

QCI

Quantum Computing Interactive

Projektarbeit

Simulation von Quantencomputern

Marc Rochel

Dezember 2002

## Inhaltsverzeichnis

1	Einleitung	2
2	Anforderungsspezifikation	3
2.1	Zustände, der Fluch der Dimensionen	3
2.2	Visualisierung	3
2.3	Interaktivität	4
3	Systementwurf	5
3.1	Klasse Complex	7
3.2	Struktur SDisplay	8
3.3	Klasse CSegmentation	8
3.4	Klasse Register	8
3.5	Klasse CQCIDlg	9
3.6	CRegistergraphDlg	10
3.7	CProbabilityDlg	11
3.8	CGatesDlg	11
4	Detailentwurf der Quantengatter	12
4.1	Basisgatter	13
4.1.1	CNot-Gatter	13
4.1.2	Walsh-Hadamard	14
4.1.3	Walsh-Hadamard-Range-Gatter	16
4.1.4	Walsh-Hadamard-Controlled-Gatter	18
4.1.5	Pi-Achtel-Gatter	20
4.2	Orakel-Gatter	22
4.2.1	Phasen-Orakel-Gatter	23
4.2.2	Bit-Orakel-Gatter	24
4.3	Grover	26
4.3.1	Groverstep-Gatter	26
4.3.2	Grover-Gatter	27
4.4	Shor	28
4.4.1	Bitexchange-Gatter	29
4.4.2	Power-Gatter	31
4.4.3	Rotate two bits controlled-Gatter	32
4.4.4	Inverse Fast Fourier Transformation-Gatter	34
4.4.5	Contineous Fractions-Gatter	36
4.4.6	Shor-Gatter	40
4.4.7	Shor-Begin-Gatter	41
5	Literaturangaben	45

# 1 Einleitung

Bei dem Gedanken, einen Quantencomputer zu simulieren, stellt sich zunächst die Frage, was denn die Zielsetzung eines solchen Simulators sein soll.

Eine Möglichkeit wäre die einfachere Entwicklung neuer Quantenalgorithmen. Einerseits bietet ein Quantencomputersimulator für diesen Zweck momentan den Vorteil, dass man ihn bereits jetzt benutzen kann, wohingegen ein echter Quantencomputer zum Zeitpunkt des Schreibens nur in Labors experimentell existieren, praktisch also noch nicht einsetzbar sind. Zum anderen bietet ein Simulator auch Vorteile bei der Entwicklung von Programmen, die ein echter Quantencomputer nicht mit sich bringt: Die Möglichkeit Programme zu debuggen. Wo auf klassischen Computer ohne Probleme ein laufendes Programm angehalten und der aktuelle Zustand betrachtet werden kann und somit leicht Fehler auffindbar sind, so ist es bei einem Quantencomputer bekanntlich nicht möglich den aktuellen Zustand zu erfahren. Läuft das in der Entwicklung befindliche Programm jedoch auf einem Simulator, so wird dies möglich. Für die Implementierung der bisher bekannten Algorithmen ist dies sicherlich nicht von Nöten, werden die Algorithmen jedoch umfangreicher so könnte dieser Vorteil jedoch von erheblicher Praxisrelevanz sein. Das Problem das sich allerdings bei dieser Zielsetzung eines Quantencomputers stellt, ist jedoch die exponentielle Laufzeit sowie die exponentielle Speicheranforderung die die Simulation eines Quantencomputers auf klassischen Computern fordert. Aufgrund dessen ist solch eine Simulation sehr beschränkt in der Anzahl der simulierten Qubits.

Im Internet stehen einige Quantencomputersimulatoren die diese Zielsetzung verfolgen zur Verfügung. Sie erlauben die Eingabe eines Algorithmuses, entweder in einer eigens definierten Quantencomputersprache oder grafisch, das schrittweise Ausführen und Betrachten des aktuellen Zustandes [4].

Die Zielsetzung des in dieser Projektarbeit entwickelten Quantencomputersimulators ist jedoch eine Andere. Die Arbeitsweise eines Quantencomputers unterscheidet sich grundlegend von der klassischer Computer. Der Mensch versteht schwere Zusammenhänge leichter und schneller wenn sie ihm in aufbereiteter Form visualisiert werden. Dadurch ergeben sich auch andere Denkweisen und Anschauungen der Probleme, und so können neue Ideen und Ansätze für neue Algorithmen entstehen. Genau dies ist die Zielsetzung des hier entwickelten Simulators.

## 2 Anforderungsspezifikation

Im Folgenden wird beschrieben, wie die anvisierte Zielsetzung erreicht werden soll.

### 2.1 Zustände, der Fluch der Dimensionen

Der Fluch der Dimensionen beschreibt im ursprünglichen Sinne den exponentiellen Anstieg von Ressourcen auf klassischen Computern, also z.B. von Rechenzeit oder Speicher, im Bezug auf die Anzahl der Dimensionen die benötigt werden um ein Problem zu lösen. Da durch hinzufügen eines Qubits zu einem Quantensystem die Dimensionsanzahl des Hilbertraums linear erhöht wird, handelt es sich bei der Simulation um solch ein Problem.

Im Bezug auf meine Zielsetzung ergibt sich jedoch ein zusätzliches Problem: Der User muss die Möglichkeit haben die entstehenden komplexen Zustände zu betrachten. Die Ressource „Ausgabefläche“ ist leider auch eine beschränkte Ressource eines Computers, bzw. die Möglichkeiten des Menschen, exponentiell wachsende Zahlenmenge zu überschauen, ist genauso beschränkt.

Damit also der User den aktuellen Zustand analysieren kann, müssen ihm Möglichkeiten zur Sortierung und zum Filtern von Teilzuständen eines Zustandes zur Verfügung stehen.

### 2.2 Visualisierung

Um eine Vorstellung komplexer Algorithmen zu bekommen, ist es recht umständlich, die einzelnen Zwischenzustände als Wertefolge zu analysieren. Dies kann wesentlich erleichtert werden, indem die Zustände visuell aufbereitet werden. Auf diesem Weg können besondere Werte direkt erkannt werden und man bekommt eine bildliche Vorstellung von dem was passiert. Aus psychologischer Sicht ist dies für den Prozess des Verstehens sicherlich von Vorteil, denkt der Mensch doch in Bildern.

Die Visualisierung dieser Implementation des Zustandes

$$\frac{1}{2}|0\rangle + \frac{1}{\sqrt{8}}(1+i)|2\rangle - \frac{1}{2}(1+i)|3\rangle \quad (1)$$

eines 2-Qubitsystems sieht wie folgt aus:

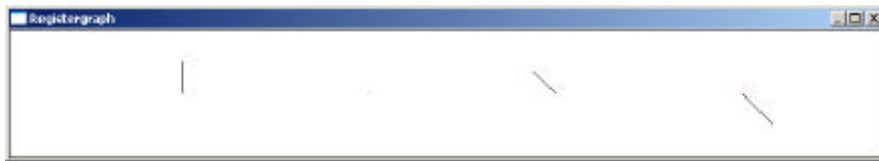


Abb 1: Visualisierung eines Zustandes

Von links nach rechts werden die komplexen Koordinaten des Zustandes im Hilbertraum aufgetragen. Dabei ist das komplexe Koordinatensystem von der typischen Darstellungsweise um  $90^\circ$  gedreht, also ist die positive reelle Achse nach oben gerichtet und die positive imaginäre Achse nach links. Siehe dazu Abbildung 2. Die obere Kante stellt den Realwert 1 dar, die untere Kante  $-1$ . Die Ursprünge der einzelnen komplexen Koordinatensysteme der Koordinaten des Zustandes im Hilbertraum liegen auf einer horizontalen Geraden in der Mitte der beiden Kanten. Die Drehung hat den Zweck die horizontale Bauweise heutiger Monitore besser auszunutzen. Da ferner z.B. bei dem Algorithmus von Grover der imaginäre Teil nicht benutzt wird, erhält man auf diese Weise keine horizontale Linie und die einzelnen Koordinaten sind gut zu erkennen.

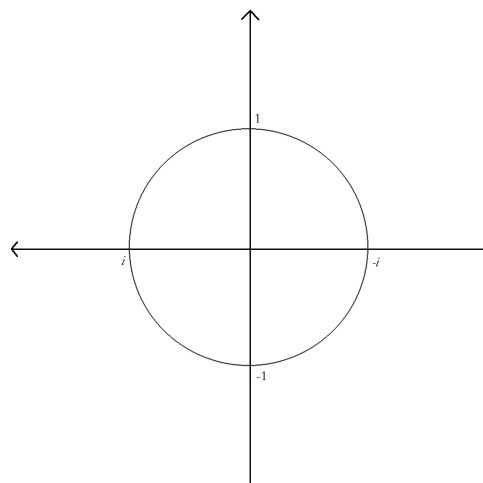


Abb 2: komplexes Koordinatensystem einer Koordinate eines Zustandes im Hilbertraum

### 2.3 Interaktivität

Ein weiterer wichtiger Punkt bei der Konstruktion eines Quantencomputersimulators mit obiger Zielsetzung ist, dem Benutzer die Möglichkeit zur Interaktivität zu bieten, sprich die Möglichkeit zu bieten, schnell mehrere Effekte auf den aktuellen Zustand durch Anwendung verschiedener Gatter ausprobieren zu können. Kombiniert mit der zuvor beschriebenen Visualisierung kann der Benutzer eine intuitive Vorstellung davon bekommen, warum bei einem Algorithmus an bestimmten Stellen genau die

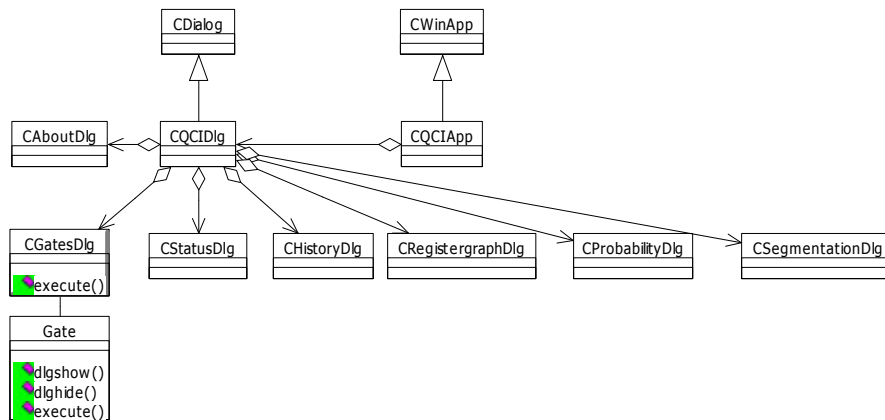
entsprechenden Gatter angewandt werden und andere die Funktionsweise des Algorithmus zerstören.

Der im Rahmen dieser Projektarbeit entwickelte Quantencomputersimulator bietet Interaktivität als zentrales Element. Befindet sich der Simulator in einem bestimmten Zustand, kann der Benutzer durch Auswählen eines der zur Verfügung stehenden Gatter dieses auf den Zustand anwenden. Zusätzlich steht eine History der angewandten Gatter zur Verfügung, ähnlich der History gängiger Webbrowser. In der History kann nachgeschaut werden wie der aktuelle Zustand entstanden ist und auch beliebig zu vorherigen Zuständen zurückgeblättert werden um z.B. von dort aus die Effekte anderer Gatter zu untersuchen. Ferner ist die History hierarchisch organisiert. Komplexere, öfter benutzte Folgen von Gattern können zu neuen Gattern zusammengefasst werden. Bei Anwendung solch einer Folge von Gattern werden die einzelnen Gatter aus denen die Folge besteht in einer Hierarchieebene tiefer dargestellt. Durch beschränken der Darstellungstiefe der History kann somit auch bei umfangreichen Algorithmen der Überblick erhalten bleiben.

### **3 Systementwurf**

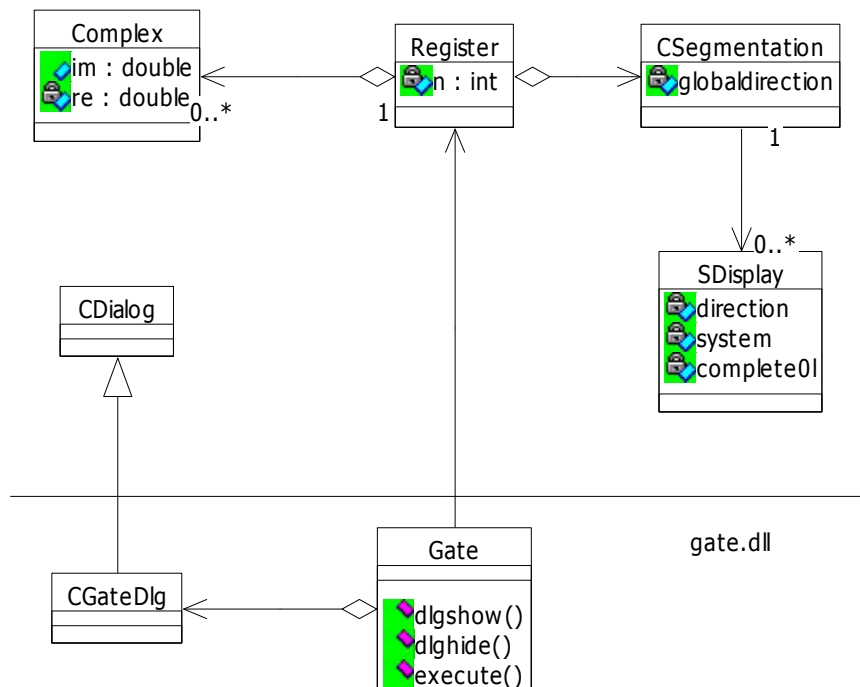
Der Quantencomputersimulator wird implementiert in C++. Zielplattform ist Windows 32 Bit. Zur Implementierung der GUI (Graphical User Interface) werden die MFC (Microsoft Foundation Classes) benutzt.

Die Übersicht der Klassenstruktur:



Gemäß den MFC wird die Hauptklasse des Programms CQCIApp von CWinApp abgeleitet. CQCIApp besitzt eine Klasse CQCIDlg, die von CDialog erbt und das Hauptfenster der Applikation, den oberen Kontrollbalken, darstellt. Diese Klasse besitzt die Klassen der Applikation, die den anderen Anwendungsfenstern hinterlegt sind: CRegistergraphDlg, CProbabilityDlg, CHistoryDlg, CStatusDlg, CHistoryDlg, CGatesDlg, CAboutDlg, CSegmentationDlg. Diese Klassen erben auch alle von CDialog.

Die Implementation der einzelnen Quantengatter erfolgt durch DLLs (Dynamic Link Libraries) die die Windows-Plattform zur Verfügung stellt. Dies ermöglicht die Implementierung neuer Gatter unabhängig von dem Quellcode des eigentlichen Simulators. Dieser muss nicht neu kompiliert werden, wird ein neues Gatter implementiert. CGatesDlg kontrolliert dabei das Nachladen der jeweils benötigten DLL.



Das Quantenregister wird implementiert durch die Klasse Register. Sie besitzt  $2^n$  Complex-Klassen um den aktuellen Zustand zu speichern. Die Klasse Complex speichert eine komplexe Zahl in zwei Variablen vom Typ double, eine für den Realanteil, die andere für den Imaginäranteil. Die Klasse Register besitzt ferner CSegmentation, die die Gruppierung mehrere Qubits des Registers zu logischen Variablen speichert. Dazu besitzt CSegmentation für jede solche logische Variable noch eine Klasse SDisplay, die die gewünschte Darstellung dieser Variable bei der Ausgabe speichert.

Jede Quantengatter-DLL enthält globale Funktionen zur Darstellung ihres Fensters und zum Ausführen des Gatters. Dazu enthält sie eine Klasse CGateDlg, die von CDialog erbt.

Im Folgenden wird auf die wichtigen Details der Implementierung dieser Klassen eingegangen.

### 3.1 Klasse Complex

Die Klasse Complex stellt eine komplexe Zahl dar. Sie besitzt zwei Attribute vom Typ double; eins für den Realanteil der Zahl, „re“, eins für den Imaginärteil der Zahl, „im“. Die Wahl fiel auf double, da dieser Datentyp eine höhere Genauigkeit bietet als float, und der halb so große Speicherbedarf nur ein Qubit mehr ermöglichen würde. Siehe dazu auch die Beschreibung der

Klasse Register. Da die Größe eines doubles 8 Byte ist, benötigt also eine Instanz von Complex 16 Byte.

Die Klasse Complex besitzt überladene Operatoren für arithmetische Standardoperationen sowie eine Methode zur Berechnung der L2-Norm und eine Methode zur Berechnung der e-Funktion. Ferner eine Methode tostring zur formatierten Ausgabe der Zahl als String.

### 3.2 Struktur SDisplay

Die Struktur SDisplay beschreibt die Darstellung einer logischen Gruppe von Qubits. Sie besitzt 3 Attribute. „direction“ vom Typ char gibt an in welcher Reihenfolge die Ziffern dargestellt werden sollen. „system“ beschreibt die Basis in der die Darstellung erfolgt. „complete0“ gibt an, ob führende Nullen angezeigt werden sollen oder nicht. Somit kann entweder eine platzsparende Ausgabe oder auch eine Ausgabe, bei der alle Werte gleich viele Zeichen besitzen, realisiert werden.

### 3.3 Klasse CSegmentation

Die Klasse CSegmentation speichert die Einteilung eines Registers in logische Segmente. Das Attribut globaldirection speichert in welcher Reihenfolge die Segmente dargestellt werden. Für jedes Segment besitzt sie ein Attribut vom Typ SDisplay welches die lokale Reihenfolge und die lokale Basis speichert sowie ob führende Nullen ergänzt werden sollen.

### 3.4 Klasse Register

Die Klasse Register repräsentiert den Zustand eines Quantencomputers. Zur Implementation gibt es zwei grundsätzliche Ansätze: Eine Implementation als Linked List erscheint zunächst als speichersparend, da nur die Werte in einer Superposition gespeichert werden, die von einem Normalwert, praktischerweise 0, abweichen. Da allerdings die gängigen Algorithmen wie z.B. Shor und Grover direkt zu Anfang auf alle Qubits Walsh-Hadamard anwenden, steigt der Speicheraufwand leicht auf den worst case:

$$2^N \cdot (\text{sizeof}(\text{complex}) + \text{sizeof}(\text{pointer})) \quad (2)$$

wobei N die Anzahl der Qubits ist. sizeof(pointer) ist auf einem 32 Bit System 4 Byte. Diese 4 Bytes fallen weg, wenn man zur Implementation ein Array benutzt. Dies erscheint auf den ersten Blick nicht optimal, da der best case bereits

$$2^N \cdot \text{sizeof}(\text{complex}) \quad (3)$$

beträgt. Jedoch ist dieser Wert konstant und auch gleichzeitig der worst case, da die Zeiger auf das jeweils nächste Element der Liste entfallen.

Die hier vorgestellte Implementation verwendet also die Variante mit einem Array. Der maximal adressierbare Speicherbereich auf einem 32 Bit System beträgt  $2^{32}$  Byte. Da eine Instanz vom Typ Complex 16 Byte =  $2^4$  Byte benötigt, beträgt die theoretische obere Grenze an simulierbaren Qubits 28. Da ein Programm allerdings noch zusätzlich Speicher benötigt für z.B. Stack, Zwischenvariablen, grafische Ressourcen, ist diese Obergrenze nochmals um 1 Qubit zu reduzieren und somit maximal 27 Qubits. Diese Berechnung gilt jedoch praktisch nur dann, besitzt das System 4 Gigabyte RAM. Besitzt ein System weniger RAM ist dies bei der Berechnung zu beachten, wobei der benötigte Speicher sich wie folgt berechnen lässt:

$$2^N \cdot (\text{sizeof}(\text{complex})) \quad (4)$$

Sollte dieser Speicher nicht zur Verfügung stehen und das System anfangen zu swappen, ist mit erheblichen Geschwindigkeitseinbussen der Simulation zu rechnen, da auf die einzelnen Werte des Arrays aufgrund der Funktionsweise einiger Quantengatter nicht linear zugegriffen wird.

Eine Instanz von Register besitzt also ein Array von  $2^N$  Instanzen von Complex das den Zustand repräsentiert.

Da der hier vorgestellte Quantencomputersimulator über eine Historyfunktion verfügt, wächst der Speicherbedarf zudem linear in der Anzahl der Gatter die angewandt werden, wodurch sich ein Speicherbedarf von insgesamt

$$2^N \cdot (\text{sizeof}(\text{complex})) \cdot \text{steps} \quad (5)$$

ergibt. Durch die History verringert sich die Anzahl der simulierbaren Qubits auf einem gegebenen System. Jedoch wurde an dieser Stelle unter der Berücksichtigung der Zielsetzung des Simulators und des Benutzerkomforts auf die History nicht verzichtet.

### 3.5 Klasse CQCIDlg

Die Klasse CQCIDlg besitzt eine Methode OnButtonmeasure die auf Wunsch des Benutzers durch Anklicken des entsprechenden Buttons ausgeführt wird. Sie simuliert die Messung des aktuellen Zustandes.

```
double z=((double)rand())/RAND_MAX;
unsigned int i=0;
while ((z>0) && (i<(1UL<<r->n)))
```

```

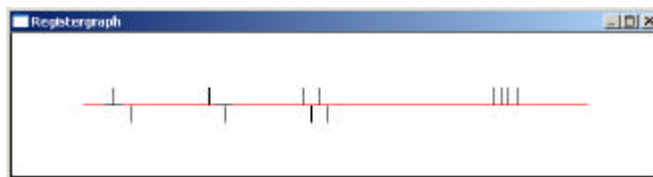
{
    z--r->p[i].L2Norm2();
    i++;
}
i--;

```

Zuerst wird ein zufälliger Wert  $z$  zwischen 0 und 1 generiert. In der folgenden while-Schleife wird derjenige Wert  $i$  gesucht so dass die Verteilungsfunktion an der Stelle  $i$   $z$  ist. Die einzelnen Wahrscheinlichkeiten ergeben sich aus dem Quadrat der Beträge der einzelnen Koeffizienten des aktuellen Zustandes. Dieses geschieht durch die Methode `L2Norm2` der Klasse `Complex`. Dieses entspricht der Umkehrung der Verteilungsfunktion und somit ist  $i$  der simulierte gemessene Wert. Dieser Algorithmus läuft offensichtlich in  $O(2^N)$

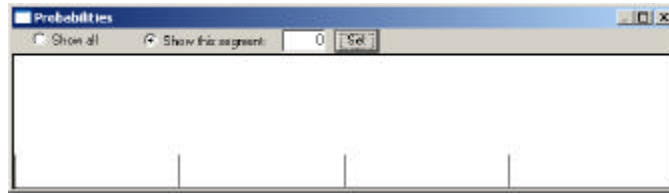
Dieses Verfahren kann noch optimiert werden, sollen mehrere Messungen eines Zustandes durchgeführt werden. Eine einmalige Invertierung der Verteilungsfunktion benötigt  $O(2^N)$  Schritte. Anschließend kann allerdings jede simulierte Messung durch Anwendung binärer Suche in  $O(N)$  Schritten durchgeführt werden. Von der Implementation dieser Optimierung wurde jedoch abgesehen. Erstens, da eine wiederholte Messung eines Zustandes nicht sonderlich oft vorkommt, viel mehr untersucht man lieber den Zustand des Systems. Zweitens, da der obige Algorithmus bei der Anzahl der möglichen Qubits auf heutigen Computern sowieso schnell genug ist, wie durch simples Ausprobieren überprüft wurde. Drittens, was der ausschlaggebende Punkt ist: Die Berechnung der Inversen der Verteilungsfunktion benötigt nicht nur Zeit, sondern auch Speicher, und dieser ist bei der Simulation eines Quantencomputers sowieso sehr knapp.

### 3.6 CRegistergraphDlg



Diese Klasse visualisiert wie zuvor beschrieben den aktuellen Zustand des Quantenregisters.

### 3.7 CProbabilityDlg



Diese Klasse visualisiert die Wahrscheinlichkeiten mit denen die möglichen Messwerte gemessen werden. Ähnlich wie bei der Visualisierung des Zustandes werden dabei die Wahrscheinlichkeiten auf einer imaginären vertikalen Achse als Linien abgetragen. Die Ursprünge der Achsen der Wahrscheinlichkeiten der einzelnen Messwerte werden dabei in aufsteigender Reihenfolge horizontal nebeneinander positioniert. Bei der Berechnung der Wahrscheinlichkeiten wird das ausgewählte Segment berücksichtigt.

### 3.8 CGatesDlg

Diese Klasse besitzt ein CTreeCtrl um die zur Verfügung stehenden Quantengatter darzustellen. Der Tree stellt die Verzeichnisstruktur im Unterverzeichnis „gates\“ dar. Durch Unterverzeichnisse kann somit Struktur in die zur Verfügung stehenden Gatter gebracht werden. Durch Auswahl eines der Gatter wird die entsprechende DLL des Gatters geladen. Dies geschieht in der Methode OnSelchangedTree.

Eine Gate-DLL muss Folgendes exportieren:

- Eine Funktion void dlgshow(int x, int y, int cx, int cy) die ein Fenster mit Einstellmöglichkeiten des Gatters anzeigen sollte. Dabei sollte das Fenster an Bildschirmposition (x, y) mit Größe (cx, cy) erscheinen.
- Eine Funktion void dlghide() die das Fenster schließt.
- Eine Funktion void execute(unsigned int param) die das Gatter ausführt, auch wenn das Fenster nicht dargestellt wird. Die dadurch wegfallenden Einstellmöglichkeiten werden in param übergeben. Vorzugsweise sollte diese Funktion intern von dem Gatter aufgerufen werden wenn der Benutzer durch das Fenster des Gatters seine Funktion ausführt.

Der Simulator stellt seinerseits folgendes der DLL zur Verfügung:

- Eine Funktion void execute(const char \*gate, unsigned int param) die die Möglichkeit bietet, andere Gatter aus einem Gatter heraus aufzurufen. Der erste Parameter gibt den Namen des aufzurufenden Gatters an. Der zweite Parameter wird an die

execute-Funktion der aufzurufenden DLL weitergereicht. Diese Funktion ermöglicht direkt die hierarchische Gliederung der Gatter in der History.

- Eine Funktion void step(const char \*s) die einen Schritt in der History einfügt mit dem übergebenen String als anzuzeigenden Text. Empfehlenswert ist, dies am Ende der execute-Funktion auszuführen um die Änderungen des Zustandes als Schritt in der History darzustellen.
- Einen Zeiger Register \*r auf die Instanz der Klasse Register die den zu verändernden Zustand beschreibt.
- Das Handle auf die geladene DLL: HMODULE hDll.

Die .def-Datei einer Gate-DLL sollte also wie folgt aussehen:

```
LIBRARY gate

EXPORTS
    execute                @1
    dlgshow                @2
    dlghide                @3
    procexecute            @10
    procstep               @11
    r                      @12
    hDll                   @13
```

Wählt der Benutzer ein Gatter aus der Liste der zur Verfügung stehenden Gatter aus, so wird zunächst dlghide von der aktuellen Gatter-DLL aufgerufen und dann die DLL entladen. Anschließend wird die neu ausgewählte geladen und dlgshow aufgerufen.

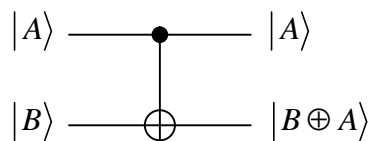
## 4 Detailentwurf der Quantengatter

Quantengatter lassen sich generell alle mittels Matrixmultiplikation implementieren. Dabei wird eine  $2^N \times 2^N$  Matrix mit einem  $2^N$  Vektor multipliziert. Dieses ist allerdings recht uneffizient, da  $O(2^{2N})$  komplexe Multiplikationen benötigt werden. Einige Gatter können jedoch effizienter simuliert werden. Darauf wird im Folgenden im Detail eingegangen. Dabei wird speziell die execute-Funktion der einzelnen DLLs besprochen, da diese die Funktionalität des Gatters definieren.

Im Folgenden werden die Koordinaten eines Zustandes im mehrdimensionalen Hilbertraum auch mit Koeffizienten eines Zustandes bezeichnet.

## 4.1 Basisgatter

### 4.1.1 CNot-Gatter



Dieses Gatter entspricht der unitären Matrix

$$U_{CN} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad (6)$$

Das erste Bit kontrolliert also ob das zweite Bit negiert wird.

Implementiert werden kann dieses Gatter schneller als in  $O(2^{2N})$  wie folgt:

```
void __stdcall execute(unsigned int param)
{
    unsigned int j;
    unsigned int a=((unsigned int *)param)[0];
    unsigned int b=((unsigned int *)param)[1];

    Register t;
    t.SetBitn(r->n);
    for (j=0;j<(1UL<<r->n);j++)
    {
        // Controlbit gesetzt?
        if ((j&(1<<a))==0)
        {
            // Nein
            t.p[j]=r->p[j];
        }
        else
        {
            // Ja
            t.p[j^(1<<b)]=r->p[j];
        }
    }
    r->copyfrom(&t);

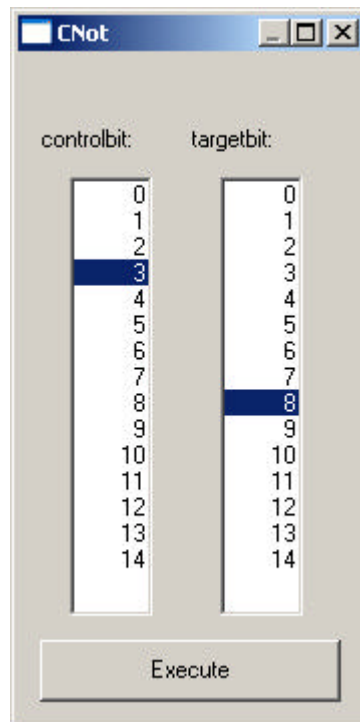
    char buf[128];
    sprintf(buf,"cnot ( %i, %i )",a,b);
    procstep(buf);
}
```

}

Die Variable  $a$  bezeichnet die Position des kontrollierenden Bits,  $b$  die Position des evtl. zu negierenden Zielbits im Register. Es wird ein neues Register  $t$  leer initialisiert. In der for-Schleife wird nun das Original-Register  $r$  durchgegangen. Wird ein Wert gefunden, der das Kontrollbit nicht gesetzt hat, so wird dessen Koeffizient unverändert in  $t$  übernommen. Wird ein Wert gefunden, der das Kontrollbit gesetzt hat, so wird sein Koeffizient dem Wert in  $t$  zugewiesen, der bist auf das Zielbit exakt mit dem Wert übereinstimmt.

Dieser Algorithmus hat eine Laufzeit von  $O(2^N)$  und ist somit linear in der Anzahl der Koeffizienten und effizienter als die Durchführung einer Matrixmultiplikation. Bis auf konstante Faktoren lässt sich dieses Ergebnis auch nicht weiter optimieren, da die Hälfte aller Koeffizienten getauscht werden müssen, was also auch in  $O(2^{N-1}) = O(2^N)$  liegt.

Das Fenster des Gatters sieht wie folgt aus:



In der Linken Spalte kann der Benutzer das Kontrollbit auswählen und in der rechten das Zielbit. Durch Anklicken von Execute wird das Gatter auf das Register angewandt.

#### 4.1.2 Walsh-Hadamard

$$a|0\rangle + b|1\rangle \longrightarrow \boxed{H} \longrightarrow a \frac{|0\rangle + |1\rangle}{\sqrt{2}} + b \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

Dieses Gatter entspricht der unitären Matrix

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \quad (7)$$

Die Implementation sieht wie folgt aus:

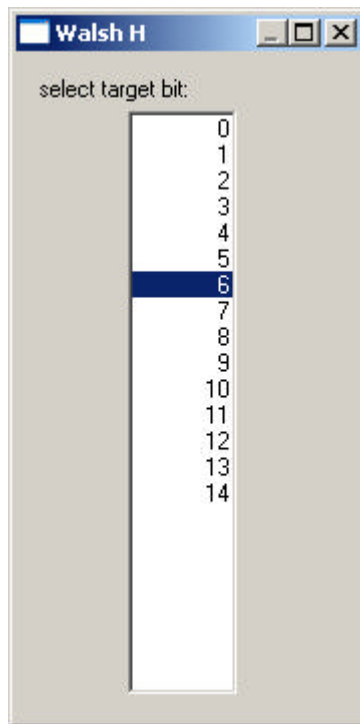
```
void __stdcall execute(unsigned int param)
{
    int i=param;
    int j;
    Register t;
    t.SetBitn(r->n);
    for (j=0;j<(1<<r->n);j++)
    {
        // wenn Bit nicht gesetzt ist
        if ((j&(1<<i))==0)
        {
            t.p[j]+=r->p[j]*(1.0/sqrt(2));
            t.p[j+(1<<i)]+=r->p[j]*(1.0/sqrt(2));
        }
        // wenn Bit gesetzt ist
        else
        {
            t.p[j-(1<<i)]+=r->p[j]*(1.0/sqrt(2));
            t.p[j]-=r->p[j]*(1.0/sqrt(2));
        }
    }
    r->copyfrom(&t);

    char buf[32];
    sprintf(buf,"Walsh ( %i )",param);
    procstep(buf);
}
```

Die Variable  $i$  bezeichnet das Bit, auf das das Walsh-Hadamard-Gatter angewandt werden soll. Ein neues Register  $t$  wird leer initialisiert. Das Register  $r$  wird in einer for-Schleife durchlaufen. Wird ein Wert gefunden, an dem Bit  $i$  nicht gesetzt ist, so wird in  $t$  zu dem Koeffizienten  $k$  des Wertes, als auch zu dem Koeffizienten des Wertes mit gesetztem Bit  $i$ ,  $1/\sqrt{2} \cdot k$  addiert. Dies entspricht der Ausmultiplikation der oberen Zeile der Matrix  $H$ . Wird ein Wert gefunden, dessen Bit  $i$  gesetzt ist, so wird in  $t$  zu dem Koeffizienten  $k$  des Wertes  $-1/\sqrt{2} \cdot k$  addiert und zu dem Koeffizienten des bis auf das  $i$ -te Bit gleichen Wertes  $1/\sqrt{2} \cdot k$  addiert. Dies entspricht der Ausmultiplikation der unteren Zeile der Matrix  $H$ .

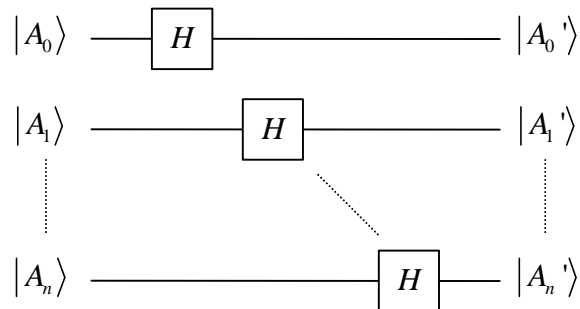
Dieser Algorithmus hat offensichtlich auch eine Laufzeit von  $O(2^N)$  Schritten. Bis auf konstante Faktoren ist dieser bereits optimal, da prinzipiell alle Koeffizienten durch das Walsh-Hadamard-Gatter beeinflusst werden können.

Das Fenster des Gatters sieht wie folgt aus:



Durch anklicken eines Bits aus der Liste wird das Walsh-Hadamard-Gatter auf dieses Bit angewandt.

#### 4.1.3 Walsh-Hadamard-Range-Gatter



Dieses Gatter entspricht der unitären Matrix

$$WR = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} & 0 & \dots & 0 \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} & 0 & \dots & \dots \\ 0 & 0 & \dots & 0 & 0 \\ \dots & \dots & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & \dots & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \quad (8)$$

Dieses Gatter entspricht der Anwendung des Walsh-Hadamard-Gatters auf einen Bereich von Qubits aus dem Register.

Die Implementation sieht wie folgt aus:

```
void __stdcall execute(unsigned int param)
{
    unsigned int i;
    unsigned int from=((unsigned int *)param)[0];
    unsigned int to=((unsigned int *)param)[1];
    if (from<=to)
    {
        for (i=from;i<=to;i++)
            procexecute("basic\\walsh",i);

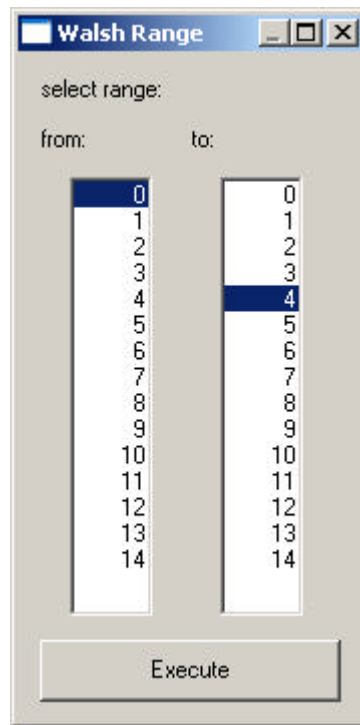
        char buf[128];
        sprintf(buf,"Walshrange ( %i - %i )",from,to);
        procstep(buf);
    }
}
```

Die Variablen from, bzw. to bezeichnen den Beginn, bzw. das Ende des Bereichs an Qubits im Register auf die das Walsh-Hadamard-Gatter angewandt werden soll. Dieses Gatter stellt kein elementar neues Gatter dar. Um dies zu verdeutlichen, wird bei der Implementation die Möglichkeit genutzt, auf bereits implementierte andere Gatter zugreifen zu können. Somit wird einfach der Bereich an Zielqubits in einer for-Schleife durchlaufen und jedes Mal das Walsh-Hadamard-Gatter aufgerufen (s. procexecute). Dies wird bei Anwendung des Gatters dem Benutzer direkt verdeutlicht, durch die zuvor erwähnte hierarchische History-Funktionalität.

Somit besitzt dieser Algorithmus eine Laufzeitkomplexität von  $O(m \cdot 2^N)$ , wobei  $m=to-from+1$ , also die Anzahl der auszuführenden Walsh-Hadamard-Gatter. Im worst-case, wenn also auf alle Qubits Walsh-Hadamard angewandt werden soll, ist  $m=N$ . Damit liegt der Algorithmus dann in  $O(N \cdot 2^N)$  und ist immer noch effizienter als eine vollständige Matrixmultiplikation. Die Frage bleibt jedoch, ob nicht durch die Verwendung des zuvor implementierten Walsh-Hadamard-Gatters Optimierungspotential verloren geht. Die momentane Implementation dieses Walsh-Hadamard-Range-Gatters besteht aus zwei verschachtelte for-Schleifen. Die äußere, hier im Quellcode sichtbare,

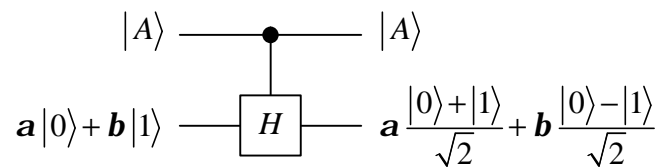
wird  $m$  Mal durchlaufen, die innere, im Walsh-Hadamard-Gatter implementierte, jeweils  $2^N$  Mal. Ein Verzicht auf die hierarchische Gliederung würde die Möglichkeit bieten die beiden Schleifen zu vertauschen, jedoch würde dies die Laufzeitkomplexität nicht verändern.

Das Fenster des Gatters sieht wie folgt aus:



In der linken Spalte kann das Startbit, in der rechten Spalte das Endbit des Bereichs angegeben werden, auf den das Walsh-Hadamard-Range-Gatter angewandt werden soll. Durch Anklicken von Execute wird die Transformation ausgeführt.

#### 4.1.4 Walsh-Hadamard-Controlled-Gatter



Dieses Gatter entspricht der unitären Matrix

$$WC = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ 0 & 0 & \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix} \quad (9)$$

Wobei das erste Bit kontrolliert ob auf das zweite Bit Walsh-Hadamard angewandt wird.

Implementiert wurde dieses Gatter analog zu Walsh-Hadamard:

```
void __stdcall execute(unsigned int param)
{
    unsigned int j;
    unsigned int a=((unsigned int *)param)[0];
    unsigned int b=((unsigned int *)param)[1];

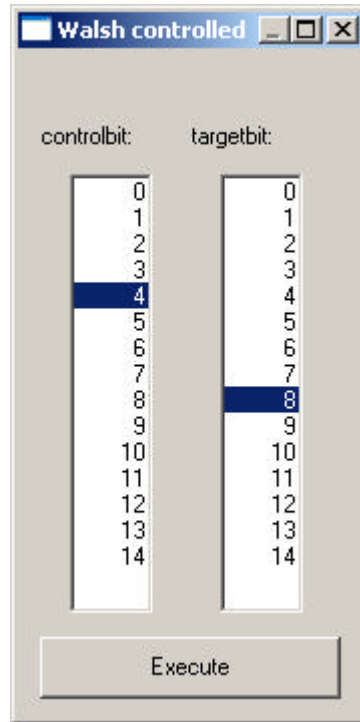
    Register t;
    t.SetBitn(r->n);
    for (j=0;j<(1UL<<r->n);j++)
    {
        if ((j&(1<<a))==0)
        {
            t.p[j]+=r->p[j];
        }
        else
        {
            // wenn bit nicht gesetzt ist
            if ((j&(1<<b))==0)
            {
                t.p[j]+=r->p[j]*(1.0/sqrt(2));
                t.p[j+(1<<b)]+=r->p[j]*(1.0/sqrt(2));
            }
            // wenn bit gesetzt ist
            else
            {
                t.p[j-(1<<b)]+=r->p[j]*(1.0/sqrt(2));
                t.p[j]-=r->p[j]*(1.0/sqrt(2));
            }
        }
    }
    r->copyfrom(&t);

    char buf[128];
    sprintf(buf,"Walsh controlled ( %i, %i )",a,b);
    procstep(buf);
}
```

Zu Beachten ist, dass hier nun a das Kontrollbit enthält und b das Zielbit, auf das die Transformation gegebenenfalls angewandt wird. Die Implementation unterscheidet sich ansonsten nur durch die zusätzliche Abfrage, ob das Kontrollbit gesetzt ist. Ist es nicht gesetzt, wird zu dem Koeffizient des aktuellen Wertes der aktuelle Koeffizient addiert. Ansonsten wird verfahren wie im Fall des gewöhnlichen Walsh-Hadamard-Gatters.

Die Laufzeitkomplexität verändert sich durch die zusätzliche Fallunterscheidung nicht. Sie ist immer noch Element von  $O(2^N)$

Das Fenster des Gatters sieht wie folgt aus:



In der linken Spalte kann das Kontrollbit, in der rechten Spalte das Zielbit gewählt werden. Durch Anklicken von Execute wird die Transformation ausgeführt.

#### 4.1.5 Pi-Achtel-Gatter

$$\mathbf{a}|0\rangle + \mathbf{b}|1\rangle \text{ --- } \boxed{T} \text{ --- } \mathbf{a}|0\rangle + \mathbf{b}e^{\frac{i\pi}{4}}|1\rangle$$

Dieses Gatter entspricht der unitären Matrix

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{pmatrix} \quad (10)$$

Implementiert wird dieses Gatter wie folgt:

```
void __stdcall execute(unsigned int param)
{
```

```

int i=param;
int j;
Complex faktor;
faktor.set(sqrt(0.5),sqrt(0.5));
for (j=0;j<(1<<r->n);j++)
{
    // wenn Bit nicht gesetzt ist
    if ((j&(1<<i))==0)
    {
    }
    // wenn Bit gesetzt ist
    else
    {
        r->p[j]*=faktor;
    }
}

char buf[32];
sprintf(buf,"Pi-eighth ( %i )",param);
procstep(buf);
}

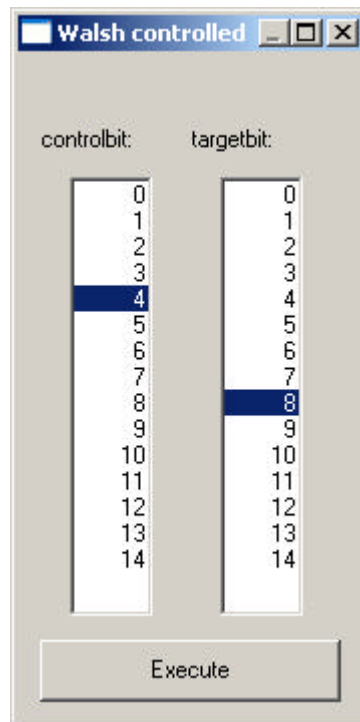
```

Für die Implementation wird kein zusätzliches temporäres Register benötigt, da die Veränderung der einzelnen Koeffizienten nicht auf andere wirkt. Da gilt

$$faktor = e^{\frac{i\pi}{4}} = \frac{1}{\sqrt{2}}(1+i) \quad (11)$$

genügt es, in einer for-Schleife das Register zu durchlaufen und die Koeffizienten, bei dessen Wert das Zielbit gesetzt ist, mit *faktor* zu multiplizieren. Die Laufzeitkomplexität beträgt wieder  $O(2^N)$ .

Das Fenster des Gatters sieht wie folgt aus:



Durch Auswahl eines Bits wird das Pi-Achtel-Gatter auf es angewandt.

## 4.2 Orakel-Gatter

Es ist Fakt, dass jede auf klassischen Computern berechenbare Funktion auch auf einem Quantencomputer in derselben Geschwindigkeit berechnet werden kann. In theoretischen Betrachtungen werden solche Funktionen zu Orakel-Gattern abstrahiert. Sie stehen für die Quantenimplementation der klassischen Funktion.

Da diese Funktionen ja auch ursprünglich klassisch berechenbar sind, bietet es sich an, in der Simulation eines Quantencomputers diese Abstraktion beizubehalten und die Funktion klassisch zu berechnen um Zeit zu sparen im Vergleich zu einer vollständigen Simulation der für den Quantencomputer übersetzten klassischen Funktion.

Der hier vorgestellte Quantencomputersimulator bietet zwei Orakel-Gatter an, die z.B. in der theoretischen Betrachtung des Algorithmus von Grover verwandt werden. Diese Gatter implementieren Funktionen von

$$\{0,1\}^n \rightarrow \{0,1\} \quad (12)$$

### 4.2.1 Phasen-Orakel-Gatter

Implementiert ist dieses Gatter wie folgt:

```
void __stdcall execute(unsigned int param)
{
    unsigned int i,j;
    unsigned int from, to, n;
    unsigned int value;
    from=((unsigned int *)param)[0];
    to=((unsigned int *)param)[1];
    n=((unsigned int *)param)[2];

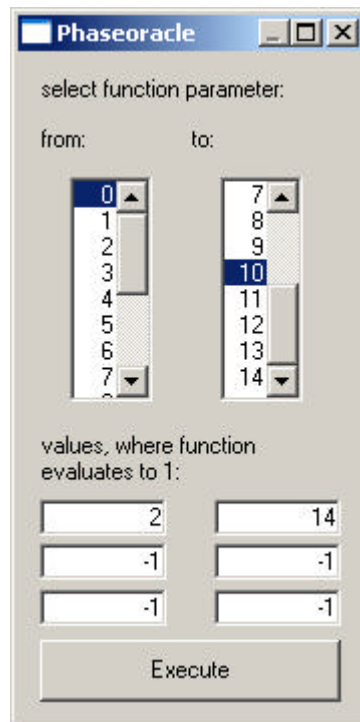
    for (j=0;j<(1UL<<r->n);j++)
    {
        value=j;
        value=value<<(32-to-1);
        value=value>>(32-to-1);
        value=value>>from;
        for (i=0;i<n;i++)
        {
            if (value==((unsigned int*)param)[3+i])
                r->p[j]*=-1;
        }
    }
    procstep("Phaseoracle");
}
```

Die Funktion  $f$  die das Orakel-Gatter darstellt, wird derart in einer Liste  $n$  gespeichert, so dass gilt:

$$\forall a \in \{0,1\}^n. (a \in n \Leftrightarrow f(a) = 1) \quad (13)$$

Alle Koeffizienten der  $a \in n$  werden dann mit  $e^{-ip} = -1$  multipliziert. Der Bereich from-to spezifiziert die Qubits des Parameters von  $f$  im Quantenregister.

Das Fenster des Gatters sieht wie folgt aus:



Im oberen Bereich kann in der linken Spalte das Startbit, in der rechten Spalte das Endbit des Bereichs des Funktionsparameters angegeben werden. Im unteren Bereich können die Werte  $x$  aus dem Definitionsbereich von  $f$  angegeben werden, für die  $f(x)=1$ . Eine -1 bedeutet, dass dieses Eingabefeld nicht beachtet werden soll. Durch Anklicken von Execute wird die Transformation ausgeführt.

#### 4.2.2 Bit-Orakel-Gatter

Die Implementation des Gatters:

```
void __stdcall execute(unsigned int param)
{
    unsigned int i,j;
    unsigned int from, to, n, target;
    unsigned int value;
    from=((unsigned int *)param)[0];
    to=((unsigned int *)param)[1];
    n=((unsigned int *)param)[2];
    target=((unsigned int *)param)[n+3];

    Register t;
    t.SetBitn(r->n);
    bool result1;
    for (j=0;j<(1UL<<r->n);j++)
    {
        value=j;
        value=value<<(32-to-1);
        value=value>>(32-to-1);
```

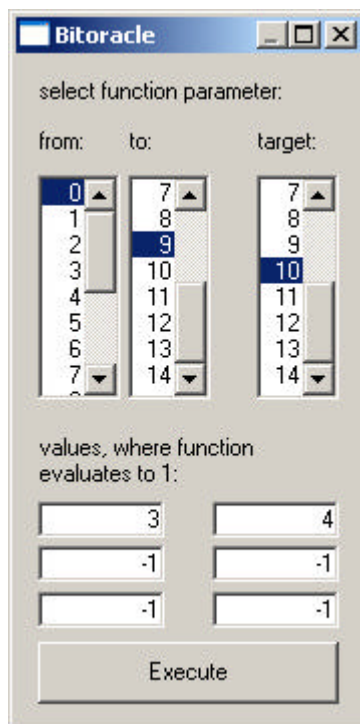
```

value=value>>from;
result1=false;
for (i=0;i<n;i++)
{
    if (value==((unsigned int*)param)[3+i])
    {
        t.p[j^(1UL<<target)]+=r->p[j];
        result1=true;
    }
}
if (!result1)
    t.p[j]+=r->p[j];
}
r->copyfrom(&t);
procstep("Bitoracle");
}

```

Für diese Implementation gilt dasselbe wie für die des Phasenorakels. Nur wird, abhängig vom jeweiligen Funktionswert, ein Zielbit negiert, anstatt die Phase zu drehen.

Das Fenster des Gatters sieht wie folgt aus:



Im oberen Bereich kann in der linken Spalte das Startbit, in der mittleren Spalte das Endbit des Bereichs des Funktionsparameters angegeben werden. In der rechten Spalte wird das Zielbit ausgewählt. Im unteren Bereich können die Werte  $x$  aus dem Definitionsbereich von  $f$  angegeben werden, für die  $f(x)=1$ . Eine -1 bedeutet, dass dieses Eingabefeld nicht beachtet werden soll. Durch Anklicken von Execute wird die Transformation ausgeführt.

## 4.3 Grover

### 4.3.1 Groverstep-Gatter

Dieses Gatter führt eine Iteration des Grover-Algorithmuses durch, das Register muss zuvor in Superposition gebracht werden:

1. Phasenorakel wird ausgeführt.
2. Walshrange wird angewandt.
3. Phase von  $|0\rangle$  wird gedreht.
4. Walshrange wird angewandt.

Punkt 3 kann einfach implementiert werden durch Aufruf des Phasenorakels mit der Funktion  $f$  mit  $f(0)=1$  und  $f(x)=0$  für  $x>1$ .

Äquivalent entspricht eine Iteration des Grover-Algorithmus folgendem Operator:

$$GS = WR \ P(0) \ WR \ P(f) \quad (14)$$

Wobei  $P(f)$  für das Phasenorakel von  $f$  steht und  $P(0)$  für das Phasenorakel der Funktion  $g$  für die nur  $g(0)=1$ .

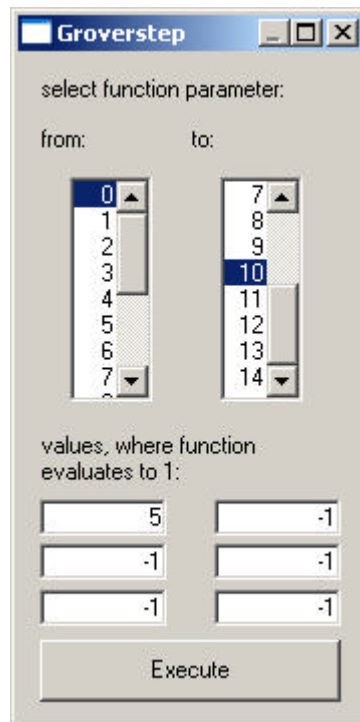
Die Implementation des Gatters:

```
void __stdcall execute(unsigned int param)
{
    unsigned int param0[4];
    param0[0]=((unsigned int*)param)[0];
    param0[1]=((unsigned int*)param)[1];
    param0[2]=1;
    param0[3]=0;

    // Groverstep
    procexecute("oracle\\phaseoracle",param);
    procexecute("basic\\walshrange",param);
    procexecute("oracle\\phaseoracle",(unsigned int)param0);
    procexecute("basic\\walshrange",param);

    char buf[128];
    sprintf(buf,"Groverstep ( %i - %i )",param0[0],param0[1]);
    procstep(buf);
}
```

Das Fenster des Gatters sieht wie folgt aus:



Im oberen Bereich kann in der linken Spalte das Startbit, in der rechten Spalte das Endbit des Bereichs des Funktionsparameters angegeben werden. Im unteren Bereich können die Werte  $x$  aus dem Definitionsbereich von  $f$  angegeben werden, für die  $f(x)=1$ . Diese  $x$  entsprechen den zu suchenden Werten. Eine -1 bedeutet, dass dieses Eingabefeld nicht beachtet werden soll. Durch Anklicken von Execute wird die Transformation ausgeführt.

### 4.3.2 Grover-Gatter

Dieses Gatter implementiert den vollständigen Grover-Algorithmus. Dazu wird zunächst das Register in Superposition gebracht mittels Walshrange-Gatter. Anschließend wird  $k = \left\lceil \frac{p}{4} \sqrt{\frac{2^{to-from+1}}{M}} \right\rceil$  mal Groverstep ausgeführt, wobei  $M$  Anzahl der  $x$  ist, für die  $f(x)=1$ .

Der Groveralgorithmus entspricht also folgendem Operator:

$$G = GS^k WR \quad (15)$$

Die Implementation des Gatters:

```
void __stdcall execute(unsigned int param)
{
    procexecute("basic\\walshrange",param);
    unsigned int i;
```

```

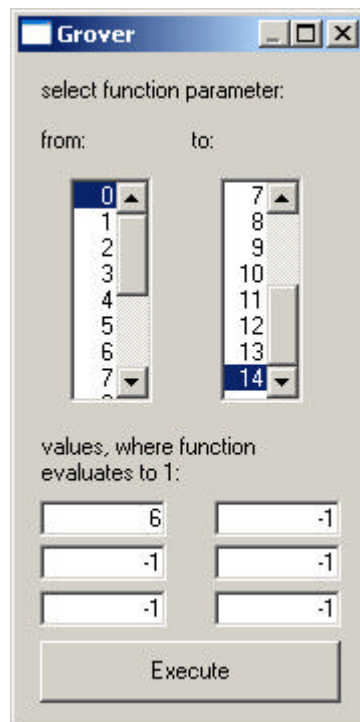
unsigned int n=(unsigned int)(PI/4
    *sqrt(pow(2,((unsigned int*)param)[1]
        -((unsigned int*)param)[0]+1)/((unsigned int*)param)[2]))+1;

for (i=0;i<n;i++)
    procexecute("grover\\groverstep",param);

char buf[128];
sprintf(buf,"Grover ( %i )",n);
procstep(buf);
}

```

Das Fenster des Gatters sieht wie folgt aus:



Im oberen Bereich kann in der linken Spalte das Startbit, in der rechten Spalte das Endbit des Bereichs des Funktionsparameters angegeben werden. Im unteren Bereich können die Werte  $x$  aus dem Definitionsbereich von  $f$  angegeben werden, für die  $f(x)=1$ . Diese  $x$  entsprechen den zu suchenden Werten. Eine -1 bedeutet, dass dieses Eingabefeld nicht beachtet werden soll. Durch Anklicken von Execute wird die Transformation ausgeführt.

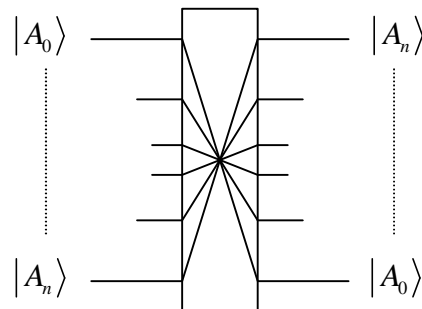
#### 4.4 Shor

Im Folgenden werden die Gatter vorgestellt, die Shors folgenden Algorithmus zur Faktorisierung eine Zahl  $N$  in Polynomialzeit auf einem Quantencomputer implementieren:

1. Ist  $N$  gerade? Wenn ja liefere 2 als Faktor und beende
2. Überprüfe, ob es  $a$  und  $b$  gibt mit  $a^b=N$ . Ist dies der Fall, liefere  $a$  als Faktor zurück
3. Wähle mehrmals zufällig ein  $x$  aus  $[1, N-1] \subset \mathbb{N}$ . Ist  $f := \gcd(x, N) > 1$ , liefere  $f$  als Faktor zurück
4. Wähle ein  $x$  mit  $\gcd(x, N) = 1$ . Bestimme  $r := \text{ord}_N(x)$  mit dem Quantenalgorithmus.
5. Wenn  $r$  gerade, dann überprüfe ob  $x^{\frac{r}{2}} \bmod N \neq N-1$ . Ist dies der Fall, so gib  $\gcd\left(x^{\frac{r}{2}}+1, N\right)$  oder  $\gcd\left(x^{\frac{r}{2}}-1, N\right)$  als Faktor zurück, wenn einer von ihnen nicht trivial ist. Sind beide trivial, kehre zurück zu Schritt 3. Ist  $r$  ungerade, so kehre auch zurück zu Schritt 3.

#### 4.4.1 Bitexchange-Gatter

Dieses Gatter vertauscht die Qubits eines Bereichs des Quantenregisters.



Die Implementation des Gatters:

```
void __stdcall execute(unsigned int param)
{
    unsigned int from=((unsigned int *)param)[0];
    unsigned int to=((unsigned int *)param)[1];
    if (from<=to)
    {
        unsigned int i,j,src,dest;

        Register t;
        t.SetBitn(r->n);

        unsigned int mask=0xffffffff;
        mask=mask<<(32-to-1);
        mask=mask>>(32-to-1);
        mask=mask>>from;
        mask=mask<<from;

        for (i=0;i<(1UL<<r->n);i++)
        {
```

```

        src=i&mask;
        src=src>>from;

        dest=0;
        for (j=0;j<to-from+1;j++)
        {
            dest=dest<<1;
            dest|=src&1;
            src=src>>1;
        }

        dest=dest<<from;
        dest|=i&(mask^0xffffffff);

        t.p[dest]=r->p[i];
    }

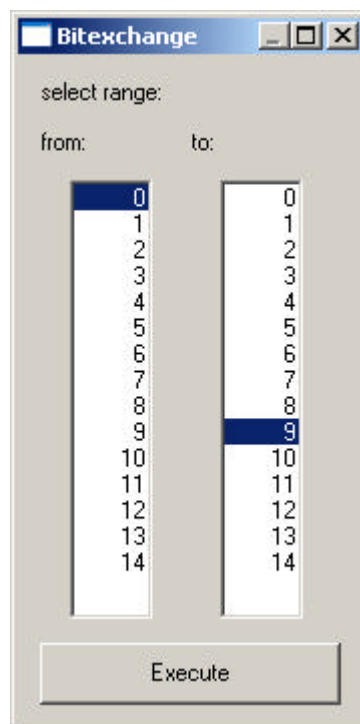
    r->copyfrom(&t);

    char buf[128];
    sprintf(buf,"Bitexchange ( %i - %i )",from,to);
    procstep(buf);
}
}

```

Dieses Gatter vertauscht die Qubits des Registers im angegebenen Bereich.

Das Fenster des Gatters sieht wie folgt aus:



In der linken Spalte kann das Startbit, in der rechten Spalte das Endbit des Bereichs angegeben werden, dessen Bits vertauscht werden. In der rechten

Spalte wird das Zielbit ausgewählt. Durch Anklicken von Execute wird die Transformation ausgeführt.

#### 4.4.2 Power-Gatter

Dieses Gatter führt die Berechnung von  $a^x \bmod N$  durch, die im Shor-Algorithmus benötigt wird, was folgender Transformation entspricht:

$$\frac{1}{\sqrt{2^t}} \sum_{a=0}^{2^t-1} |a\rangle |0\rangle \rightarrow \frac{1}{\sqrt{2^t}} \sum_{a=0}^{2^t-1} |a\rangle |x^a \bmod N\rangle \quad (16)$$

wobei  $t$  die Anzahl der Qubits sind die potenziert werden sollen und  $t = to - from + 1$ .

Implementation des Gatters:

```
void __stdcall execute(unsigned int param)
{
    unsigned int from,to,x,n;
    from=((unsigned int*)param)[0];
    to=((unsigned int*)param)[1];
    x=((unsigned int*)param)[2];
    n=((unsigned int*)param)[3];

    if (x>0)
    {
        unsigned int i,j,a,erg;
        erg=1;
        for (i=0;i<(1UL<<(to-from+1));i++)
        {

            erg*=x;
            erg%=n;

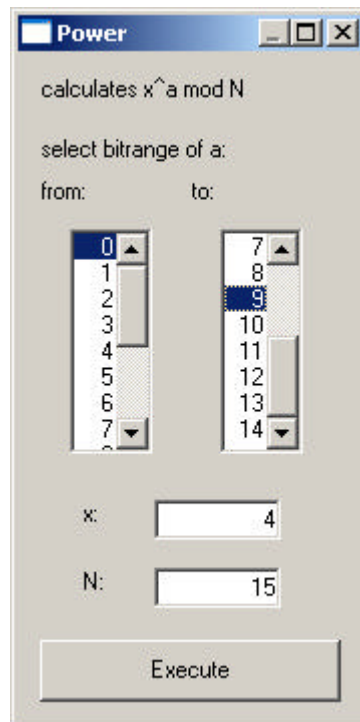
            if ((erg<<(to+1))<(1UL<<r->n))
                for (j=0;j<(1UL<<from);j++)
                {
                    r->p[j]|(i<<from)|(erg<<(to+1))]
                    =r->p[j]|(i<<from)];
                    r->p[j]|(i<<from)]=0;
                }
        }

        char buf[128];
        sprintf(buf,"Power ( %i - %i, x = %i, n = %i )",from,to,x,n);
        procstep(buf);
    }
}
```

Zu Beachten bei der Implementation ist, dass nicht die sonst gängige Methode zur Beschleunigung der Berechnung der Potenz modulo  $n$  verwendet wird. Da dieses Gatter später für die Simulation von Shors Algorithmus verwendet wird und sich die Qubits dabei in Superposition befinden, muss sowieso  $x^i$  für alle  $i \in [0, 2^{to-from+1} - 1]$  berechnet werden. Somit ist es

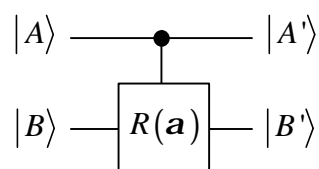
schneller, in der Schleife über die  $i$  die vorherige Potenz immer mit  $x$  zu multiplizieren.

Das Fenster des Gatters sieht wie folgt aus:



In der linken Spalte kann das Startbit, in der rechten Spalte das Endbit des Bereichs angegeben werden, das  $a$  enthält. Im unteren Bereich können  $x$  und  $N$  angegeben werden. Das jeweilige Ergebnis von  $x^a \bmod N$  wird im Register beginnend hinter Qubit  $t_0$  abgelegt. Durch Anklicken von Execute wird die Transformation ausgeführt.

#### 4.4.3 Rotate two bits controlled-Gatter



Dieses Gatter entspricht der unitären Matrix

$$R(\mathbf{a}) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{ia} \end{pmatrix} \quad (17)$$

Implementation des Gatters:

```
void __stdcall execute(unsigned int param)
{
    unsigned int bit[2];
    bit[0]=((unsigned int*)param)[0];
    bit[1]=((unsigned int*)param)[1];
    double angle;
    angle=*((double*)&(((unsigned int*)param)[2]));

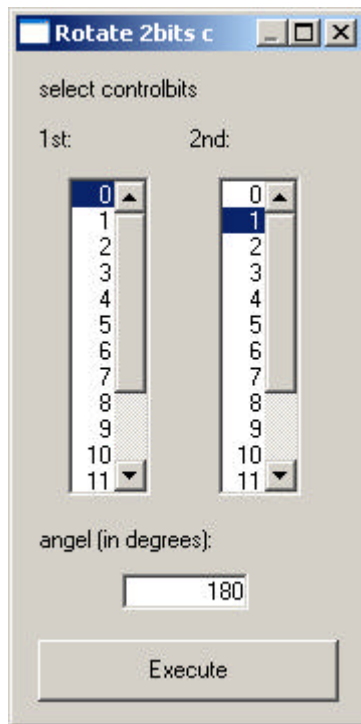
    Complex faktor;
    faktor.expi(angle);

    int i;
    for (i=0;i<(1<<r->n);i++)
    {
        if (((i&(1<<bit[0]))!=0) && ((i&(1<<bit[1]))!=0))
        {
            r->p[i]*=faktor;
        }
    }

    char buf[128];
    sprintf(buf,"Rotate 2 bits controlled ( bits: %i, %i, angle:
        %.5g)",bit[0],bit[1],angle);
    procstep(buf);
}
```

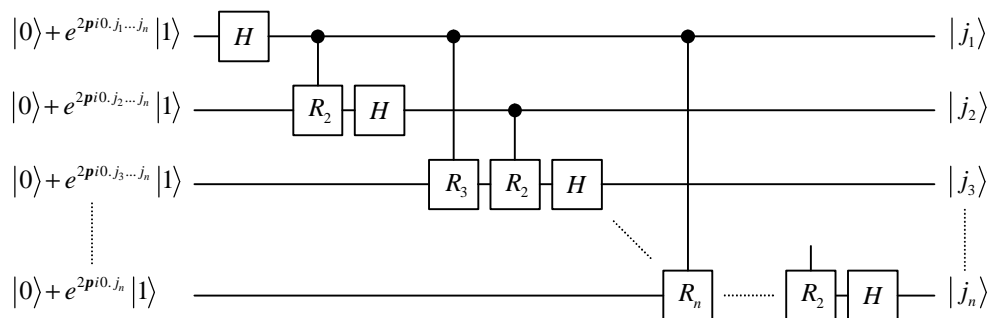
Das Register wird einfach durchlaufen. Sind die beiden Controlbits gesetzt, so wird der Koeffizient mit  $e^{ia}$  multipliziert.

Das Fenster des Gatters sieht wie folgt aus:



In der linken Spalte kann das Startbit, in der rechten Spalte das Endbit des Bereichs angegeben werden, auf den die Transformation angewandt wird. Im unteren Textfeld wird der Winkel in Grad angegeben um den gedreht werden soll. Durch Anklicken von Execute wird die Transformation ausgeführt.

#### 4.4.4 Inverse Fast Fourier Transformation-Gatter



wobei

$$R_k = R\left(-\frac{2p}{2^k}\right)$$

Die Implementation des Gatters:

```

void __stdcall execute(unsigned int param)
{
    unsigned int from=((unsigned int *)param)[0];
    unsigned int to=((unsigned int *)param)[1];
    if (from<=to)
    {
        unsigned int i,j;

        procexecute("basic\\walsh",from);
        for (i=from+1;i<=to;i++)
        {
            for (j=from;j<i;j++)
            {
                double angle=-PI/pow(2,i-j);
                unsigned int param[4];
                param[0]=i;
                param[1]=j;
                *((double*)&param[2])=angle;

                procexecute("shor\\modules\\rotate2bitscontrolled",
                    (unsigned int)param);

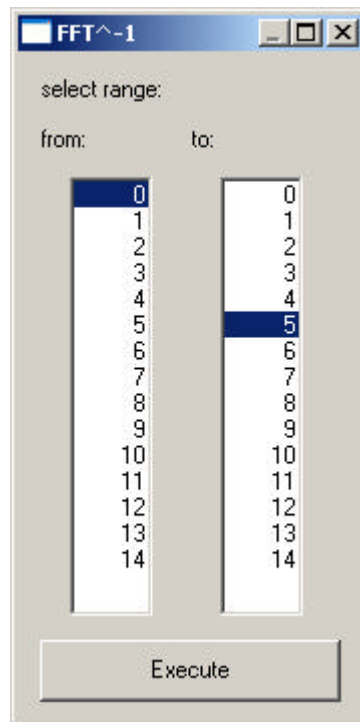
            }
            procexecute("basic\\walsh",i);
        }

        char buf[128];
        sprintf(buf,"FFT^-1 ( %i - %i )",from,to);
        procstep(buf);
    }
}

```

Die Implementation benutzt die bereits vorgestellten Gatter wie oben in der Abbildung beschrieben.

Das Fenster des Gatters sieht wie folgt aus:



In der linken Spalte kann das Startbit, in der rechten Spalte das Endbit des Bereichs angegeben werden, auf den die inverse Fast Fourier Transformation angewandt wird. Durch Anklicken von Execute wird die Transformation ausgeführt.

#### 4.4.5 Continuous Fractions-Gatter

Dieses Gatter ist eines der klassischen Teile von Shors-Primfaktorierungs-Algorithmus. Es stellt somit kein Quantengatter dar. Es rechnet nur klassisch mit Messergebnissen aus dem Zustand des simulierten Quantencomputers.

Jede rationale Zahl  $z$  kann als Kettenbruch dargestellt werden:

$$z = \frac{x}{y} = a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \frac{1}{\dots + \frac{1}{a_M}}}} =: [a_0, \dots, a_M] \quad (18)$$

wobei  $x, y, a_i \in \mathbb{Z}$ .

Die  $a_i$  berechnen sich wie folgt:

$$\begin{aligned}
b_{-1} &= x, r_{-1} = y \\
a_i &= \lfloor b_{i-1} / r_{i-1} \rfloor, r_i = b_{i-1} \bmod r_{i-1}
\end{aligned}
\tag{19}$$

Die zweite Zeile ist so oft zu iterieren bis  $r_i=0$ . Dies ist jedoch nichts Anderes als der Euklidische Algorithmus. Nebenbei erhält man durch diese Erkenntnis die Aussage, dass es tatsächlich für jede rationale Zahl eine Kettenbruchdarstellung gibt, da der Euklidische Algorithmus immer terminiert. Ferner ist die Darstellung eindeutig.

Da uns für die Verwendung des Kettenbruchalgorithmuses nur der Nenner des gekürzten Bruches  $\frac{x'}{y'}$  interessiert, berechnen wir ihn, wenn der Algorithmus durchgelaufen ist, mit

$$y' = \frac{y}{r_{i-1}} = \frac{y}{\text{GCD}(x, y)} = \text{ord}_N(x) \tag{20}$$

Die Implementation des Kettenbruchalgorithmuses sieht somit wie folgt aus:

```

unsigned int gcd(unsigned int x, unsigned int y)
{
    unsigned int rest=1;
    unsigned int value;
    while (rest!=0)
    {
        value=x/y;
        rest=x%y;
        x=y;
        y=rest;
    }
    return x;
}

unsigned int continuedfractions(unsigned int x, unsigned int y)
{
    return y/gcd(x,y);
}

```

Anschließend übernimmt dieses Gatter noch die Durchführung von Punkt 5 des Shor Algorithmus.

5. Wenn  $r$  gerade, dann überprüfe ob  $x^{\frac{r}{2}} \bmod N \neq N-1$ . Ist dies der Fall, so gib  $\text{gcd}\left(x^{\frac{r}{2}} + 1, N\right)$  oder  $\text{gcd}\left(x^{\frac{r}{2}} - 1, N\right)$  als Faktor zurück, wenn einer von ihnen nicht trivial ist. Sind beide trivial, kehre zurück zu Schritt 3. Ist  $r$  ungerade, so kehre auch zurück zu Schritt 3.

Die Implementation entspricht den Fallunterscheidungen. Zu beachten ist, dass die Simulation jedoch nicht von selber zu Schritt 3 zurückkehrt und

automatisch eine neue Simulation startet wenn kein Faktor gefunden wurde. Statt zu Schritt 3 zurückzukehren, gibt der Simulator "No factor found. The algorithm failed." bzw. "Trivial factor found. The algorithm failed." aus.

Die Implementation des Gatters:

```

unsigned int powermod(unsigned int a, unsigned int e, unsigned int n)
{
    if (a==0)
    {
        if (e!=0)
            return 0;
        else
            return -1;
    }

    unsigned int power;
    power=1;
    while (e!=0)
    {
        if (e&1)
        {
            power*=a;
            power%=n;
        }
        a*=a;
        a%=n;
        e>>=1;
    }

    return power;
}

void __stdcall execute(unsigned int param)
{
    unsigned int from=((unsigned int *)param)[0];
    unsigned int to=((unsigned int *)param)[1];
    unsigned int x=((unsigned int *)param)[2];
    unsigned int N=((unsigned int *)param)[3];

    if (from<=to)
    {
        unsigned int a,ord,pot,a0,a1;
        char measured[128],result[128],ordnung[128],buf[128*3+16];

        a=measure();
        a=a<<(32-to-1);
        a=a>>(32-to-1);
        a=a>>from;

        sprintf(measured,"Measured: a = %i",a);

        ord=continuedfractions(a,1UL<<(to-from+1));

        sprintf(ordnung,"Order: ord %i (%i) = %i",N,x,ord);
        sprintf(result,"No factor found. The algorithm failed.");

        if (ord%2==0)
        {
            pot=powermod(x,ord/2,N);

            if ((pot!=N-1)// && (pot!=1))
            {
                a0=gcd(pot-1,N);
                a1=gcd(pot+1,N);
            }
        }
    }
}

```

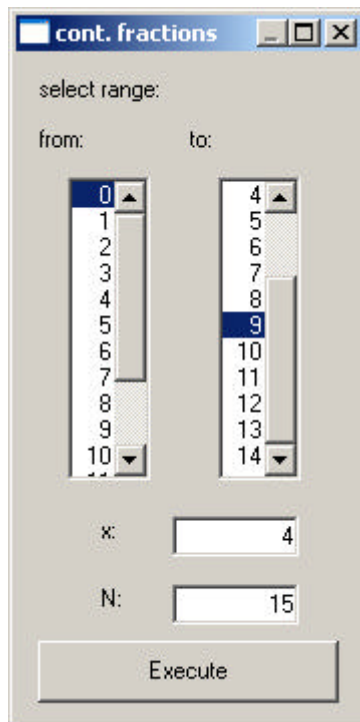
```

        if ((a0!=1) && (a0!=N) && (a1!=1) && (a1!=N))
        {
            sprintf(result,"Factors found: a = %i,
            b = %i and a = %i, b =
            %i",a0,N/a0,a1,N/a1);
        }
        else
        if ((a0!=1) && (a0!=N))
        {
            sprintf(result,"Factors found: a = %i,
            b = %i",a0,N/a0);
        }
        else
        if ((a1!=1) && (a1!=N))
        {
            sprintf(result,"Factors found: a = %i,
            b = %i",a1,N/a1);
        }
        else
        {
            sprintf(result,"Trivial factor found.
            The algorithm failed.");
        }
    }
}

sprintf(buf,"%s%c%c%c%s%c%c%c%s",measured,13,10,13,10,
ordnung,13,10,13,10,result);
MessageBox(NULL,buf,"Result",MB_OK|MB_TASKMODAL);
}
}

```

Das Fenster des Gatters sieht wie folgt aus:



In der linken Spalte kann das Startbit, in der rechten Spalte das Endbit des Bereichs des Messergebnisses angegeben werden, der als Parameter des Continued Fractions Algorithmuses verwendet wird. Im unteren Bereich werden  $x$  und  $N$  festgelegt. Durch Anklicken von Execute wird der Algorithmus gestartet.

#### 4.4.6 Shor-Gatter

Dieses Gatter implementiert den Shor Algorithmus ab Schritt 4. Die ersten drei Schritte werden nicht durchgeführt, da bei dem beschränkten verwendbaren Wertebereich Schritt 1 bis 3 im Großteil aller Fälle einen Faktor finden und der Algorithmus terminiert bevor der Quantencomputer-Teil erreicht wird.

$x$  und  $N$  sind vom Benutzer angegeben. Dieses Gatter führt im Wesentlichen die Ordnungsbestimmung  $ord_N(x)$  durch:

1. Zu Beginn des Algorithmus sei der Zustand des Quantenregisters  $|0\rangle|0\rangle$ .
2. Die  $t=to-from+1$  Qubits werden in Superposition gebracht mittels Walshrange und man erhält  $\frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle|0\rangle$ .
3. Das Power-Gatter wird angewandt mit den  $t$  Qubits als Parameter  $a$ .  
Man erhält  $\frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle|x^j \bmod N\rangle \approx \frac{1}{\sqrt{r2^t}} \sum_{s=0}^{r-1} \sum_{j=0}^{2^t-1} e^{\frac{2\pi i s j}{r}} |j\rangle|u_s\rangle$ .
4. Nach vertauschen der Qubit-Reihenfolge durch das Bitexchange-Gatter wird die Inverse Fourier Transformation angewandt. Die führt näherungsweise zu  $\frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} \left| \frac{s}{r} \right\rangle |u_s\rangle$ .
5. Dieser Zustand wird gemessen und der Kettenbruchalgorithmus auf die  $t$  Qubits und  $2^N$  angewandt. Also auf  $\frac{s/r}{2^N}$ . Das Ergebnis des Kettenbruchalgorithmuses ist  $ord_N(x)$ .

Punkt 5 und die anschließende Bestimmung eines Faktors wird dabei wie zuvor beschrieben vom Kettenbruch-Gatter durchgeführt.

Die Implementation des Gatters:

```
void __stdcall execute(unsigned int param)
{
    unsigned int from,to,x,n;
```

```

from=((unsigned int*)param)[0];
to=((unsigned int*)param)[1];
x=((unsigned int*)param)[2];
n=((unsigned int*)param)[3];

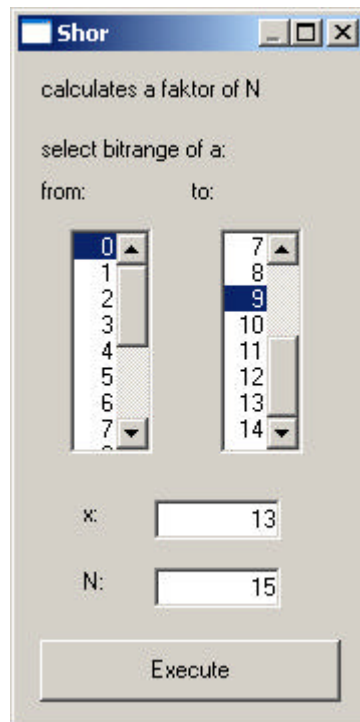
if (from<to)
{
    procexecute("basic\\walshrange",param);
    procexecute("shor\\modules\\power",param);
    procexecute("shor\\modules\\bitexchange",param);
    procexecute("shor\\modules\\fft-1",param);

    char buf[128];
    sprintf(buf,"Shor ( %i - %i, x = %i, n = %i )",from,to,x,n);
    procstep(buf);

    procexecute("shor\\modules\\contfractions",param);
}
}

```

Das Fenster des Gatters sieht wie folgt aus:



In der linken Spalte kann das Startbit, in der rechten Spalte das Endbit des Bereichs von  $a$  angegeben werden. Im unteren Bereich werden  $x$  und  $N$  festgelegt. Durch Anklicken von Execute wird der Algorithmus gestartet.

#### 4.4.7 Shor-Begin-Gatter

Dieses Gatter implementiert die ersten drei Schritte von Shors Algorithmus zur Faktorisierung. Diese Schritte sind klassisch.

1. Ist  $N$  gerade? Wenn ja liefere 2 als Faktor und beende
  2. Überprüfe, ob es  $a$  und  $b$  gibt mit  $a^b=N$ . Ist dies der Fall, liefere  $a$  als Faktor zurück
  3. Wähle mehrmals zufällig ein  $x$  aus  $[1, N-1] \subset \mathbb{N}$ . Ist  $f := \gcd(x, N) > 1$ , liefere  $f$  als Faktor zurück
- Sonst versagt der klassische Teil des Algorithmus.

Punkt 1 kann leicht implementiert werden indem in der Dual-Darstellung von  $N$  die hinterste Ziffer betrachtet wird:

```
if (!(N&1)) return 2;
```

Für Punkt 2 gilt zunächst folgende Überlegung: Da  $a \geq 2$  muss  $b \leq \lceil \log_2 N \rceil + 1$ . Da  $N$  in der Größe auf  $2^{32}$  beschränkt ist, gilt  $b \leq 33$ . Diese Grenze kann durchaus noch niedriger abgeschätzt werden, da der Simulator Speicher nicht nur für das Quantenregister benötigt und der Computer auf dem er läuft auch nicht unbedingt mit 4GByte RAM ausgerüstet ist. Ferner benötigt der Shor-Algorithmus zusätzlich noch temporäre Qubits. In der hier vorgestellten Implementation wird vom worst-case ausgegangen, also werden alle  $b \in [2, 33]$  getestet.

Um die Gleichung  $a^b = N$  für gegebenes  $b$  und  $N$  zu lösen, wird das Newton-Verfahren angewandt. Dafür gilt:

$$\begin{aligned}
 f(a) &= a^b - N \\
 g(a) &= a - \frac{f(a)}{f'(a)} \\
 &= a - \frac{a^b - N}{ba^{b-1}} \\
 &= a - \frac{a^b}{ba^{b-1}} + \frac{N}{ba^{b-1}} \\
 &= a - \frac{a}{b} + \frac{N}{ba^{b-1}} \\
 &= \frac{ba - a}{b} + \frac{N}{ba^{b-1}} \\
 &= \frac{1}{b} \left( (b-1)a + \frac{N}{a^{b-1}} \right)
 \end{aligned} \tag{21}$$

g wird fixpunktiteriert. Da gilt  $N^{\frac{1}{b-1}} \geq N^{\frac{1}{b}}$ , kann als Startpunkt der Iteration immer das Ende der vorherigen gewählt werden, wenn b von 2 aus aufwärts gezählt wird. Gestartet wird in der folgenden Implementation bei b=2 mit  $a^{(0)} = \frac{N}{2}$ .

Die Implementation des Gatters:

```

unsigned int nthroot(unsigned int N, unsigned int b, unsigned int
*start)
{
    unsigned int a;
    a=*start;
    unsigned int p=N;
    while (p*a>N)
    {
        p=power(a,b-1);
        a=((b-1)*a+N/p)/b;
    }
    *start=a;

    if (p*a==N)
        return a;
    else
        return 0xffffffff;
}

unsigned int Shor_Step2(unsigned int N)
{
    unsigned int b,f0,f1;
    f0=N/2;
    for (b=2;b<33;b++)
    {
        f0++;
        if (f0==2)
            return 0xffffffff;

        f1=nthroot(N,b,&f0);
        if (f1!=0xffffffff)
            return f1;
    }
    return 0xffffffff;
}

```

Punkt 3 ist eine Schleife über den Euklidischen Algorithmus:

```

unsigned int gcd(unsigned int x,unsigned int y)
{
    unsigned int rest=1;
    unsigned int value;
    while (rest!=0)
    {
        value=x/y;
        rest=x%y;
        x=y;
        y=rest;
    }
    return x;
}

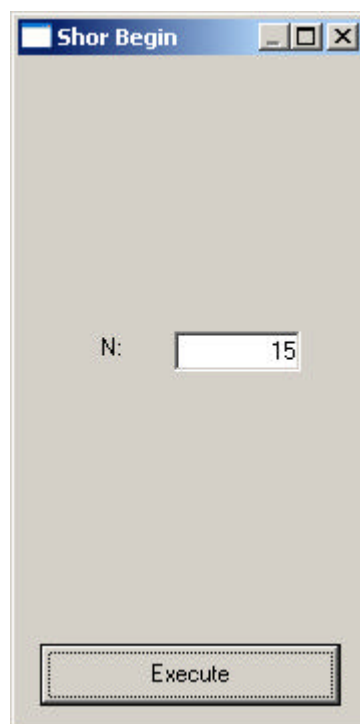
```

```

unsigned int Shor_Step3(unsigned int N)
{
    unsigned int i,x,f;
    for (i=0;i<20;i++)
    {
        x=(( (double)rand())/RAND_MAX)*(N-2) )+1;
        f=gcd(x,N);
        if (f>1)
            return f;
    }
    return 0xffffffff;
}

```

Das Fenster des Gatters sieht wie folgt aus:



In dem Textfeld kann der zu faktorisierte Wert  $N$  eingegeben werden. Durch Execute wird der Algorithmus gestartet.

## 5 Literaturangaben

- [1] Nielsen, Michael A. and Isaac L. Chuang, Quantum Computation and Quantum Information, Cambridge University Press, 2000
- [2] Heinrich, Stefan, Notes on mathematical foundations of quantum computing, Kaiserslautern, 2002
- [3] Shor, Peter W., Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, SIAM Journal on Computing, 1997
- [4] Wallace, J., Quantum Computer Simulators, <http://www.dcs.ex.ac.uk/~jwallace/simtable.htm>, 16.12.2002