNOT_timestep);
              CNOT_half_placed = false;
              hasgate = false;
            }
          }
          tempp=true;
        }
        // this checks for the deplacing of C-Not gates
        if (exact_enough == true && hasgate==false && tempp==false) {
          if (qs.has_cnot(curr_timestep)==true) {
```

```
            if (curr_qubit==qs.get_cnot_controlqubit(curr_timestep)) {
              // this runs when you sucsessfully clicked to remove a C-Not gate
              gatetype = gatepics[curr_qubit][curr_timestep];
              gatepics[qs.get_cnot_controlqubit(curr_timestep)][curr_timestep] = -1;
              gatepics[qs.get_cnot_targetqubit(curr_timestep)][curr_timestep] = -1;
              qs.remove_cnot(curr_timestep);
              hasgate = true;
            }
          }
        }
      }
    }
  }
}

/**
 * This method builds the foundation for the editor.
 * It is the first part of the GUI, used to set the two Main Parameters SCOREWIDTH and
     SCORELENGTH for later use through the user
 **/
boolean noMore = false;
void drawSetup() {
  float th = CURR_HEIGHT/10; // textheight
  float heightdiff = -CURR_HEIGHT/80.0; // needed for centered text
  // textparameters
  background(50);
  fill(180);
  stroke(255);
  textAlign(CENTER, CENTER);
  // drawfunction
  pushMatrix();
  // Programinformation
  translate(CURR_WIDTH/2, CURR_HEIGHT/3);
  textSize(th/3);
  text("This program is a simulation of a quantum computer.", 0, th*3.25);
  text("Interact with the left mouse button and continue by using the right click.", 0,
      th*3.75);
  text("You can change the qubitscorevariables in this tab, and use gates on it
      thereafter.", 0, th*4.25);
  text("At last, you will be able to look at the results as two charts.", 0, th*4.75);
  textSize(th);
  if (mousePressed == false && noMore==true) noMore = false;
  // Scorelength
  pushMatrix();
  translate(-CURR_WIDTH/6, 0);
  float trmouseX = mouseX-CURR_WIDTH/3.0; // needed for mousecoordinates
  float trmouseY = mouseY-CURR_HEIGHT/3.0;
  text(SCORELENGTH, 0, heightdiff);
  textSize(th/2);
  text("Timesteps", 0, -2*th);
  textSize(th);
  triangle(-th/2.0, th/2.0, th/2.0, th/2.0, 0, 1.5*th);
  if (trmouseX<th/2.0 && trmouseX>-th/2.0 && trmouseY<1.5*th && trmouseY>th/2.0 &&
      mousePressed == true && noMore == false) {
    SCORELENGTH--;
    player.play();
    player.rewind();
    noMore=true;
  }
  triangle(-th/2.0, -th/2.0, th/2.0, -th/2.0, 0, -1.5*th);
  if (trmouseX<th/2.0 && trmouseX>-th/2.0 && trmouseY>-1.5*th && trmouseY<-th/2.0 &&
      mousePressed == true && noMore == false) {
    SCORELENGTH++;
    player.play();
    player.rewind();
    noMore=true;
  }
  if (SCORELENGTH>20) SCORELENGTH=20;
```

```
  if (SCORELENGTH<1) SCORELENGTH=1;
  popMatrix();
  // Scorewidth
  pushMatrix();
  translate(CURR_WIDTH/6, 0);
  trmouseX = mouseX-2*CURR_WIDTH/3.0; // needed for mousecoordinates
  trmouseY = mouseY-CURR_HEIGHT/3.0;
  text(SCOREWIDTH, 0, heightdiff);
  textSize(th/2);
  text("Qubits", 0, -2*th);
  textSize(th);
  triangle(-th/2.0, th/2.0, th/2.0, th/2.0, 0, 1.5*th);
  if (trmouseX<th/2.0 && trmouseX>-th/2.0 && trmouseY<1.5*th && trmouseY>th/2.0 &&
      mousePressed == true && noMore == false) {
    SCOREWIDTH--;
    player.play();
    player.rewind();
    noMore=true;
  }
  triangle(-th/2.0, -th/2.0, th/2.0, -th/2.0, 0, -1.5*th);
  if (trmouseX<th/2.0 & trmouseY<-th/2.0 && mousePressed == true && noMore == false) {
    SCOREWIDTH++;
    player.play();
    player.rewind();
    noMore=true;
  }
  if (SCOREWIDTH>5) SCOREWIDTH=5;
  if (SCOREWIDTH<2) SCOREWIDTH=2;
  popMatrix();
  popMatrix();
}

//// visual_extra Tab:
// This imports a mp3 player.
import ddf.minim.*;
Minim minim;
AudioPlayer player;

// checks if the window is resized and then updates the two size variables
void window_resized_update() {
  CURR_HEIGHT=height;
  CURR_WIDTH=width;
}

// This standard method triggers methods and classes by a call from userinteraction with
     the mouse buttons.
void mouseClicked() {
  // checks if the user used the right click
  // changes the DISPLAYTYPE/GRAPHTYPE if the user uses the right click
  // it calculates the endoutputarray.
  if (mouseButton == RIGHT) {
    player.play();
    player.rewind();
    // Change between different graphs in the Output
    if (DISPLAYTYPE == 2) {
      GRAPHTYPE = abs(GRAPHTYPE-1);
    }
    // Change from Editor to Output
    if (DISPLAYTYPE == 1) {
      q = new Qubits(SCOREWIDTH);
      endoutput = state_calculator();
      DISPLAYTYPE = 2;
    }
    // Change from Setup to Editor
    if (DISPLAYTYPE == 0) {
      // editor gets initialized with the legth and width parameters
      e = new Editor();
      DISPLAYTYPE = 1;
```

```
    }
  }
  //
  //
  if (mouseButton == LEFT) {
    // nothing
  }
}

// This mathod loads all of the gate-icons in PImage variables to be used in the editor.
// 0:Identity i, 1:Bitflip x, 2:Bit&Phaseflip y, 3:Phaseflip z, 4:Hadamard h, 5:Pauli s,
     6:Pauli_dagger sd, 7:Troffioli t, 8:Troffioli_dagger td, 9:cnot, 10:trash
PImage[] pictures;
void load_gate_images() {
  pictures = new PImage[11];
  pictures[9] = loadImage("gates/cnot.png");
  pictures[0] = loadImage("gates/i.png");
  pictures[1] = loadImage("gates/x.png");
  pictures[2] = loadImage("gates/y.png");
  pictures[3] = loadImage("gates/z.png");
  pictures[4] = loadImage("gates/h.png");
  pictures[5] = loadImage("gates/s.png");
  pictures[6] = loadImage("gates/sd.png");
  pictures[7] = loadImage("gates/t.png");
  pictures[8] = loadImage("gates/td.png");
  pictures[10] = loadImage("icons/trash.png");
}

void check_if_too_small() {
  if (CURR_WIDTH<200 || CURR_HEIGHT<200) {
    background(255, 0, 0);
    if (CURR_WIDTH>550) {
      fill(0);
      translate(CURR_WIDTH/2, CURR_HEIGHT/2);
      textAlign(CENTER, CENTER);
      textSize(CURR_HEIGHT/2);
      text("TOO SMALL", 0, 0);
    }
  }
}

// This method convertes the gatenumbers to gates.
// without the cnot gate
// 0:Identity i, 1:Bitflip x, 2:Bit&Phaseflip y, 3:Phaseflip z, 4:Hadamard h, 5:Pauli s,
     6:Pauli_dagger sd, 7:Troffioli t, 8:Troffioli_dagger td
c[][] gatenumbers_to_gates(int gatetype) {
  switch(gatetype) {
  case 0:
    return g.i();
  case 1:
    return g.x();
  case 2:
    return g.y();
  case 3:
    return g.z();
  case 4:
    return g.h();
  case 5:
    return g.s();
  case 6:
    return g.sd();
  case 7:
    return g.t();
  case 8:
    return g.td();
  default:
    System.out.println("Number to gate conversion error");
    return g.i();
```

```
  }
}

// this method returns true one tim after a click has happend
boolean click = false;
boolean press = false;
void key_press_check() {
  if (click==true && mousePressed==true && press==true) click = false;
  if (click==false && mousePressed==true && press==false) {
    click = true;
    press = true;
  }
  if (mousePressed == false) press = false;
}
```

Eidesstattliche Erklärung

Wir erklären, dass wir die vorliegende Arbeit mit dem Titel *Qubits als Bestandteile von Quantencomputern* selbstständig, ohne unzulässige fremde Hilfe angefertigt und nur unter Verwendung der angegebenen Literatur und Hilfsmittel verfasst haben. Sämtliche Stellen, die wörtlich oder inhaltlich anderen Werken entnommen sind, wurden unter Angabe der Quellen als Entlehnung kenntlich gemacht. Dies trifft besonders auch auf Quellen aus dem Internet zu. Gleichzeitig geben wir das Einverständnis, unsere Arbeit mittels einer Plagiatssoftware durch die Schule überprüfen zu lassen.

Jena, 17. September 2017

Philip Geißler .....
Joe Schaller .....
Alexander von Mach .....