Quantum Programming in QCL

Bernhard Ömer

20th January 2000

Institute of Information Systems

Technical University of Vienna

E-mail:	`oemer@tph.tuwien.ac.at`
Homepage:	`http://tph.tuwien.ac.at/~oemer`

Contents

Chapter 1

Quantum Physics in a Nutshell

While it is possible, to introduce quantum computation in a strictly algebraic manner without ever mentioning "real world" things like electrons, particle states or charge densities[1]*, some basic knowledge about general quantum physics can vastly improve the understanding of why certain quantum algorithms or programming techniques actually work and are a good precaution against common misconceptions.*

1.1 A Brief History of Quantum Physics

1.1.1 Particles and Waves

An important problem in physics before the adoption of the quantum theory, has been the distinction between particle and wave phenomena.

At first glance, both concepts have very little in common: Nobody would treat a flying bullet as a wave packet or the propagation of sound as a particle stream, but when particles and wave-lengths get smaller, things aren't so clear:

In the 17^{th} century, Newton used both theories to cover the different aspects of light [23], explaining its periodicy and interference as wave, and it's linear propagation as particle phenomenon. Later, the wave-theory of light has been generally accepted, as scientists like Young and Fresnel could explain most particulate behavior within the realm of the wave-formalism. Except, that is, for one fundamental requirement: The obvious lack of a physical medium which lead to the somewhat far-fetched and unsatisfying "Ether" hypothesis.

[1] which should still be at a safe distance from most peoples' notion of "real"

On the particle front, Dalton's Law of Multiple Proportions suggested, that chemical substances consist of atoms of different masses. In the 19$^{\text{th}}$ century, Boltzmann developed his gas-theory based on atomistic concepts and experiments with cathode rays showed that the electric charges always come in multiples of the elementary charge e which is about 1.6×10^{-19} Coulomb.

1.1.2 Plank's Constant

In the year 1900, Max Plank explained the energy spectrum of black body radiation with the ad-hoc assumption, that the possible energy states are restricted to $E = nh\nu$, where n is an integer, ν the frequency and h the Plank constant, the fundamental constant of quantum physics, with a value of

$$h = 2\pi\hbar = 6.626075 \cdot 10^{-34}\text{Js}$$

In 1888, Hertz demonstrated, that a negatively charged plate would discharge, if exposed to ultraviolet light. Lenard later discovered the kinetic energy of the electrons is independent of the light's intensity but correlated to its frequency, such that

$$\mathrm{e}U = C\nu - P$$

with some material dependent constant P. In 1905 Einstein reformulated this relation to

$$E = \mathrm{e}(U + P) = h\nu = \hbar\omega$$

interpreting E as the energy of a light particle, later called a *photon*.

1.1.3 Bohr's Atom Model

By analyzing the visible spectrum of Hydrogen, it was found that the light intensity shows very distinct peaks at certain wavelengths. In 1885, Balmer showed that the wavelength l is very accurately given by

$$l = 364.56 \, \frac{a^2}{a^2 - 4} \, \text{nm} \tag{1.1}$$

This can be generalized to the Rydberg equation, which also accounts for the non-visible parts of the spectrum

$$\frac{1}{l} = R_H \left(\frac{1}{k^2} - \frac{1}{a^2} \right) \tag{1.2}$$

This suggests, that the electron in the Hydrogen atom is confined to certain energy levels, which is in contradiction with classical mechanics.

The Bohr-Sommerfeld model accounted for this by introducing a *quantum condition*: While the electrons are still assumed to circulate the nucleus on their classical orbits, their angular momentum has to be a multiple of $\hbar$. This restriction could be justified by attributing wave properties to the electron and demanding that their corresponding wave functions form a standing wave; however this kind of hybrid theory remained unsatisfactory.

A complete solution for the problem came in 1923 from Heisenberg who used a matrices-based formalism. In 1925, Schrödinger published an alternative solution using complex wave functions. It took two years until Dirac showed that both formalisms were in fact equivalent.

1.1.4 Wave-Particle Dualism

In 1924, de Broglie assumed that — in analogy to photons — every particle of energy E and momentum $\vec{p}$ can in fact be treated as a wave, whose frequency ω and wave-vector $\vec{k}$ are given by

$$\omega = \frac{E}{\hbar} \quad \text{and} \quad \vec{k} = \frac{\vec{p}}{\hbar} \tag{1.3}$$

This relation was verified in 1927 in diffraction experiments with electrons by Davison and Germer. The inverse effect — particle behavior of photons — has been demonstrated 1933 in electron-photon dispersion experiments by Compton.

1.2 Wave Mechanics

1.2.1 Classical States

In classical physics, the momentary state of a particle is given by it's location $\vec{r}$ and it's velocity $\vec{v}$.

When we talk about the temporal behavior of dynamic systems, however, this notion of "state" is somewhat cumbersome to deal with, since by definition, the momentary state of the system changes constantly. This is especially true when it comes to periodic movement, so it is often more adequate to talk about the current orbit of a satellite (which remains constant until it is actively altered by outside intervention) than to give the actual coordinates (which permanently change).

So in a more abstract definition, the states of an isolated classical system are the positions $\vec{r}_1, \vec{r}_2, \ldots$ of all included particles as a function of time t.[2]

1.2.1.1 State Changes

The above definition implies that the state of a system can only change when an interaction with another system occurs.

Typically, the duration of the interaction (e.g. the collision of 2 billiard-balls) is very small compared to the duration of the isolated states, so for practical purposes the interaction can often be assumed as instantaneous.

1.2.1.2 Conservation Laws

Isolated systems preserve their total energy E and momentum[3] $\vec{p}$, which are given as[4]

$$E = V(\vec{r}_1, \vec{r}_2, \ldots) + \sum_i m_i v_i^2 \quad \text{and} \quad \vec{p} = \sum_i m_i \vec{v}_i \tag{1.4}$$

Thus, states can be characterized by these quantities and often the total energy of a state is much more interesting than the state itself.

1.2.1.3 Movement Laws

Legal physical states must obey a movement law which characterizes the dynamics of a system. For classic one-particle systems, the dynamic equation is known as Newton's Second Law

$$m\frac{\partial^2 \vec{r}}{\partial t^2} = \vec{F}(\vec{r}, t) \tag{1.5}$$

For conservative fields such as non relativistic gravitational and static electric fields, the force $\vec{F}$ is the negative gradient of a scalar potential $V(\vec{r})$, thus the equation above can be written as

$$m\frac{\partial^2 \vec{r}}{\partial t^2} = -\mathrm{grad}\, V(\vec{r}) \tag{1.6}$$

---

[2] Note, that since $\vec{v} = \dot{\vec{r}} = \partial\vec{r}/\partial t$, this also includes the velocities.

[3] There are several other conservation laws for angular momentum, electric charge, baryon count, etc. which are not mentioned here

[4] The form of the potential energy V actually defines the physical problem and can also depend on particle velocities, time, spins, etc.

Any momentary state of the system can be used as an initial value for the above equation to determine its temporal behavior.

1.2.2 The Wave Function

In quantum physics, the state of a one-particle system is characterized by a complex distribution function $\psi(\vec{r}, t)$ with the normalization

$$\int |\psi(\vec{r}, t)|^2 \, d^3\vec{r} = 1 \tag{1.7}$$

Two states differing by a constant phase factor $\mathrm{e}^{\mathrm{i}\phi}$ are considered equivalent.

1.2.2.1 Particle Location

The classical notion of particle location is replaced by a spatial probability distribution $\rho = \psi^*\psi$, which can be characterized by its expectation value $\langle\vec{r}\rangle$ and its uncertainty Δr, which are defined as

$$\langle\vec{r}\rangle = \int \psi^*(\vec{r}, t)\vec{r}\psi(\vec{r}, t) \, d^3\vec{r} \quad \text{and} \quad \Delta r = \sqrt{\langle r^2\rangle - \langle\vec{r}\rangle^2} \tag{1.8}$$

1.2.2.2 Time Dependency

When a classical system involves moving particles, the location of the particles is time dependent. This is not necessarily the case with quantum systems and the describing probability distribution ρ: If the quantum state ψ is of the form $\psi(\vec{r}, t) = \psi(\vec{r})\phi(t)$ with $|\phi(t)| = 1$, then $\rho = \psi^*(\vec{r})\psi(\vec{r})$ is time independent.

Figure 1.1 shows a particle that is trapped between two reflecting "mirrors".[5] A classical particle will move periodically from on end to another at a constant speed, it's location can be described by a periodic triangle-function of the time. An undisturbed quantum particle in a similar trap, however, doesn't have a defined location; the probability to "meet" (i.e. measure) the particle at a certain location remains constant over time[6] but changes throughout space, or in more physical terms, the particle forms a standing *wave* just as a vibrating piano-string between 2 fixed ends.

[5] Mathematically, such ideal "mirrors" are described by infinitely deep potential-wells.

[6] This is only the case with eigenstates; in mixed states, the local probability can oscillate due to the different periods of the involved phase functions $\phi_i(t)$

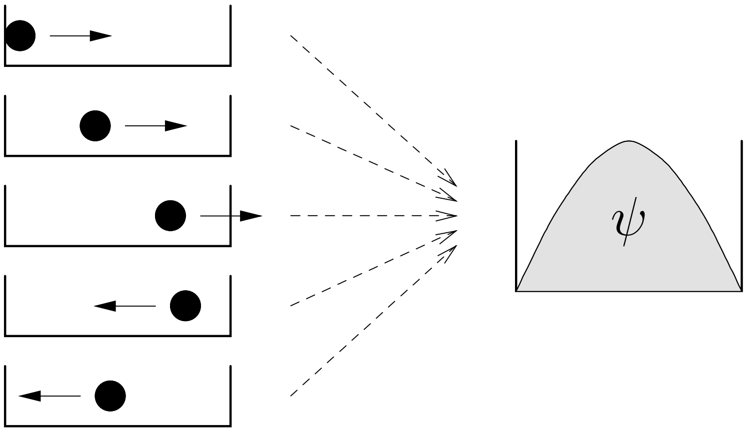

Figure 1.1: A ball trapped between two mirrors as classical and as quantum particle

A constant probability distribution is typical for bound states of defined energy, i.e. for particles trapped in a constant potential well, e.g. an electron in the electric field of a proton.

1.2.2.3 Expectation Values

It has been shown above how the classical concept of a well defined particle location has been replaced by the quantum concept of a statistical expectation value. This correspondence, however, is not just restricted to space. In fact, all classical physical quantities of a system can be described as the expectation value of an appropriate *operator* (see table 1.1 for some examples).

Observable	classical value	Operator
Location	$\vec{r}$	$\vec{R} = \vec{r},\ R_i = r_i$
Momentum	$\vec{p} = m\frac{\partial \vec{r}}{\partial t}$	$\vec{P} = -i\hbar\nabla,\ P_i = -i\hbar\frac{\partial}{\partial r_i}$
Angular Momentum	$\vec{L} = \vec{p} \times \vec{r}$	$\vec{\mathcal{L}} = \vec{R} \times \vec{P},\ \mathcal{L}_i = -i\hbar\epsilon_{ijk} r_j \frac{\partial}{\partial r_k}$
Energy	$E = \frac{p^2}{2m} + V(\vec{r})$	$H = -\frac{\hbar^2}{2m}\Delta + V$

Table 1.1: Some observables and their corresponding operators

In analogy to equation 1.2.2.1, the expectation value $\langle O \rangle$ and the uncertainty for an observable $\mathcal{O}$ are defined as

$$\langle O \rangle = \int \psi^*(\vec{r},t)\, O\, \psi(\vec{r},t)\, d^3\vec{r} \quad \text{and} \quad \Delta O = \sqrt{\langle O^2 \rangle - \langle O \rangle^2} \tag{1.9}$$

1.2.3 The Schrödinger Equation

The quantum analogy to Newton's Third Law (see equation 1.6) is the *Schrödinger Equation*

$$H\,\psi = i\hbar \frac{\partial}{\partial t}\,\psi \tag{1.10}$$

which determines the dynamics of a particle system. The *Hamilton operator* H describes the total energy of the system at a given time and can be very complicated.

1.2.3.1 The Time-Independent Schrödinger Equation

If we take the simple case of a particle in a static potential field V, equation 1.10 can be written as

$$\left(-\frac{\hbar^2}{2m}\nabla^2 + V(\vec{r})\right)\psi(\vec{r},t) = i\hbar\frac{\partial}{\partial t}\,\psi(\vec{r},t) \tag{1.11}$$

If we split off the time dependent part, using the ansatz $\psi(\vec{r},t) = \psi(\vec{r})\phi(t)$ from 1.2.2.2 and the separation parameter E, we get

$$E\,\phi(t) = i\hbar\frac{\partial}{\partial t}\,\phi(t) \quad \text{and} \quad H\,\psi(\vec{r}) = E\,\psi(\vec{r}) \tag{1.12}$$

The time part is solved by $\phi = \mathrm{e}^{-\mathrm{i}\omega t}$ with $\omega = E/\hbar$. E is the energy of the state, since

$$\langle H \rangle = \int \psi^*(\vec{r})\,(H\,\psi(\vec{r}))\; d^3\vec{r} = E\int \psi^*(\vec{r})\,\psi(\vec{r})\, d^3\vec{r} = E \tag{1.13}$$

The remaining eigenvalue problem $E\,\psi = H\,\psi$ is also called the *time-independent Schrödinger Equation.*[7]

1.2.3.2 Energy Spectra

Depending on the imposed boundary conditions, the Schrödinger Equation is often only solvable for particular values of E, i.e. it has a discrete *energy*

[7] Note that this requires the Hamilton operator to be time-independent, which is not necessarily the case

spectrum and the possible eigenvalues E_n (also called energy terms) can be enumerated. The solution for the lowest eigenvalue E_0 is called the ground-state ψ_0 of the system.

Since for most physical applications, only the value of the energy terms is of importance, it is hardly ever necessary to actually compute the eigenstates.

It has been the discovery of discrete energy states, which gave quantum physics its name, as any state change from eigenstate ψ_n to ψ_m involves the exchange of an energy quantum $\Delta E = E_m - E_n$.

1.2.3.3 Electron in a Capacitor

As an example, let's consider an electron in a capacitor. To keep things simple, the capacitor should be modeled by an infinitely deep, one-dimensional potential well (see also 1.2.2.2), thus

$$V(\vec{r}) = V(x) = \begin{cases} 0 & \text{if } 0 < x < l \\ \infty & \text{otherwise} \end{cases} \tag{1.14}$$

This leads us the the time-independent Schrödinger Equation

$$-\frac{\hbar^2}{2m_e}\,\psi''(x) = E\,\psi(x) \tag{1.15}$$

and the boundary conditions

$$\psi(0) = 0 \quad \text{and} \quad \psi(l) = 0 \tag{1.16}$$

The ansatz $\psi(x) = N\sin(kx)$ automatically satisfies the first BC and leads to $k = \sqrt{2m_e E}/\hbar$. To satisfy the second BC, we have to introduce the quantization-condition $kl = n\pi$. The normalization constant N has to be chosen such that $\int |\psi(x)|^2 dx = 1$, thus the final result is

$$\psi_n(x) = \sqrt{\frac{2}{l}}\,\sin(k_n x) \quad \text{with} \quad k_n = \frac{\pi}{l}n \tag{1.17}$$

and the corresponding energies

$$E_n = \frac{\hbar^2 k_n^2}{2m_e} = \frac{\pi^2\hbar^2}{2m_e l^2}n^2, \quad n = 1, 2, \ldots\infty \tag{1.18}$$

The general time dependent solution is

$$\psi(x,t) = \sum_{n=1}^{\infty} c_n \psi_n(x,t) \quad \text{with} \quad \sum_{n=1}^{\infty} |c_n|^2 = 1 \quad \text{and} \tag{1.19}$$

$$\psi_n(x,t) = \sqrt{\frac{2}{l}}\,\mathrm{e}^{-\mathrm{i}\omega_n t}\sin(k_n x), \quad k_n = \frac{\pi}{l}n,\ \omega_n = \frac{E_n}{\hbar} \tag{1.20}$$

Figure 1.2 shows the first 3 eigenstates ψ_1, ψ_2 and ψ_3 and their corresponding spatial probability distributions $\rho_n = |\psi_n|^2$.

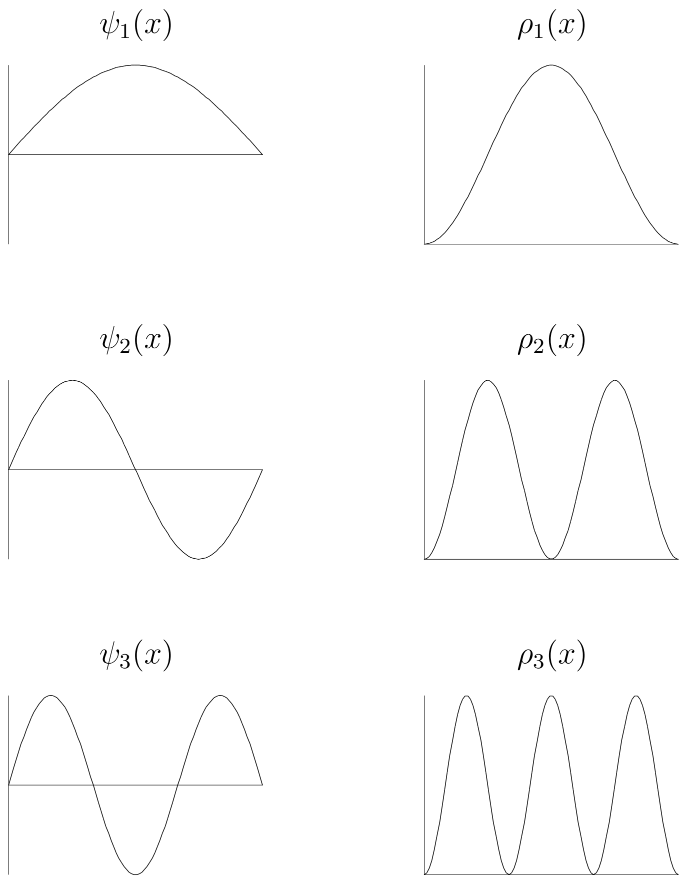

Figure 1.2: The first three eigenstates for an electron in a potential well

1.2.3.4 3-dimensional Trap

The above example can easily be extended to 3 dimensions, using the potential

$$V(\vec{r}) = V\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{cases} 0 & \text{if } 0 < x < l \text{ and } 0 < y < l \text{ and } 0 < z < l \\ \infty & \text{otherwise} \end{cases} \tag{1.21}$$

This leads to the eigenfunctions

$$\psi_{n_1,n_2,n_3}(\vec{r}) = \left(\frac{2}{l}\right)^{\frac{3}{2}} \sin(\frac{\pi}{l}n_1 x)\sin(\frac{\pi}{l}n_2 y)\sin(\frac{\pi}{l}n_3 z) \tag{1.22}$$

and the energies

$$E_{n_1,n_2,n_3} = E_{n_1+n_2+n_3} = \frac{\pi^2\hbar^2}{2m_e l^2}(n_1^2 + n_2^2 + n_3^2) \tag{1.23}$$

Since the different states can have the same energy (e.g. $E_{211} = E_{121} = E_{112}$) i.e. the eigenvalues of the Hamilton operator H are degenerated, measuring the energy is not sufficient for determining the actual electron distribution.

1.3 Algebraic Quantum Physics

While the Schrödinger Equation, in principle, allows to compute all details of the particle distribution and the exact energy terms, having to deal with partial differential equations, boundary conditions and normalization factors, is usually very cumbersome and often can't be done analytically, anyway.

Just a nobody would try to develop a color TV set by solving Maxwell equations, the discussion of complex quantum systems requires a more abstract formalism.

1.3.1 The Hilbert Space

1.3.1.1 States as Vectors

The solutions $\psi_n(x)$ from the examples in section 1.2.3.3 and 1.2.3.4 are complex functions over the intervals $\mathcal{I} = [0, l]$ or $\mathcal{I} = [0, l]^3$, respectively. Let's introduce the following abbreviations[8]

$$|n\rangle \equiv |\psi_n\rangle \equiv \psi_n(x) \quad \text{and} \quad \langle n| \equiv \langle\psi_n| \equiv \psi_n^*(x) \tag{1.24}$$

[8]This formalism is called *Braket notation* and has been introduced by Dirac: The $\langle\cdot|$ terms are referred to as "bra"- and the $|\cdot\rangle$ terms as "ket"-vectors.

or, for the case of k indices

$$|n_1, n_2, \ldots n_k\rangle \equiv \psi^*_{n_1 \ldots n_k}(\vec{r}) \quad \text{and} \quad \langle n_1, n_2, \ldots n_k| \equiv \psi_{n_1 \ldots n_k}(\vec{r}) \tag{1.25}$$

and also introduce a scalar Product $\langle\phi|\chi\rangle$ defined as

$$\langle\phi|\chi\rangle \equiv \int_{\mathcal{I}} \phi^*(\vec{r})\chi(\vec{r})d\vec{r} \tag{1.26}$$

The scalar product $\langle i|j\rangle$ of the eigenfunctions ψ_i and ψ_j from the on dimensional capacitor example (1.2.3.3) gives

$$\langle i|j\rangle = \int_{\mathcal{I}} \psi_i{}^*(x)\psi_j(x)dx = \frac{2}{l}\int_0^l \sin(\frac{\pi}{l}ix)\sin(\frac{\pi}{l}jx)dx \tag{1.27}$$

The substitution $\xi = \frac{\pi}{l}x$ leads to

$$\langle i|j\rangle = \frac{2}{\pi}\int_0^\pi \sin(i\xi)\sin(j\xi)d\xi = \delta_{ij} \tag{1.28}$$

So the eigenfunctions of the Hamilton operator H are orthonormal according to the scalar product (1.26) and therefor form the base of the orthonormal vector space $\mathcal{H}$ consisting of all possible linear combinations of $\{\psi_1, \psi_2, \ldots\}$. This space is the *Hilbert space* for this particular problem and it can be shown that the eigenvalues of any operator describing a physical observable form an orthogonal base.[9]

1.3.1.2 Completeness

Since the Schrödinger Equation is a linear differential equation, any linear combination of solutions is also a solution and thus a valid physical state. To calculate the expectation value $\langle H\rangle$ of the energy for a given state $\psi(x,t)$ we have to solve the integral

$$\langle H\rangle = \langle\psi|H|\psi\rangle = \int \psi^*(x,t)H\psi(x,t)dx \tag{1.29}$$

If $\psi(x,t)$ is given as a sum of eigenfunctions as in equation 1.19, integration can be avoided, as

$$\langle H\rangle = \overbrace{\sum_i c_i{}^*\langle i|}^{\langle\psi|}\, H \overbrace{\sum_j c_j|j\rangle}^{|\psi\rangle} = \sum_{ij} c_i{}^* c_j \langle i|H|j\rangle \tag{1.30}$$

[9]As physical observables are real values, their corresponding operators O have to be self-adjoint i.e. $O^\dagger = O$

Since $H\,|i\rangle = E_i|i\rangle$ and $\langle i|j\rangle = \delta_{ij}$, $\langle H\rangle$ can be expressed as a weighted sum of eigenvalues:

$$\langle H\rangle = \sum_i |c_i|^2 E_i \tag{1.31}$$

Using the eigenfunctions for the one-dimensional capacitor (1.2.3.3) the complex amplitudes c_i for an arbitrary continuous function $f(x)$ over $[0,l]$ are given by

$$c_i = \langle i|f\rangle = \sqrt{\frac{2}{l}} \int_0^l \sin(\frac{\pi}{l} ix) f(x) dx \tag{1.32}$$

This describes a standard sine-*Fourier Transform*. The original function can be reconstructed by a composition of eigenfunctions $\psi_n(x)$ with the Fourier components c_i

$$f(x) = \sum_i c_i \psi_i(x) = \sqrt{\frac{2}{l}} \sum_{i=1}^{\infty} c_i \sin(\frac{\pi}{l} ix) \tag{1.33}$$

As before, it can be shown that the eigenvalues of any Hamilton operator always form a complete orthonormal base, thus

$$I = \sum_i |i\rangle\langle i| \quad \text{with} \quad I|\psi\rangle = |\psi\rangle \tag{1.34}$$

1.3.1.3 Definitions

A Hilbert space $\mathcal{H}$ is a linear vector space over the scalar body $\mathbf{C}$. Let $|f\rangle, |g\rangle, |h\rangle \in \mathcal{H}$ and $\alpha, \beta \in \mathbf{C}$, then the following operations are defined [23]:

$$|f\rangle + |g\rangle \in \mathcal{H} \quad \text{linear combination} \tag{1.35}$$
$$\alpha|f\rangle \in \mathcal{H} \quad \text{scalar multiplication} \tag{1.36}$$
$$|f\rangle + |0\rangle = |f\rangle \quad \text{zero-element} \tag{1.37}$$
$$|f\rangle + |-f\rangle = |0\rangle \quad \text{inverse element} \tag{1.38}$$

The inner product $\langle\cdot|\cdot\rangle$ meets the following conditions:

$$\langle f|g+h\rangle = \langle f|g\rangle + \langle f|h\rangle \tag{1.39}$$
$$\langle f|\alpha g\rangle = \alpha\langle f|g\rangle \tag{1.40}$$
$$\langle f|g\rangle = (\langle g|f\rangle)^* \tag{1.41}$$
$$\langle f|f\rangle = 0 \iff |f\rangle = |0\rangle \tag{1.42}$$
$$||f|| \equiv \sqrt{\langle f|f\rangle} \geq 0 \tag{1.43}$$

1.3.2 Operators

1.3.2.1 Operators as Matrices

As we have shown in 1.3.1.2, all valid states ψ can be expressed as a sum of eigenfunctions, i.e.

$$\psi(\vec{r},t) = \sum_{i=0}^{\infty} c_i \psi_i(\vec{r},t) \tag{1.44}$$

If we use $\{\psi_0, \psi_1, \ldots\}$ as unit vectors, we can write the bra- and ket-vectors of ψ as infinitely dimensional row- and column-vectors

$$\langle\psi| \equiv ({c_0}^*, {c_1}^*, \ldots) \quad \text{and} \quad |\psi\rangle \equiv \begin{pmatrix} c_0 \\ c_1 \\ \vdots \end{pmatrix} \tag{1.45}$$

The time independent Schrödinger equation can then be written as

$$\begin{pmatrix} E_0 & 0 & 0 & \cdots \\ 0 & E_1 & 0 & \\ 0 & 0 & E_2 & \\ \vdots & & & \ddots \end{pmatrix} |\psi\rangle = E\,|\psi\rangle \tag{1.46}$$

The Hamilton Operator is the diagonal matrix $H = \operatorname{diag}(E_0, E_1, \ldots)$. In the case of multiple indices as in 1.2.3.4, a diagonalization such as e.g. $\{\psi_{000}, \psi_{100}, \psi_{010}, \psi_{001}, \psi_{110}, \ldots\}$, can be used to order the eigenfunctions. If such an diagonalization exists for a Hilbert space $\mathcal{H}$, then every linear operator O of $\mathcal{H}$ can be written in matrix form with the matrix elements $O_{ij} = \langle i|O|j\rangle$.

$$O = \begin{pmatrix} O_{00} & O_{01} & \cdots \\ O_{10} & O_{11} & \cdots \\ \vdots & & \ddots \end{pmatrix} \quad \text{with} \quad O_{ij} = \langle i|O|j\rangle \tag{1.47}$$

1.3.2.2 Physical Observables

As has been mentioned in 1.2.2.3, in quantum physics, a physical observable $\mathcal{O}$ is expressed as a linear operator O (see table 1.1) while the classical value of $\mathcal{O}$ is the expectation value $\langle O\rangle$. Obviously, the value of an observable such as position or momentum must be real, as a length of $(1+\mathrm{i})$ meter would have no physical meaning, so we require $\langle O\rangle \in \mathbf{R}$.

$O^\dagger$ is called *adjoint operator* to O if

$$\langle\hat{f}|g\rangle = \langle f|O|g\rangle \quad \text{with} \quad |\hat{f}\rangle = O^\dagger|f\rangle \tag{1.48}$$

If O is given in matrix form, the $O^\dagger$ is the conjugated transposition of O, i.e. $O^\dagger = (O^{\mathrm{T}})^*$. An operator O with $O^\dagger = O$ is called *self adjoint* or *Hermitian.*

All quantum observables are represented by Hermitian operators as we can reformulate the requirement $\langle O \rangle \in \mathbf{R}$ as $\langle O \rangle = \langle O \rangle^*$ or

$$\langle \psi | O | \psi \rangle = (\langle \psi | O | \psi \rangle)^* = \langle \psi | O^\dagger | \psi \rangle \tag{1.49}$$

1.3.2.3 Measurement

In classical physics, the observables of a system such as particle location, momentum, Energy, etc. where thought to be well defined entities which change their values over time according to certain dynamic laws and which could — technical difficulties aside — in principle be observed without disturbing the system itself. It is a fundamental finding of quantum physics that this is not the case.

- **Measured Values:** Measured values o_i are always eigenvalues of their according operator O.
- **Probability Spectrum:** If the eigenvalue o_i isn't degenerated and has the eigenvector ψ_i, then the probability to measure o_i is $p_i = |\langle \psi_i | \psi \rangle|^2$. If the eigenvalue o_i is d_i-fold degenerated and $\{\psi_{i,1}, \psi_{i,2}, \ldots \psi_{i,d_i}\}$ is an orthonormal base of the according eigenspace, then

$$p_i = \sum_{j=1}^{d_i} |\langle \psi_{ij} | \psi \rangle|^2 \tag{1.50}$$

- **Reduction of the Wave Function:** If the eigenvalue o_i isn't degenerated, the post-measurement state $|\psi'\rangle = |\psi_i\rangle$, otherwise

$$|\psi'\rangle = \frac{1}{\sqrt{p_i}} \sum_{j=1}^{d_i} |\psi_{ij}\rangle \langle \psi_{ij} | \psi \rangle \tag{1.51}$$

Consider a state $|\psi\rangle$ which is a composition of two eigenstates $|\psi_1\rangle$ and $|\psi_2\rangle$ of the time-independent Schrödinger equation with the assorted energy-eigenvalues E_1 and E_2

$$|\psi\rangle = c_1 |\psi_1\rangle + c_2 |\psi_2\rangle \quad \text{with} \quad |c_1|^2 + |c_2|^2 = 1 \tag{1.52}$$

The expectation value of energy $\langle H \rangle = |c_1|^2 E_1 + |c_2|^2 E_2$, but if we actually perform the measurement, we will measure either E_1 or E_2 with the probabilities $|c_1|^2$ and $|c_2|^2$. However, if we measure the resulting state again, we will always get the same energy as in the first measurement as the wave function has collapsed to either ψ_1 or ψ_2.

$$|\psi\rangle \to \begin{cases} |\psi_1\rangle & \text{with probability } |c_1|^2 \\ |\psi_2\rangle & \text{with probability } |c_2|^2 \end{cases} \tag{1.53}$$

The fact that $\langle H \rangle$ is only a statistical value, brings up the question when it is reasonable to speak about the energy of a state (or any other observable, for the matter) or, with other words, whether a physical quality of a system exists for itself or is invariably tied to the process of measuring.

The *Copenhagen interpretation* of quantum physics argues that an observable $\mathcal{O}$ only exists if the system in question happens to be in an eigenstate of the according operator O [22].

1.3.2.4 The Uncertainty Principle

The destructive nature of measurement raises the question whether 2 observables $\mathcal{A}$ and $\mathcal{B}$ can be measured simultaneously. This can only be the case if the post-measurement state ψ' is an eigenfunction of A and B

$$A|\psi'\rangle = a|\psi'\rangle \quad \text{and} \quad B|\psi'\rangle = b|\psi'\rangle \tag{1.54}$$

Using the *commutator* $[A, B] = AB - BA$, this is equivalent to the condition $[A, B] = 0$. If A and B don't commute, then the uncertainty product (see 1.2.2.3) $(\Delta A)(\Delta B) > 0$. To find a lower limit for $(\Delta A)(\Delta B)$ we introduce the operators $\delta A = A - \langle A \rangle$ and $\delta B = B - \langle B \rangle$ and can express the squared uncertainty product as

$$(\Delta A)^2(\Delta B)^2 = \langle(\delta A)^2\rangle\langle(\delta B)^2\rangle = \langle\psi|(\delta A)(\delta A)|\psi\rangle\langle\psi|(\delta B)(\delta B)|\psi\rangle \tag{1.55}$$

Since δA and δB are self adjoint, we express the above as $(\Delta A)^2(\Delta B)^2 = ||\delta A\psi||^2||\delta B\psi||^2$. Using Schwarz's Inequality $||f||^2||g||^2 \geq ||fg||^2$ and the fact that $[A, B] = [\delta A, \delta B]$ we get

$$(\Delta A)(\Delta B) \geq \frac{1}{2}||[A, B]|| \tag{1.56}$$

Observables with a nonzero commutator $[A, B]$ of the dimension of *action* (i.e. a product of energy and time) are *canonically conjugated*. If we take e.g. the location and momentum operators from 1.2.2.3, we find that

$$(\Delta R_i)(\Delta P_j) \geq \frac{1}{2}||[r_i, -i\hbar\frac{\partial}{\partial r_j}]|| = \frac{1}{2}\hbar\delta_{ij} \tag{1.57}$$

This means that it is impossible to define the location and the impulse for the same coordinate to arbitrary precision; it is, however, possible the measure the location in x-direction together with the impulse in y-direction.

1.3.2.5 Temporal Evolution

In 1.2.3.1 we have shown how the Schrödinger equation can be separated if the Hamilton operator is time independent.

If we have the initial value problem with $\psi(t=0)=\psi_0$ we can define an operator $U(t)$ such that

$$HU(t)\,|\psi_0\rangle = i\hbar\frac{\partial}{\partial t}U(t)\,|\psi_0\rangle \quad \text{and} \quad U(0)|\psi\rangle = |\psi\rangle \tag{1.58}$$

We get the operator equation $HU = i\hbar\frac{\partial}{\partial t}U$ with the solution

$$U(t) = \mathrm{e}^{-\frac{\mathrm{i}}{\hbar}Ht} = \sum_{n=0}^{\infty}\frac{1}{n!}\,\frac{(-i)^n t^n}{\hbar^n}H^n \tag{1.59}$$

U is the *operator of temporal evolution* and satisfies the criterion

$$U(t)\,|\psi(t_0)\rangle = |\psi(t_0+t)\rangle \tag{1.60}$$

If $|\psi\rangle = \sum_i c_i|i\rangle$ is a solution of the time-independent Schrödinger equation, then

$$|\psi(t)\rangle = U(t)\,|\psi\rangle = \sum_i c_i \mathrm{e}^{-\mathrm{i}\omega_i t}|i\rangle \quad \text{with} \quad \omega_i = \frac{E_i}{\hbar} \tag{1.61}$$

is the corresponding time dependent solution (see 1.2.3.1).

1.3.2.6 Unitary Operators

The operator of temporal evolution satisfies the condition

$$U^\dagger(t)U(t) = \mathrm{e}^{\frac{\mathrm{i}}{\hbar}Ht}\mathrm{e}^{-\frac{\mathrm{i}}{\hbar}Ht} = 1 \tag{1.62}$$

Operators U with $U^\dagger = U^{(-1)}$ are called *unitary*. Since the temporal evolution of a quantum system is described by a unitary operator and $U^\dagger(t) = U(-t)$ it follows that the temporal behavior of a quantum system is *reversible*, as long a no measurement is performed.[10]

Unitary operators can also be used to describe abstract operations like rotations

$$R_z(\alpha)\,|n_1,n_2,n_3\rangle = \cos(\alpha)|n_1,n_2,n_3\rangle + \mathrm{i}\sin(\alpha)|n_2,n_1,n_3\rangle \tag{1.63}$$

[10] since a measurement can result in a reduction of the wave-function (see 1.3.2.3), it is generally impossible to reconstruct $|\psi\rangle$ from the post-measurement state $|\psi'\rangle$

or the flipping of eigenstates

$$\mathrm{Not}\,|n\rangle = \begin{cases} |1\rangle & \text{if } n = 0 \\ |0\rangle & \text{if } n = 1 \\ |n\rangle & \text{otherwise} \end{cases} \tag{1.64}$$

without the need to specify how this transformations are actually performed or having to deal with time-dependent Hamilton operators.

Mathematically, unitary operations can be described as base-transformations between 2 orthonormal bases (just like rotations in $\mathbf{R}^3$). Let A and B be Hermitian operators with the orthonormal eigenfunctions ψ_n and $\tilde{\psi}_n$ and $|\psi\rangle = \sum_i c_i|\psi_i\rangle = \sum_i \tilde{c}_i|\tilde{\psi}_i\rangle$, then the *Fourier coefficients* $\tilde{c}_i$ are given by

$$\begin{pmatrix} \tilde{c}_0 \\ \tilde{c}_1 \\ \vdots \end{pmatrix} = U \begin{pmatrix} c_0 \\ c_1 \\ \vdots \end{pmatrix} \quad \text{with} \quad U = \sum_{i,j} |\tilde{\psi}_i\rangle\langle\tilde{\psi}_i|\psi_j\rangle\langle\psi_j| \tag{1.65}$$

1.3.3 Composed systems

1.3.3.1 Spin

In section 1.2.3.4 we have calculated the eigenstates ψ_{n_1,n_2,n_3} for an electron in a 3 dimensional trap. Real electrons are also characterized by the orientation of their spin which can be either "up" ($\uparrow$) or "down" ($\downarrow$). The spin-state $|\chi\rangle$ of an electron can therefor be written as

$$|\chi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \alpha|\uparrow\rangle + \beta|\downarrow\rangle \quad \text{with} \quad |\alpha|^2 + |\beta|^2 = 1 \tag{1.66}$$

The spins also form a finite Hilbert space $\mathcal{H}_S = \mathbf{C}^2$ with the orthonormal base $\{|\uparrow\rangle, |\downarrow\rangle\}$. If we combine $\mathcal{H}_S$ with the solution space $\mathcal{H}_R$ for the spinless problem (equation 1.22), we get a combined Hilbert space $\mathcal{H} = \mathcal{H}_R \times \mathcal{H}_B$ with the base-vectors

$$|n_1, n_2, n_3, s\rangle = |\psi_{n_1,n_2,n_3}\rangle|s\rangle \quad \text{with} \quad n_1, n_2, n_3 \in \mathbf{N},\ s \in \{\uparrow, \downarrow\} \tag{1.67}$$

1.3.3.2 Product States

If we have two independent quantum systems A and B described by the Hamilton operators H_A and H_b with the orthonormal eigenvectors ψ_i^A and ψ_j^B, which are in the states

$$|\psi^A\rangle = \sum_i a_i|\psi_i^A\rangle \quad \text{and} \quad |\psi^B\rangle = \sum_j b_j|\psi_j^B\rangle \tag{1.68}$$

then the common state $|\Psi\rangle$ is given by

$$|\Psi\rangle = |\psi^A\rangle|\psi^B\rangle = \sum_{i,j} a_i b_j |\psi_i^A\rangle|\psi_j^B\rangle = \sum_{i,j} a_i b_j |i,j\rangle \tag{1.69}$$

Such states are called *product states*. Unitary transformations and measurements applied to only one subsystem don't affect the other as

$$U^A|\Psi\rangle = (U \times I)|\psi^A\rangle|\psi^B\rangle = \sum_{i,j,k,l} U_{ik} a_k \delta_{jl} b_l |i,j\rangle = \left(U|\psi^A\rangle\right)|\psi^B\rangle \tag{1.70}$$

and the probability p_i^A to measure the energy E_i^A in system A is given by[11]

$$p_i^A = \left|\left(\langle\psi_i^A|\langle\psi^B|\right)|\Psi\rangle\right|^2 = \left|\sum_{j,k,l} b_j^* a_k b_l \langle i,j|k,l\rangle\right|^2 = \left|a_i \sum_j b_j^* b_j\right|^2 = |a_i|^2 \tag{1.71}$$

1.3.3.3 Entanglement

If $|\Psi\rangle$ is not a product state, then operations on one subsystem can affect the other. Consider two electrons with the common spin state

$$|\Psi\rangle = \frac{1}{\sqrt{2}}\left(|\uparrow\downarrow\rangle + |\downarrow\uparrow\rangle\right) \tag{1.72}$$

If we measure the spin of the first electron, we get either $|\uparrow\rangle$ or $|\downarrow\rangle$ with the equal probability $p = 1/2$ which the resulting post-measurement states $|\uparrow\downarrow\rangle$ or $|\downarrow\uparrow\rangle$. Consequently, if we measure the spin of the second electron, we will always find it to be anti-parallel to the first.

Two systems whose common wave-function $|\Psi\rangle$ is not a product state are *entangled*.

[11] We assume here that the eigenvalue E_i^A isn't degenerated, otherwise the solution is analog to equation 1.50.

Chapter 2

Quantum Computers

The application of quantum physical principles to the field of computing leads to the concept of the quantum computer, in which data isn't stored as bits in conventional memory, but as the combined quantum state of many 2-state systems of qubits.

This chapter introduces the theoretical foundations, components and basic operations of a quantum computer as well several models of quantum computation.

2.1 Introduction

2.1.1 The Church-Turing Thesis

The basic idea of modern computing science is the view of computation as a mechanical, rather than a purely mental process. A method, or procedure $\mathcal{M}$ for achieving some desired result is called *effective* or *mechanical* just in case [17]

1. $\mathcal{M}$ is set out in terms of a finite number of exact instructions (each instruction being expressed by means of a finite number of symbols);

2. $\mathcal{M}$ will, if carried out without error, always produce the desired result in a finite number of steps;

3. $\mathcal{M}$ can (in practice or in principle) be carried out by a human being unaided by any machinery save paper and pencil;

4. $\mathcal{M}$ demands no insight or ingenuity on the part of the human being carrying it out.

Alan Turing and Alonzo Church both formalized the above definition by introducing the concept of *computability by Turing machine* and the mathematically equivalent concept of *recursive functions* with the following conclusions:

Turing's thesis LCMs *[logical computing machines i.e. Turing machines]* can do anything that could be described as "rule of thumb" or "purely mechanical". [19]

Church's thesis A function of positive integers is effectively calculable only if recursive. [18]

As the above statements are equivalent, they are commonly referred to as the *Church-Turing Thesis* which defines the scope of classical computing science.

2.1.2 Computing Machines

Despite its operationalistic approach, the above computability concept doesn't have much in common with the continuous nature of physics, so in order to build a computing machine $\mathcal{M}$, we have to introduce a labeling function m which maps the analog physical states $S(t)$ (e.g. the tension of a capacitor) to digital computational states $s = (S(t))$ (e.g. the value of a bit) The digital states have to be strings over some finite alphabet Σ.

Since the above definition of computability requires a finite number of both, symbols and instructions, the labeling function only needs to apply on discrete intermediate machine states $S(t_0), S(t_1), \ldots$ so the temporal evolution of the machine state $S(t)$ is mapped onto a sequence of computational states $\{s_0, s_1, \ldots s_n\}$ where each transition $s_i \rightarrow s_{i+1}$ corresponds to one function $I_i : \Sigma^* \rightarrow \Sigma^*$ from a enumerable set $\mathbf{I}$ of (simple) instructions.[1] The sequence $\pi = \{I_0, I_1, \ldots I_{n-1}\}$ is called *program*.

[1]For hypothetical machines with unlimited memory, the instruction set $\mathbf{I}$ might also be infinitely which is not in accordance with Turing's original definition of computability.

The Turing Machine avoids this problem by extending the computational states s_i with an integer p and using this "head position" as an additional parameter to the generated instructions. As p (which can get arbitrarily large) would be "stored" in the physical position of the head and not in the state of the head itself, it can still be claimed that the TM operates *"by finite means"*.

Even with our simpler model, we could avoid an infinite instruction set, e.g. by interpreting a state $s = \underline{\mathbf{1}}^p \underline{\mathbf{0}} t$ with $t \in \Sigma^*$ as the pair $s = (p, t)$, and define the instructions as $I_i(p, t) : \mathbf{N}_0 \times \Sigma^* \rightarrow \mathbf{N}_0 \times \Sigma^*$.

As we are discussing physical computers, which usually don't have unlimited memory, The states s_0 and s_n are called the *input-* and the *output-state*. The machine $\mathcal{M} = (S, m, \Sigma, \pi)$ thus implements the function

$$f(s_0) = (I_0 \circ I_1 \circ \ldots \circ I_{n-1})(s_0) \quad \text{with} \quad s_0 = m(S(0)) \in \Sigma^* \tag{2.1}$$

2.1.3 Computation as a Physical Process

The above definition of a computing machine poses severe restrictions on the interpretation of physical states. If we consider computation as a physical process, rather than a "mechanical" manipulation of symbols as defined in 2.1.1, we can drop all restrictions in the above definition which don't have a physical equivalent.

2.1.3.1 Indeterminism

As we have showed in 1.3.2.3, the measurement of an observable O with the according operator O is only deterministic, if a system is in an eigenstate of O. To account for the stochastic nature of quantum measurement, we have to replace the labeling function m by a probabilistic operator $M : \mathcal{H} \rightarrow \Sigma^*$ which randomly chooses a string s according to some probability distribution $\delta_\Psi : s \rightarrow [0, 1]$ with $\sum_s \delta_\Psi(s) = 1$.

2.1.3.2 Temporal Evolution

Since it is not possible to non-destructively measure a quantum system and we are only interested in the result of a computation, anyway, it is not necessary that a labeling is defined for the intermediate steps $\Psi(t_1)$ to $\Psi(t_{n-1})$ of a computation i.e. it is not required to "watch" the temporal evolution of the system, as long as a labeling for the input- and output-state Ψ_0 and Ψ_n is given.

While the transitions $\Psi(t_i) \rightarrow \Psi(t_{i+1})$ still have to correspond to (simple) operations U_i from a enumerable instruction set of quantum transformations, the operators U_i, don't have to directly correspond to functions in Σ^*.[2]

In 1.3.2.6 we have shown that the temporal evolution of a quantum system is mathematically described by unitary operators, so a *quantum program* $\pi = \{U_0, U_1, \ldots U_{n-1}\}$ is a composition of elementary unitary transformations.

we can ignore this problem, and use the simpler and more general computer definition given above.

[2]Because of the reversibility of unitary operators, a direct correspondence would only be possible for bijective functions $f : \Sigma^* \rightarrow \Sigma^*$

2.2 Components of a Quantum Computer

A classical, as well as a quantum computer, essentially consists of 3 parts: a memory, which holds the current machine state, a processor, which performs elementary operations on the machine state, and some sort of input/output which allows to set the initial state and extract the final state of the computation.

2.2.1 Quantum Memory

2.2.1.1 The Qubit

The quantum analogy to the classical bit is the quantum bit or *qubit*. Just as a classical bit is represented by a system which can adopt one of two distinct states "0" and "1" we can define a quantum bit as follows:

Definition 1 (Qubit) *A qubit or quantum bit is a quantum system whose state can be fully described by a superposition of two orthonormal eigenstates labeled* $|0\rangle$ *and* $|1\rangle$*.*

The general state $|\psi\rangle \in \mathcal{H}$ of a qubit is given by

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \quad \text{with} \quad |\alpha|^2 + |\beta|^2 = 1 \tag{2.2}$$

The value of a qubit is the observable $\mathcal{N}$ with the Hermitian operator $N\,|i\rangle = i\,|i\rangle$ over the Hilbert space $\mathcal{H} = \mathbf{C}^2$, or in matrix representation

$$N = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \tag{2.3}$$

The expectation value of N is given by

$$\langle N \rangle = \langle \psi | N | \psi \rangle = \begin{pmatrix} \alpha^* & \beta^* \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = |\beta|^2 \tag{2.4}$$

thus, $\langle N \rangle$ gives the probability to find the system in state $|1\rangle$ if a measurement is performed on the qubit.

2.2.1.2 Combination of Qubits

If we combine 2 qubits, the general state of the resulting system is

$$|\Psi\rangle = \alpha|00\rangle + \beta|10\rangle + \gamma|01\rangle + \delta|11\rangle \quad \text{with} \quad |\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1 \tag{2.5}$$

While we still can define distinct observables $\mathcal{N}^{(1)}$ and $\mathcal{N}^{(2)}$ for the value of each qubit with the operators $N^{(1)}|ij\rangle = i\,|ij\rangle$ and $N^{(2)}|ij\rangle = j\,|ij\rangle$, their expectation values

$$\langle N^{(1)}\rangle = |\beta|^2 + |\delta|^2 \quad \text{and} \quad \langle N^{(2)}\rangle = |\gamma|^2 + |\delta|^2 \tag{2.6}$$

don't allow us to reconstruct the actual probability distribution among the eigenvalues. To illustrate this, consider the states

$$|\Psi_A\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle), \quad |\Psi_B\rangle = \frac{1}{\sqrt{2}}(|10\rangle + |01\rangle) \tag{2.7}$$

$$\text{and} \quad |\Psi_C\rangle = \frac{1}{2}(|00\rangle + |10\rangle + |01\rangle + |11\rangle)$$

All of these states have the expectation values $\langle N^{(1)}\rangle = \langle N^{(2)}\rangle = 1/2$, i.e. if we measure a single qubit, we get either $|0\rangle$ or $|1\rangle$ with equal probability; we get, however, totally different post-measurement states.

If we measure $|1\rangle$ in the first qubit, the resulting post-measurement states are

$$|\Psi'_A\rangle = |11\rangle, \quad |\Psi'_B\rangle = |10\rangle \quad \text{and} \quad |\Psi'_C\rangle = \frac{1}{\sqrt{2}}(|10\rangle + |11\rangle) \tag{2.8}$$

and the expectation values for the second qubit are now given by

$$\langle N^{(2)}_A\rangle = 1, \quad \langle N^{(2)}_B\rangle = 0 \quad \text{and} \quad \langle N^{(2)}_C\rangle = \frac{1}{2} \tag{2.9}$$

2.2.1.3 Machine State

While the state of a classical computer can be given as the distinct states of all bits in memory and processor registers, the "state of a qubit" is a meaningless term, if the machine state is the combined state of more than one system.[3]

Definition 2 (Machine State) *The machine state Ψ of an n-qubit quantum computer is the current state of a combined system of n identical qubit subsystems.*

Generally, the machine state Ψ of an n-qubit quantum computer is given by

$$|\Psi\rangle = \sum_{(d_0 \dots d_{n-1}) \in \mathbf{B}^n} c_{d_0 \dots d_{n-1}} |d_0 \dots d_{n-1}\rangle \quad \text{with} \quad \sum |c_{d_0 \dots d_{n-1}}|^2 = 1 \tag{2.10}$$

[3]Unless the machine state happens to be a product state, that is (see 1.3.3.2).

The combined Hilbert space $\mathcal{H}$ is thus a composition of n 1-qubit-Hilbert spaces $\mathcal{H}_i = \mathbf{C}^2$, i.e.

$$\mathcal{H} = \mathcal{H}_1 \times \mathcal{H}_2 \times \ldots \times \mathcal{H}_n = \mathbf{C}^{2^n} \tag{2.11}$$

If we interpret the eigenvectors $|d_0 \ldots d_{n-1}\rangle$ as binary digits, with d_0 as least significant bit, we can write this as

$$|\Psi\rangle = \sum_{i=0}^{2^n-1} c_i |i\rangle \quad \text{with} \quad |d_0 + 2d_1 + \ldots + 2^{n-1} d_{n-1}\rangle \equiv |d_0 \ldots d_{n-1}\rangle \tag{2.12}$$

The operator N_i for value $\mathcal{N}_i$ of the i-th qubit is given by

$$N_i \, |d_0 \ldots d_{n-1}\rangle = d_i \, |d_0 \ldots d_{n-1}\rangle \tag{2.13}$$

and has the expectation value

$$\langle N_i \rangle = \sum_{(d_0 \ldots d_{n-1}) \in \mathbf{B}^n} d_i |c_{d_0 \ldots d_{n-1}}|^2 \tag{2.14}$$

2.2.1.4 Subsystems

As we have shown above, the memory of an n-qubit quantum computer is a combined system of n identical qubit-subsystems. Since the partition into subsystems is merely methodical, we can consider the first m qubits $(m < n)$ as a single subsystem and write Ψ as

$$|\Psi\rangle = \sum_{i=0}^{2^m-1} \sum_{j=0}^{2^{n-m}-1} c_{ij} |i, j\rangle \quad \text{with} \quad |\Psi\rangle \in \mathcal{H} = \mathbf{C}^{2^n} \tag{2.15}$$

As the base vectors $|i, j\rangle$ are product states $|i, j\rangle = |i\rangle|j\rangle$, the Hilbert space $\mathcal{H}$ can be written as a combination of

$$\mathcal{H} = \mathcal{H}' \times \mathcal{H}'' \quad \text{with} \quad \mathcal{H}' = \mathbf{C}^{2^m} \quad \text{and} \quad \mathcal{H}'' = \mathbf{C}^{2^{n-m}} \tag{2.16}$$

Let U' and U'' be unitary operators over $\mathcal{H}'$ and $\mathcal{H}''$, then the commutator $[U', U''] = 0$ as

$$[U', U''] \, |\Psi\rangle = \sum_{i,j} c_{ij} \, [U'|i\rangle(U''|j\rangle) - U''(U'|i\rangle)|j\rangle] = 0 \tag{2.17}$$

This means that unitary transformations applied to distinct subsets of qubits are independent.

A unitary transformation U' over the first m qubits also doesn't affect a measurement of the remaining qubits since the probability p''_j to measure j in the remaining $n - m$ qubits, i.e. to get a post-measurement state of the form $|\Psi'\rangle = |\psi_j\rangle|j\rangle$, is invariant to U', as

$$p''_j = \sum_i c^*_{ij} c_{ij} \langle i|i\rangle\langle j|j\rangle = \sum_i c^*_{ij} c_{ij} \langle i|{U'}^\dagger U'|i\rangle\langle j|j\rangle \tag{2.18}$$

2.2.1.5 Quantum Registers

The above notion of m-qubit subsystems can easily be extended to arbitrary sequences of qubits.

Definition 3 (Quantum Register) *An m qubit quantum Register* $\mathbf{s}$ *is a sequence of mutually different zero-based qubit positions $\langle s_0, s_1 \dots s_{m-1}\rangle$ of some machine state $|\Psi\rangle \in \mathbf{C}^{2^m}$ with $n \geq m$.*

Reordering Operators Let $\mathbf{s}$ be an m qubit register of the n qubit state $|\Psi\rangle$. Using an arbitrary permutation π over n elements with $\pi_i = s_i$ for $i < m$, we can construct a unitary *reordering operator* $\Pi_\mathbf{s}$ by permutating the qubits.

$$\Pi_\mathbf{s}\,|d_0, d_1 \dots d_{n-1}\rangle = |d_{\pi_0}, d_{\pi_1} \dots d_{\pi_{n-1}}\rangle \tag{2.19}$$

Note that there exist $(n-m)!$ permutations $\Pi_\mathbf{s}^{(k)}$ which fit into the above definition, since π_i is only defined for $i < m$.

Unitary operators correspond to base transformations (see 1.3.2.6), so we can write $|\tilde{\Psi}\rangle = \Pi_\mathbf{s}|\Psi\rangle$ as

$$|\tilde{\Psi}\rangle = \sum_{i=0}^{2^m-1}\sum_{j=0}^{2^{n-m}-1} \tilde{c}_{ij}|i,j\rangle \quad \text{with} \quad \tilde{c}_{ij} = c_{\tilde{\imath}\tilde{\jmath}} \quad \text{and} \quad |\tilde{\imath},\tilde{\jmath}\rangle = \Pi_\mathbf{s}|i,j\rangle \tag{2.20}$$

As above, the transformed Hilbert space $\tilde{\mathcal{H}}$ can be written as a combination

$$\tilde{\mathcal{H}} = \tilde{\mathcal{H}}' \times \tilde{\mathcal{H}}'' \quad \text{with} \quad \tilde{\mathcal{H}}' = \mathbf{C}^{2^m} \quad \text{and} \quad \tilde{\mathcal{H}}'' = \mathbf{C}^{2^{n-m}} \tag{2.21}$$

Unitary Operators Let $\tilde{U}'$ be a m-qubit unitary operator over $\tilde{\mathcal{H}}'$ and $\tilde{U} = \tilde{U}' \times I(n-m)$ with $I(k)$ being the k-qubit identity operator.

$$\tilde{U}\,|\tilde{\Psi}\rangle = \sum_{i,j} \tilde{c}_{ij}(\tilde{U}'|i\rangle)|j\rangle \tag{2.22}$$

For each permutation $\Pi_\mathbf{s}^{(k)}$, we can define a back-transformed operator

$$U^{(k)} = \Pi_\mathbf{s}^{(k)\dagger}\tilde{U}\,\Pi_\mathbf{s}^{(k)} = \sum_{i',j',i,j} |i',j'\rangle\, u^{(k)}_{i'j'ij}\,\langle i,j| \tag{2.23}$$

with the matrix elements

$$u^{(k)}_{i'j'ij} = \langle\tilde{\imath}'^{(k)}|\tilde{U}'|\tilde{\imath}^{(k)}\rangle\langle\tilde{\jmath}'^{(k)}|\tilde{\jmath}^{(k)}\rangle \quad \text{and} \quad |\tilde{\imath}^{(k)},\tilde{\jmath}^{(k)}\rangle = \Pi_\mathbf{s}^{(k)}|i,j\rangle \tag{2.24}$$

Since the transformed base vectors $\tilde{\imath}^{(k)}$ are identical for all permutations $\Pi_\mathbf{s}^{(k)}$ and $\langle\tilde{\jmath}'^{(k)}|\tilde{\jmath}^{(k)}\rangle = \delta_{j'j}$, it follows that the back-transformation $\tilde{U}' \times I \rightarrow U$ is independent from the chosen permutation $\Pi_\mathbf{s}^{(k)}$.

Register Observables Just as with single qubits, we can define an observable $\mathcal{S}$ for a given m-qubit register $\mathbf{s}$ with the operator

$$S = (N_{\pi_0}, N_{\pi_1}, \dots N_{\pi_{m-1}}) \quad \text{and} \quad N_i \,|d_0 \dots d_{n-1}\rangle = d_i \,|d_0 \dots d_{n-1}\rangle \tag{2.25}$$

or, if we interpret the binary vectors as integers,

$$S = \sum_{i,j} \Pi_{\mathbf{s}}{}^{\dagger} \,|i,j\rangle i \langle i,j|\, \Pi_{\mathbf{s}} = \sum_{i,j} |i,j\rangle \tilde{\imath} \langle i,j| \tag{2.26}$$

2.2.2 Processing Units

2.2.2.1 Unitary Operators

In a classical n-bit computer, every computational step can be described by a transition function $I : \mathbf{B}^n \to \mathbf{B}^n$ which takes the current state S of all bits as input and returns the appropriate post-instruction state S'.

As we have shown in 1.3.2.6, the temporal evolution of a quantum system can be described by unitary operators. The general form of a n-qubit unitary operator U over the Hilbert space $\mathcal{H} = \mathbf{C}^{2^n}$ is

$$U = \sum_{i=0}^{2^n-1} \sum_{j=0}^{2^n-1} |i\rangle\, u_{ij}\, \langle j| \quad \text{with} \quad \sum_{k=0}^{2^n-1} u_{ki}^* u_{kj} = \delta_{ij} \tag{2.27}$$

If we compare boolean functions to unitary operators from a strictly functional point of view we can identify three major differences between classical and quantum operations:

- **Reversibility:** Since unitary operators, by definition, match the condition $U^\dagger U = I$, for every transformation U there exists the inverse transformation $U^\dagger$. As a consequence, quantum computation is restricted to reversible functions.[4]

- **Superposition:** An eigenstate $|\Psi\rangle = |k\rangle$ can be transformed into a superposition of eigenstates.

$$|\Psi'\rangle = U\,|k\rangle = \sum_{k'} U_{k'k} |k\rangle \tag{2.28}$$

  The mathematical explanation of this feature lies in the fact that the requirement $\langle i|U^\dagger U|j\rangle = \delta_{ij}$ is weaker than the pseudo-classical (see 2.2.2.4) condition

$$\langle i|U^\dagger|\pi_i\rangle\langle\pi_i|\pi_j\rangle\langle\pi_j|U|j\rangle = \delta_{ij} \tag{2.29}$$

[4]A classical analogon would be the class of reversible boolean functions

which requires transformed eigenstates not only to be orthonormal, but also to be of the form $U\,|k\rangle = |\pi_k\rangle$ with some appropriate permutation (i.e. reversible function) π over $\mathbf{Z}_{2^n}$.

- **Parallelism:** If the machine state $|\Psi\rangle$ already is a superposition of several eigenstates, then a transformation U is applied to all eigenstates simultaneously.

$$U \sum_i c_i|i\rangle = \sum_i c_i U|i\rangle \tag{2.30}$$

This feature of quantum computing is called *quantum parallelism* and is a consequence of the linearity of unitary transformations.

2.2.2.2 Register Operators

The basic instructions of a classical computer usually operate only on a very small number of bits and are typically independent from the total amount of available memory. Therefor it is more useful to describe those instructions not as boolean functions F over the whole state space $\mathbf{B}^n$ (in the case of an n bit machine), but as parameterized functions $f_{\mathbf{x}}$ over $\mathbf{B}^m$, where the vector $\mathbf{x} \in \mathbf{Z}^n$ only holds the bit-positions of the relevant arguments. Consequently we refer to the resulting instruction F as "applying f to the bits $x_0, x_1 \dots x_{n-1}$".

While it is clear what we mean by e.g. "swapping the bits 3 and 5" on a classical computer, we cannot blindly adopt this concept to quantum computing, because unitary operators operate on states and a single qubit doesn't have a state.[5]

In 2.2.1.5 we have defined a *quantum register* as a sequence of (mutually different) qubit-positions $\mathbf{s} = \langle s_0, s_1 \dots s_{m-1}\rangle$, which is the quantum analogon to the above argument vector $\mathbf{v}$, and a class of $(n-m)!$ reordering operators $\Pi_{\mathbf{s}}^{(k)}$ which meet the condition

$$\Pi_{\mathbf{s}}^{(k)}\,|d_0, d_1 \dots d_{n-1}\rangle = |d_{s_0}, d_{s_1} \dots d_{s_{m-1}}\rangle|\tilde{\jmath}^{(k)}\rangle \tag{2.31}$$

Definition 4 (Register Operator) *The register operator $U(\mathbf{s})$ for an m-qubit unitary operator $U : \mathbf{C}^{2^m} \to \mathbf{C}^{2^m}$ and a m-qubit quantum register $\mathbf{s}$ on an n-qubit quantum computer is the n-qubit operator*

$$U(\mathbf{s}) = \Pi_{\mathbf{s}}^{\dagger}\,(U \times I(n-m))\,\Pi_s \tag{2.32}$$

with an arbitrary reordering operator $\Pi_{\mathbf{s}}$

[5] unless it's the only qubit in the quantum computer at which point the whole question of addressed instructions becomes moot, anyway.

So $U(\mathbf{s})\,|\Psi\rangle$ is what we actually mean, by "application of operator U to quantum register $\mathbf{s}$". Since there are $\frac{n!}{(n-m)!}$ possible m-qubit registers on an n-qubit machine, a given m-qubit operator U can describe $\frac{n!}{(n-m)!}$ different transformations $U(\mathbf{s})$.

In analogy to boolean networks, unitary operators which can be applied to arbitrary sets of qubits are also referred to as *quantum gates*.

2.2.2.3 Universal Quantum Gates

A well known result from classical boolean logic, is that any possible function $f : \mathbf{B}^n \to \mathbf{B}^m$ can be constructed as a composition from a small universal set of operators if we can "wire" the inputs and outputs to arbitrary bits in a feed-forward network. Examples for universal sets of logical gates are $\{\vee, \neg\}$, $\{\rightarrow, \neg\}$ or $\{\bar{\wedge}\}$.

As mentioned in 1.3.2.6, unitary operations can be described as abstract "rotations" in the Hilbert space. A complex rotation of a single qubit has the general form

$$U2(\omega, \alpha, \beta, \phi) = \mathrm{e}^{-\mathrm{i}\phi} \begin{pmatrix} \mathrm{e}^{\mathrm{i}\alpha} \cos\omega & -\mathrm{e}^{-\mathrm{i}\phi} \sin\omega \\ \mathrm{e}^{\mathrm{i}\phi} \sin\omega & \mathrm{e}^{-\mathrm{i}\alpha} \cos\omega \end{pmatrix} \tag{2.33}$$

If this operator could be applied to arbitrary 2-dimensional subspaces $\mathcal{H}' = \mathbf{C}^2$ of $\mathcal{H} = \mathcal{H}' \times \mathcal{H}''$, then any unitary transformation could be constructed by composition in at most $\binom{\dim \mathcal{H}}{2}$ steps [14], very much like a general rotation in $\mathbf{R}^n$ can be decomposed into $\binom{n}{2}$ simple rotations in the coordinate planes.

In our definition of quantum gates, however, we are restricted to subspaces corresponding to quantum registers (see 2.2.1.5), so in the case of an n-qubit quantum computer ($\dim \mathcal{H} = 2^n$), this leaves us with merely n possible 1-qubit subspaces $\mathcal{H}'$ and the corresponding sets of register operators $U2^{(i)}(\omega, \alpha, \beta, \phi)$. Since $[U2^{(i)}, U2^{(j)}] = 0$, any composition U of $U2$ gates, would result in a transformation of the form

$$U|d_0, d_1, \ldots d_{n-1}\rangle = (U_1|d_0\rangle)(U_2|d_2\rangle)\ldots(U_{n-1}|d_{n-1}\rangle) \tag{2.34}$$

So just as the NOT gate itself is not universal for boolean logic, to construct a universal set of quantum gates, we require an additional 2-qubit operation, to create entangled multi-qubit states.

One possibility for a nontrivial 2-qubit operator is XOR which is defined as $XOR : |x, y\rangle \to |x, x \oplus y\rangle$ or in matrix notation:

$$XOR = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.35}$$

Deutsch [6] has shown that the set $\{U2(\omega, \alpha, \beta, \phi), XOR\}$ is in fact universal for unitary transformation. Furthermore, since $\{U2(\omega', \alpha', \beta', \phi')^n\}$ is dense in $\{U2(\omega, \alpha, \beta, \phi)\}$ for almost any[6] set of parameters, $\{U2, XOR\}$ is universal for most $U2$ in the sense that any unitary transformation U can be approximated to arbitrary precision.

Deutsch also proposed a 3-qubit gate $D(\theta)$ which is universal, while only requiring one parameter:

$$D(\theta) : |i, j, k\rangle \rightarrow \begin{cases} i\cos\theta\,|i, j, k\rangle + \sin\theta\,|i, j, 1-k\rangle & \text{for } i = j = 1 \\ |i, j, k\rangle & \text{otherwise} \end{cases} \tag{2.36}$$

2.2.2.4 Pseudo-classical Operators

The general form of a unitary operator U over n qubits is

$$U = \sum_{i=0}^{2^n-1} \sum_{j=0}^{2^n-1} |i\rangle\, u_{ij}\, \langle j| \quad \text{with} \quad \sum_{k=0}^{2^n-1} u_{ki}^* u_{kj} = \delta_{ij} \tag{2.37}$$

If the matrix elements u_{ij} are of the form $u_{ij} = \delta_{i\pi_j}$ with some permutation π, then their effect on basis states can be described in terms of classical reversible boolean logic.

Definition 5 (Pseudo-classical Operator) *A n-qubit pseudo-classical operator is a unitary operator of the form $U : |i\rangle \rightarrow |\pi_i\rangle$ with some permutation π over $\mathbf{Z}^{2^n}$.*

For $\theta = \pi/2$ the universal Deutsch gate $D(\theta)$ (2.36) degenerates into the pseudo-classical operator

$$T = D(\frac{\pi}{2}) = |i, j, (i \wedge j) \oplus k\rangle\langle i, j, k| \quad \text{with} \quad i, j, k \in \mathbf{B} \tag{2.38}$$

T is the 3-bit controlled-not or *Toffoli gate*, which is a universal gate for reversible boolean logic.

Let $f : \mathbf{Z}_{2^n} \rightarrow \mathbf{Z}_{2^n}$ be a bijective function, then the corresponding pseudo-classical operator F is given as

$$F = \sum_{i=0}^{2^n-1} |f(i)\rangle\langle i| \quad \text{and} \quad F^{-1} = F^\dagger = \sum_{i=0}^{2^n-1} |i\rangle\langle f(i)| \tag{2.39}$$

[6] basically, it is just required that the quotients between $\omega', \alpha', \beta', \phi'$ and π are irrational.

2.2.2.5 Quantum Functions

One obvious problem of quantum computing is its restriction to reversible computations. Consider a simple arithmetical operation like integer division by 2 ($\mathit{DIV2}'|i\rangle = |i/2\rangle$ for even i and $|(i-1)/2\rangle$ for odd i). Clearly, this operation is non-reversible since $\mathit{DIV2}'|0\rangle = \mathit{DIV2}'|1\rangle$, so no corresponding pseudo-classical operator exists.

However, if we use a second register with the initial value $|0\rangle$, then we can define an operator *DIV2* which matches the condition $\mathit{DIV2}|x,0\rangle = |x,x/2\rangle$ or $|x,(x-1)/2\rangle$ respectively. The behavior of $\mathit{DIV2}|x,y\neq 0\rangle$ is undefined and can be set arbitrarily as long as *DIV2* remains pseudo-classical.[7].

Definition 6 (Quantum Functions) *For any function $f : \mathbf{B}^n \to \mathbf{B}^m$ (or equivalently $f : \mathbf{Z}_{2^n} \to \mathbf{Z}_{2^m}$) there exists a class of pseudo-classical operators $F : \mathbf{C}^{2^{n+m}} \to \mathbf{C}^{2^{n+m}}$ working on an n-qubits input and an m-qubits output register with $F|x,0\rangle = |x,f(x)\rangle$. Operators of that kind are referred to as quantum functions.*

For any boolean function $f : \mathbf{B}^n \to \mathbf{B}^m$ there exist $(2^{n+m}-2^n)!$ different quantum functions F.

2.2.2.6 Conditional Operators

Classical programs allow the conditional execution of code in dependence on the content of a boolean variable (conditional branching).

A unitary operator, on the other hand, is static and has no internal flow-control. Nevertheless, we can conditionally apply an n qubit operator U to a quantum register by using an *enable* qubit and define an $n+1$ qubit operator U'

$$U' = \begin{pmatrix} I(n) & 0 \\ 0 & U \end{pmatrix} \tag{2.40}$$

So U is only applied to base-vectors where the enable bit is set. This can be easily extended to enable-registers of arbitrary length.

Definition 7 (Conditional Operator) *A conditional operator $U_{[[\mathbf{e}]]}$ with the enable register $\mathbf{e}$ is a unitary operator of the form*

$$U_{[[\mathbf{e}]]} : |i,\epsilon\rangle = |i\rangle|\epsilon\rangle_{\mathbf{e}} \to \begin{cases} (U\,|i\rangle)\,|\epsilon\rangle_{\mathbf{e}} & \mathit{if}\ \epsilon = 111\ldots \\ |i\rangle|\epsilon\rangle_{\mathbf{e}} & \mathit{otherwise} \end{cases} \tag{2.41}$$

[7] In this special case, just one additional qubit to hold the lowest bit of the argument would suffice to extend $\mathit{DIV2}'$ to a unitary operator.

Conditional operators a frequently used in arithmetic quantum functions and other pseudo-classical operators.

If the architecture allows the efficient implementation of the *controlled-not* gate $C : |x, y_1, y_2 \ldots\rangle \to |(x \oplus \bigwedge_i y_i), y_1, y_2 \ldots\rangle$, then conditional pseudo-classical operators can be realized by simply adding the enable string to the control register of all controlled-not operations.

2.2.3 Input and Output

2.2.3.1 Quantum Computing and Information Processing

In 2.1.3 we have shown that the interpretation of computing as a physical process, rather than the abstract manipulation of symbols, leads to an extended notion of computability. We have also identified the the concept of unitary transformations as an adequate paradigm for "physical computability".

Unitary transformations describe the transition between machine states and thereby the temporal evolution of a quantum system. The very notion of a (quantum) computer as a "computing machine" requires, however, that the evolution of the physical system corresponds to a processing of information.

Classical information theory requires that any "reasonable" information can be expressed as a series of answers to yes-no questions, i.e. a string of bits. But unlike classical symbolic computation, where every single step of a computation can be mapped onto a bit-string, physical computation requires such a labeling only for the initial and the final machine state (see 2.1.3.2), the labels of which make up the input and output of the computation.

This requirement is in full accordance with the Copenhagen interpretation of quantum physics, which states that the setup and the outcome of any experiment has to be described in classical terms.

2.2.3.2 Labeling of States

As the machine state Ψ is not directly accessible, any physically realizable labeling has to correspond to an observable $\mathcal{O}$. As has been shown in 1.3.2.2, in quantum physics, an observable $\mathcal{O}$ is expressed by a Hermitian operator O.

A natural choice for $\mathcal{O}$ on an n-qubit quantum computer would be the classical values $\mathcal{N} = (\mathcal{N}_0, \mathcal{N}_1, \ldots \mathcal{N}_{n-1})$ of the singe qubits with the Hermitian operators

$$N = (N_0, N_1, \ldots N_{n_1}) = N_0 + 2N_1 + \ldots + 2^{n-1} N_{n-1} \tag{2.42}$$

$$N_i \,|d_0 \ldots d_{n-1}\rangle = d_i \,|d_0 \ldots d_{n-1}\rangle$$

As $\mathcal{N}$ is only defined for eigenstates of N (see 1.3.2.3), the labeling $m : \mathcal{H} \to \mathbf{B}^n$ is only defined for states $\Psi \in \mathcal{H}$ of the form

$$|\Psi\rangle = \mathrm{e}^{\mathrm{i}\phi}|d_0 \dots d_{n-1}\rangle \tag{2.43}$$

2.2.3.3 Initialization

To set a quantum computer to a desired initial state $|\Psi_0\rangle = |s_0\rangle$ corresponding to the boolean input string s_0, it suffices to provide means to initially "cool" all qubits to $|0\rangle$ and then apply any unitary transformation U which matches the condition $U|0\rangle = |s_0\rangle$.

Definition 8 *The reset operator R is a constant operator over $\mathcal{H}$ and defined as $R|\Psi\rangle = |0\rangle$.*

2.2.3.4 Measurement

As has been described in 1.3.2.3, it is impossible to observe a quantum state ψ without, at the same time, forcing the system to adopt a state ψ' which is an eigenstate of the Hermitian operator O corresponding to the observed quantity $\mathcal{O}$. The transition probability is thereby given as

$$p_{\psi \to \psi'} = |\langle \psi' | \psi \rangle|^2 \tag{2.44}$$

If we measure the binary values $\mathcal{N}$ of an n-qubit quantum computer in the state

$$|\Psi\rangle = \sum_{i=0}^{2^n-1} c_i |i\rangle \tag{2.45}$$

the probabilities to measure i and the assorted post measurement states are consequently

$$p_i = |c_i|^2 \quad \text{and} \quad |\psi_i'\rangle = |i\rangle \tag{2.46}$$

2.2.3.5 Partial Measurement

Measurements don't have to cover the whole machine state, but can also be restricted to single qubits or quantum registers.

Consider two quantum registers with n and m qubits in the state

$$|\psi\rangle = \sum_{i=0}^{2^n-1} \sum_{j=0}^{2^m-1} c_{i,j} |i, j\rangle \quad \text{with} \quad \sum_{i,j} c_{i,j}^* c_{i,j} = 1 \tag{2.47}$$

The probability p_i to measure the number i in the first register and the according post measurement state $|\psi_i'\rangle$ are given by

$$p_i = \sum_{j=0}^{2^m-1} c_{i,j}^* c_{i,j}, \quad \text{and} \quad |\psi_i'\rangle = \frac{1}{\sqrt{p_i}} \sum_{j=0}^{2^m-1} c_{i,j} |i,j\rangle \tag{2.48}$$

The measurement of qubits is the only non unitary operation, a quantum computer must be able to perform during calculation.

2.3 Models of Quantum Computation

In classical information theory, the concept of the universal computer can be represented by several equivalent models, corresponding to different scientific approaches. From a mathematical point of view, a universal computer is a machine capable of calculating *partial recursive functions*, computer scientists often use the *Turing machine* as their favorite model, an electro-engineer would possibly speak of *logic circuits* while a programmer certainly will prefer a *universal programming language*.

As for quantum computation, each of these classical concepts has a quantum counterpart: [25]

Model	**classical**	**quantum**
Mathematical	partial recursive funct.	unitary operators
Machine	Turing Machine	QTM
Circuit	logical circuit	quantum gates
Algorithmic	univ. programming language	QPLs

Table 2.1: classical and quantum computational models

2.3.1 The Mathematical Model of QC

The paradigm of computation as a physical process requires that QC can — in principle — be described by the same means as any other physical reality, which, for the field of quantum physics, is the mathematical formalism of Hilbert space operator algebra. The basics of this formalism, as far as they are relevant to QC, have been the topic of 1.3 and chapter 2.

The moral equivalent in QC to partial recursive functions, the mathematical concept of classical computability, are unitary operators. As every classically computable problem can be reformulated as calculating the value of a partial recursive function, each quantum computation must have a corresponding unitary operator.

The mathematical description of an operator is inherently declarative; the actual implementation for a certain quantum architecture i.e. the algorithmic decomposition into elementary operations, is beyond the scope of this formalism. Also, since the mathematical model treats unitary operators as black boxes, no complexity measure is provided.

2.3.2 Quantum Turing Machines

In analogy to the classic Turing Machine (TM) several propositions of Quantum Turing Machines (QTM), as a model of a universal quantum computer have been made [3, 1].

The complete machine-state $|\Psi\rangle$ is thereby given by a superposition of base-states $|l, j, s\rangle$, where l is the inner state of the head, j the head position and s the binary representation of the tape-content. To keep $\mathcal{H}$ separable, the (infinite) bit-string s has to meet the *zero tail state* condition i.e. only a finite number of bits with $s_m \neq 0$ are allowed.

The quantum analogon to the transition function of a classic probabilistic TM is the *step operator* T, which has to be unitary to allow for the existence of a corresponding Hamiltonian (see 1.3.2.5) and meet locality conditions for the effected tape-qubit, as well as for head movement.

QTMs provide a measure for execution times, but — as with the classical TM — finding an appropriate step operator can be very hard and runtime-complexity (i.e. the number of applications of T in relation to the problem size) remains an issue. Outside quantum complexity theory, QTMs are of minor importance.

2.3.3 Quantum Circuits

Quantum circuits are the QC equivalent to classical boolean feed-forward networks, with one major difference: since all quantum computations have to be unitary, all quantum circuits can be evaluated in both directions (as with classical reversible logic). Quantum circuits are composed of elementary gates and operate on qubits, thus $\dim(\mathcal{H}) = 2^n$ where n is the (fixed) number of qubits. The "wiring" between the gates thereby corresponds to unitary reordering operators $\Pi_{\mathbf{s}}$ (see 2.2.1.5).

In comparison with classical boolean feed-forward networks, this imposes the following restrictions:

- Only n-to-n networks are allowed i.e. the total number of inputs has to match the total number of outputs.
- Only n-to-n gates are allowed.
- No forking of inputs is allowed. This is directly related to the fact that qubits can't be copied, i.e. that there exists no unitary operation

$$\text{Copy}\, |\psi\rangle|0\rangle \to |\psi\rangle|\psi\rangle \quad \text{with} \quad |\psi\rangle \in \mathbf{C}^2 \tag{2.49}$$

  which can turn a general qubit-state into a product state of itself.
- No "dead ends" are allowed. Again, this is because the erasure of a qubit

$$\text{Erase}\, |\psi\rangle \to |0\rangle \quad \text{with} \quad |\psi\rangle \in \mathbf{C}^2 \tag{2.50}$$

  is not a unitary operation.

To allow for implementation of all possible unitary transformations, a universal set of elementary gates must be available, out of which composed gates can be constructed (see 2.2.2.3). Each m-qubit gate U thereby describes up to $\frac{n!}{(n-m)!}$ different unitary transformations $U(\mathbf{s})$, depending on the wiring of the inputs (see 2.2.2.2).

As opposed to the operator formalism, the gate-notation is an inherently constructive method and — other than QTMs — the complexity of the problem is directly reflected in the number of gates necessary to implement it.

2.3.4 Quantum Programming Languages

When it comes to programming and the design of non-classic algorithms, we can look at the mathematical description as the specification and quantum circuits as the assembly language of QC.

Just as classical programming languages, quantum programming languages (QPLs) provide a constructive means to specify the sequence of elementary operators, while allowing nested levels of abstraction.

2.3.4.1 Flow Control

In it's simplest form, a quantum algorithm merely consists of a unitary transformation and a subsequent measurement of the resulting state. This would e.g. be the case, if a quantum computer is used to emulate the behavior of another quantum system.

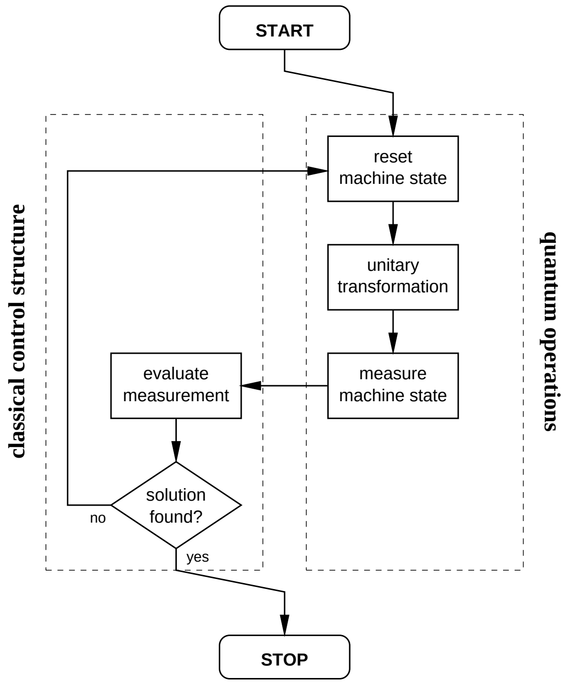

Figure 2.1: A simple non-classical algorithm

For more "traditional" computational tasks, as e.g. searching or mathematical calculations, efficient quantum implementations often have the form of probabilistic algorithms. Figure 2.1 shows the basic outline of a probabilistic non-classical algorithm with a simple evaluation loop.

More complex quantum algorithms, as e.g. Shor's algorithm for quantum factoring (see 4.2), can also include classical random numbers, partial measurements, nested evaluation loops and multiple termination conditions: The actual quantum operations as resetting of the machine state, unitary transformations and measurements are embedded into a classical flow-control framework.

A formal way to describe the classical control structure, is to consider quantum operations as special statements within a classical procedural language. Therefor any QPL also has to be a universal programming language.

2.3.4.2 Operator Specification

Classical procedural languages provide different levels of abstraction by allowing the grouping of primitives into reusable subroutines (procedures) which can operate on different data (parameters, references) and use temporally allocated memory (local variables).

If this concept is to be used for the definition of unitary operators, then language elements have to be provided which account for the reversibility of unitary transformation and the non-local nature of entangled quantum bits.

- **Mathematical Semantics:** The effect of an operator has to be *uniform* and has to be restricted to the quantum machine state i.e. the use of an operator must not interfere with the classical state of the machine.

  This means that the implementation of an operator must only depend on its parameters and must not produce any side-effects. This esp. excludes the use of global variables and the use of non-deterministic functions (such as a random numbers).

- **Unitarity:** It has to be assured that operators are restricted to unitary transformation. This excludes non-unitary quantum operations such as measurement.

- **Reversibility:** Since for any unitary operator, there exists an inverse adjoint operator, a QPL should provide means to execute operators in reverse.

- **Symbolic Registers:** An operator must be able to operate on any set of qubits. This requires the ability to define symbolic quantum registers.

Chapter 3

Quantum Programming

This chapter discusses the programming of quantum computers and the design of quantum algorithms in the experimental quantum programming language QCL.

3.1 Introduction

3.1.1 Computers and Programming

As has been illustrated in 2.1.2, a computer is basically a device which

1. holds a physical *machine state* S
2. is capable of performing a set of well defined instructions $\mathbf{I}$ to transform between machine states
3. provides means to initialize and measure the machine state while interpreting S as discrete symbolic *computational states* s

The sequence of instructions $\pi = \langle I_1, I_2, \dots I_n \rangle$ to transform the initial state S into the final state S_n is called a *program*.

The way π is actually specified, depends on the computational model; possibilities vary from explicit enumeration, over feed forward networks (as in logical circuits) and decision trees up to finite automatons (as in the Turing machine).

A general requirement of any specification method is, that the mechanism used to produce π must not be more powerful or complex than the machine it is executed on, which would defy the purpose of using a computer in the first place.

3.1.2 Complexity Requirements

As has been pointed out in 2.3.4, QPLs use a classical universal programming language to define the actual sequence π of instructions for a quantum computer. According to the above criterion, this approach is useful, only if quantum computers are at least as powerful as universal classical computers.

If we consider a quantum computer with the Toffoli gate (see 2.2.2.4) as the only available instruction, then any transformation of the machine state has to be of the form

$$|\Psi\rangle = |i\rangle \longrightarrow |g(i)\rangle = |\Psi\rangle \quad \text{with} \quad g : \mathbf{B}^n \to \mathbf{B}^n \tag{3.1}$$

Since the Toffoli gate is universal for reversible boolean logic, any bijective boolean function g can directly be implemented on a quantum computer.

A general boolean function f over $\mathbf{B}^n$, can be implemented by using a pseudo-classical operator F

$$F\,|i,0\rangle = |i, f(i)\rangle \quad \text{with} \quad F^\dagger F = I \tag{3.2}$$

So any classically computable function f can also be implemented on a quantum computer. Moreover, C. H. Bennet has shown that a reversible implementation of f can be done with a maximal overhead of $O(2)$ in time and $O(\sqrt{n})$ in space complexity (see 3.5.2). [8]

On the other hand, as a general n-qubit quantum state consists of maximally 2^n eigenstates with a non-zero amplitude and unitary transformations take the form of linear operators and consequently can be described as

$$U : \, |i\rangle \to \sum_{j=0}^{2^n-1} u_{ij}|j\rangle \quad \text{with} \quad i, j \in \mathbf{Z}^{2^n}, \tag{3.3}$$

a classical computer can simulate any unitary operator with arbitrary precision by encoding the complex amplitudes as fixed point binary numbers. In the general case, however, this will require an exponential overhead in time as well as in space complexity.

Due to the stochastic nature of quantum measurements, the emulating computer will also need a source of true randomness (like e.g. the probabilistic Turing machine).

3.1.3 Hybrid Architecture

So QPLs can be regarded as a meta-programming languages, as a program isn't intended to run on a quantum computer itself, but on a (probabilistic) classical computer which in turn controls a quantum computer. In the

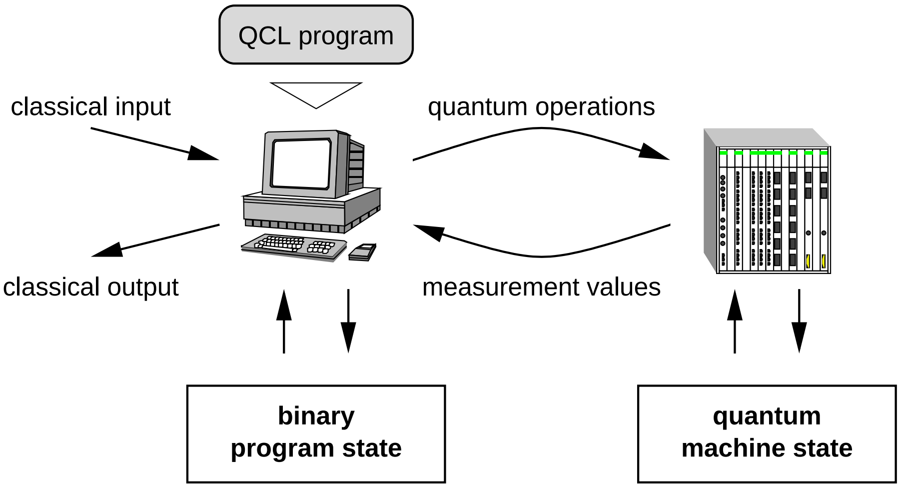

Figure 3.1: The hybrid architecture of QCL

terms of classical computer science, you can describe this setting as a universal computer with a quantum oracle. Figure 3.1 illustrates this hybrid architecture.

From the perspective of the user, quantum programs behave exactly like any other classical program, in the sense that it takes classical input such as startup parameters or interactive data, and produces classical output. The state of the controlling computer (i.e. program counter, variable values, but also the mapping of quantum registers) is also strictly classical and referred to as *program state*.

The actual program π, i.e. the sequence of quantum instructions consisting of elementary gates, measurement- and initialization-instructions is passed over a well defined interface to the quantum computer, while the returned output of is restricted to binary measurements values. The quantum computer doesn't require any control logic, it's computational state can therefor be fully described by the common quantum state Ψ of its qubits, also referred to as *machine state*.

3.2 QCL as a Classical Language

Since the computational model of QPLs is that of a classical computer with a quantum oracle, QCL contains all features of a classical universal programming language, such as variables, loops, subroutines and conditional branching.

3.2.1 Structure of a QCL Program

The syntactic structure of a QCL program is described by a context free LALR(1) grammar (see appendix A) with *statements* and *definitions* as top symbols:

$$\textit{qcl-input} \leftarrow \{\, \textit{stmt} \,|\, \textit{def} \}$$

3.2.1.1 Statements

Statements range from simple commands, over procedure-calls to complex control-structures and are executed when they are encountered.

```
qcl> if random()>=0.5 { print "red"; } else { print "black"; }
: red
```

3.2.1.2 Definitions

Definitions are not executed but bind a value (variable- or constant-definition) or a block of code (routine-definition) to a *symbol* (identifier).

```
qcl> int counter=5;
qcl> int fac(int n) { if n<=0 {return 1;} else {return n*fac(n-1);} }
```

Consequently, each symbol has an associated *type*, which can either be a *data type* or a *routine type* and defines whether the symbol is accessed by reference or call.

3.2.1.3 Expressions

Many statements and routines take arguments of certain data types. These *expressions* can be composed of *literals*, variable references and sub-expressions combined by operators and function calls.

```
qcl> print "5 out of 10:",fac(10)/fac(5)^2,"combinations."
: 5 out of 10: 252 combinations.
```

3.2.2 Data Types and Variables

The classic data-types of QCL are the arithmetic types `int`, `real` and `complex` and the general types `boolean` and `string`.

Type	Description	Examples
`int`	integer	`1234, -1`
`real`	real number	`3.14, -0.001`
`complex`	complex number	`(0,-1), (0.5, 0.866)`
`boolean`	logic value	`true, false`
`string`	character string	`"hello world", ""`

Table 3.1: classic types and literals

3.2.2.1 Constants

Frequently used values can be defined as symbolic constants. The syntax of a constant declaration is

const-def ← `const` *identifier* `=` *expr* `;`

The definition of `pi` in the standard include file is e.g.

```
const pi=3.141592653589793238462643383279502884197;
```

3.2.2.2 Variables

The definition of variables in QCL is analogous to C:

var-def ← *type* *identifier* [`=` *expr*] `;`

If no initial value is given, the new variable is initialized with zero, `false` or `""`, respectively. The value of a variable can be changed by an *assignment*, user input (see 3.2.4.3) and quantum measurement (see 3.4.1):

```
qcl> complex z;          // declare complex variable z
qcl> print z;            // z was initialized with 0
: (0.000000,0.000000)
qcl> z=(0,1);            // setting z to i
qcl> print z;
: (0.000000,1.000000)
qcl> z=exp(z*pi);        // assignment to z may contain z
qcl> print z;
: (-1.000000,0.000000)
qcl> input z;            // ask for user input
? complex z [(Re,Im)] ? (0.8,0.6)
qcl> print z;
: (0.800000,0.600000)
```

3.2.3 Expressions

3.2.3.1 Operators

Table 3.2 shows all QCL operators ordered from high to low precedence.[1] All binary operators are left associative, thus $a \circ b \circ c = (a \circ b) \circ c$. Explicit grouping can be achieved by using parentheses.

Op	Description	Argument type
`#`	register size	quantum types
`^`	power integer power	all arithmetic `int`
`-`	unary minus	all arithmetic
`*` `/`	multiplication division integer division	all arithmetic all arithmetic `int`
`mod`	integer modulus	`int`
`+` `-`	addition subtraction	all arithmetic all arithmetic
`&`	concatenation	`string`, quantum types

`==`
`!=`
`<`
`<=`
`>`
`>=`
	equal
	unequal
	less
	less or equal
	greater
	greater or equal
		all arithmetic, `string`
		all arithmetic, `string`
		`integer`, `real`
		`int`, `real`
		`int`, `real`
		`int`, `real`

`not`	logic not	`boolean`
`and`	logic and	`boolean`
`or` `xor`	logic inclusive or logic exclusive or	`boolean` `boolean`

Table 3.2: QCL operators

Arithmetic operators generally work on all arithmetic data types and return the most general type (operator overloading), e.g.

[1] For the sake of completeness, table 3.2 also includes the operators `#` and `&`, which take quantum registers as arguments, see 3.4.3.1 and 3.3.3.2

```
qcl> print 2+2;          // evaluates to int
: 4
qcl> print 2+2.0;        // evaluates to real
: 4.000000
qcl> print 2+(2,0);      // evaluates to complex
: (4.000000,0.000000)
```

To allow for clean integer arithmetic there are two exceptions to avoid type-casts:

- The division operator `/` does integer division if both arguments are integer.
- The power operator `^` for integer bases is only defined for non-negative, integer exponents. For real exponents, the base must be non-negative.

3.2.3.2 Functions

QCL expressions may also contain calls to built-in or user defined functions. Table 3.3 shown all built-in unary arithmetic functions.

Trigonometric Funct.		Hyperbolic Funct.	
`sin(x)`	sine of x	`sinh(x)`	hyperbolic sine of x
`cos(x)`	cosine of x	`cosh(x)`	hyperbolic cosine of x
`tan(x)`	tangent of x	`tanh(x)`	hyperbolic tangent of x
`cot(x)`	cotangent of x	`coth(x)`	hyperbolic cotangent of x
Complex Funct.		**Exponential an related Funct.**	
`Re(z)`	real part of z	`exp(x)`	e raised to the power of x
`Im(z)`	imaginary part of z	`log(x)`	natural logarithm of x
`abs(z)`	magnitude of z	`log(x,n)`	base-n logarithm of x
`conj(z)`	conjugate of z	`sqrt(x)`	square root of x

Table 3.3: QCL arithmetic functions

In addition to the above, QCL also contains n-ary functions such as minimum or gcd, conversion functions and the the pseudo function `random()` (table 3.4). As the latter is no function in the mathematical sense, it may not be used within the definition of user-functions and quantum operators.

3.2.4 Simple Statements

3.2.4.1 Assignment

The value of any classic variable can be set by the assignment operator `=`. The right-hand value must be of the same type as the variable. In con-

Funct.	Description
`ceil(x)`	nearest integer to x (rounded upwards)
`floor(x)`	nearest integer to x (rounded downward)
`max(x,...)`	maximum
`min(x,...)`	minimum
`gcd(n,...)`	greatest common divisor
`lcm(n,...)`	least common multiple
`random()`	random value from $[0, 1)$

Table 3.4: other QCL functions

trast to arithmetic operators and built-in functions, no implicit typecasting is performed.

```
qcl> complex z;
qcl> z=pi;          // no typecast
! type mismatch: invalid assignment
qcl> z=conj(pi);    // implicit typecast
```

3.2.4.2 Call

The call of a `procedure` has the syntax

$$\mathit{stmt} \leftarrow \mathit{identifier}\ \texttt{(}\ [\,\mathit{expr}\ \{\ \texttt{,}\ \mathit{expr}\,\}]\ \texttt{)}\ \texttt{;}$$

As with assignments, no typecasting is performed for classical argument types.

Due to the potential side-effects on the program state, procedure-call may not occur within the definition of functions or operators.

3.2.4.3 Input

The `input` command prompts for user input and assigns the value to the variable *identifier*. Optionally a prompt string *expr* can be given instead of the standard prompt which indicates the type and the name of the variable.

```
qcl> real n;
qcl> input "Enter Number of iterations:",n;
? Enter Number of iterations: 1000
```

3.2.4.4 Output

The `print` command takes a comma separated list of expressions and prints them to the console. Each output is prepended by a colon and terminated with newline.

```
qcl> int i=3; real x=pi; complex z=(0,1); boolean b;
qcl> print i,x,z,b;
: 3 3.141593 (0.000000,1.000000) false
```

3.2.5 Flow Control

3.2.5.1 Blocks

All flow control statements operate on blocks of code. A block is a list of statements enclosed in braces:

$$\textit{block} \leftarrow \texttt{\{}\ \textit{stmt}\ \{\ \textit{stmt}\ \}\ \texttt{\}}$$

Blocks may only contain executable statements, no definitions. Unlike C, a block is not a compound statement and always part of a control structure. To avoid ambiguities with nesting, the braces are obligatory, even for single commands.

3.2.5.2 Conditional Branching

The `if` and `if-else` statements allow for the conditional execution of blocks, depending on the value of a boolean expression.

$$\textit{stmt} \leftarrow \texttt{if}\ \textit{expr}\ \textit{block}\ [\,\texttt{else}\ \textit{block}\,]$$

If *expr* evaluates to `true`, the `if`-block is executed. If *expr* evaluates to `false`, the `else`-block is executed if defined.

3.2.5.3 Counting Loops

`for`-loops take a counter *identifier* of type integer which is incremented from $expr_{from}$ to $expr_{to}$. The loop body is executed for each value of *identifier*.

$$\textit{stmt} \leftarrow \texttt{for}\ \textit{identifier}\ \texttt{=}\ \textit{expr}_{\textit{from}}\ \texttt{to}\ \textit{expr}_{\textit{to}}\ [\,\texttt{step}\ \textit{expr}_{\textit{step}}\,]\ \textit{block}$$

Inside the body, the counter is treated as a constant.

```
qcl> int i;
qcl> for i=10 to 2 step -2 {  print i^2; }
: 100
: 64
: 36
: 16
: 4
qcl> for i=1 to 10 {  i=i^2; }          // i is constant in body
! unknown symbol: Unknown variable i
```

When the loop is finished, *identifier* is set to $expr_{to}$.

3.2.5.4 Conditional Loops

QCL supports two types of conditional loops:

$$\begin{aligned} \mathit{stmt} &\leftarrow \texttt{while}\ \mathit{expr}\ \mathit{block} \\ &\leftarrow \mathit{block}\ \texttt{until}\ \mathit{expr}\ ; \end{aligned}$$

A `while`-loop is iterated as long as a the condition *expr* is satisfied. When *expr* evaluates to `false`, the loop terminates. An `until`-loop is executed at least once and iterated until the condition *expr* is satisfied.

3.2.6 Classical Subroutines

3.2.6.1 Functions

User defined functions may be of any classical type and may take an arbitrary number of classical parameters. The value of the function is passed to the invoking expression by the `return` statement. Local variables can be defined at the top of the function body.

```
int Fibonacci(int n) {        // calculate the n-th
  int i;                      // Fibonacci number
  int f;                      // by iteration
  for i = 1 to n {
    f = 2*f+i;
  }
  return f;
}
```

QCL requires functions to have mathematical semantics, so their value has to be deterministic and their execution must not have any side-effects on the program state.

```
qcl> int randint(int n) { return floor(n*random()); }
! in function randint: illegal scope: function random is not allowed
in this scope
qcl> int foo=4711;
qcl> int bar(int n) { foo=foo+n; return foo; }
! in function bar: unknown symbol: Unknown local variable foo
```

Functions can call other functions within their body. Recursive calls are also allowed.

```
int fac(int n) {              // calculate n!
  if n<2 {                    // by recursion
    return 1;
  } else {
    return n*fac(n-1);
  }
}
```

3.2.6.2 Procedures

Procedures are the most general routine type and used to implement the classical control structures of quantum algorithms which generally involve evaluation loops, the choice of applied operators, the interpretation of measurements and classical probabilistic elements.

With the exception of routine declarations, procedures allow the same operations as are available in global scope (e.g. at the shell prompt) allowing arbitrary changes to both the program and the machine state. Operations exclusive to procedures are

- Access to global variables
- (Pseudo) Random numbers by using the pseudo-function `random()`
- Non-unitary operations on the machine state by using the `reset` and `measure` commands (see 3.4.1)
- User input by using the `input` command (see 3.2.4.3)

Procedures can take any number of classical or quantum arguments and may call all types of subroutines.

3.3 Quantum States and Variables

3.3.1 Quantum Memory Management

3.3.1.1 Machine State and Program State

The memory of a quantum computer is usually a combination of 2-state subsystems, referred to as quantum bits (qubits). As pointed out in 2.2.1.3 the "memory content" is the combined state $|\Psi\rangle$ of all qubits. This state is referred to as the (quantum) *machine state* as opposed to the *program state* which is the current state of the controlling (classic) algorithm (e.g. contents of variable, execution stack, etc.) described by the QCL program.

The machine state $|\Psi\rangle$ of an n qubit quantum computer is a vector in the Hilbert space $\mathcal{H} = \mathbf{C}^{2^n}$, however — due to the destructive nature of measurement (see 1.3.2.3) — $|\Psi\rangle$ cannot be directly observed and consequently isn't accessible from within QCL.

Due to the current lack of real-live quantum computers, the interpreter `qcl` contains the emulation library `libqc` which can simulate a quantum computer with an arbitrary number of qubits. It also provides an interface to access the simulated machine state via the `load`, `save` and `dump` commands (see 3.3.1.6). These commands, however don't interfere with the program state.

3.3.1.2 Quantum Registers

QCL uses the concept of quantum registers (see 2.2.1.5) as an interface between the machine state and the controlling classical computer. A quantum register is a pointer to a sequence of (mutually different) qubits and thus, while referring to a quantum subsystem, is still a classical variable.

All operations on the machine state (except for the `reset` command, see 3.4.1) take quantum registers as operands. Since an n qubit quantum computer allows for $\frac{n!}{(n-m)!}$ different m qubit registers $\mathbf{s} \in \mathbf{Z}_n^m$, any unitary or measurement operation on a m qubit register, can result in $\frac{n!}{(n-m)!}$ different operations on the machine state: This requires that all elementary unitary operations of the quantum computer to be applicable to arbitrary qubits and requires the physical architecture to allow the measurement of single qubits.[2]

3.3.1.3 The Quantum Heap

In QCL, the relation between registers and qubits is handled transparently by allocation and deallocation of qubits from the *quantum heap*, which allows the use of local quantum variables. All free (i.e. unallocated) quantum memory has to be *empty*.

Definition 9 (Empty Registers) *A quantum register* $\mathbf{s}$ *is empty iff*

$$P_0(\mathbf{s})\,|\Psi\rangle = |\Psi\rangle \quad \text{with} \quad P_0 = |0\rangle\langle 0| \tag{3.4}$$

At startup or after the `reset` command, the whole machine state is empty, thus $|\Psi\rangle = |0\rangle$.

The machine state of an n-qubit quantum computer with m allocated qubits therefor is a product state of the form

$$|\Psi\rangle = |\psi\rangle_{\mathbf{s}}|0\rangle_{\mathbf{s}_\perp} \quad \text{with} \quad \mathbf{s} \in \mathbf{Z}_n^m \quad \text{and} \quad \mathbf{s}_\perp \in \mathbf{Z}_n^{n-m} \tag{3.5}$$

As has been pointed out in 1.3.3.2, two quantum systems whose common wave function is a product state are physically independent. This esp. means that neither measurements nor unitary transformations on the allocated bits $\mathbf{s}$ will affect $\mathbf{s}_\perp$ being in substate $|0\rangle$.

The concept of the quantum heap allows two important abstractions:

[2]Since the operators N_i for the value of the qubits commute (i.e $[N_i, n_j] = 0$), the number of physically different measurement operations is merely $\binom{n}{m}$ as the additional bit-permutations are in fact classical operations.

- Since the allocation of registers is transparent, no qubit positions need to be specified.
- Since allocated and unallocated qubits are in a product state, the definition of quantum algorithms is independent from the total number of qubits.

3.3.1.4 Register allocation

Quantum registers are allocated, when a quantum variable is defined. The qubit positions for each register can be inspected using the `print` statement.

```
$ qcl -b10  # start qcl-interpreter with 10 qubits
qcl> qureg a[4];    // allocate a 4-qubit register
qcl> qureg b[3];    // allocate another 3-qubit register
qcl> print a,b;     // show actual qubit mappings
: |......3210> |...210....>
qcl> qureg c[5];    // try to allocate another 5 qubits
! memory error: not enough quantum memory
```

In QCL, the quantum heap is organized as a stack: qubits are pushed on allocation and poped on deallocation. A quantum register is deallocated, when the scope of the variable is left.

```
qcl> qureg a[3];    // allocate 3 qubits
qcl> procedure foo() { qureg b[2]; print a,b; }
qcl> foo();         // temp. register b gets allocated
: |.......210> |.....10...>
qcl> qureg c[3];    // allocate another 3 qubits
qcl> print a,c;     // qubits from b have been reclaimed
: |.......210> |....210...>
```

3.3.1.5 Scratch Space Management

If temporary registers are used, then, in order to avoid the corruption of the quantum heap, it has to be assured that the register is empty befor it is deallocated. Quantum functions (see 2.2.2.5) allow the declaration of local quantum variables as scratch space (see 3.3.1.5), in which case the "uncomputing" of the temporary registers is transparently taken care of by using the following procedure suggested by Bennet: [8]

Let F be a quantum function with the argument register $\mathbf{x}$ (type `quconst`, see 3.3.2.2), the target register $\mathbf{y}$ (type `quvoid`, see 3.3.2.3) and the scratch register $\mathbf{s}$ (type `quscratch`, see 3.3.2.4)

$$F(\mathbf{x},\mathbf{y},\mathbf{s}) : |i\rangle_{\mathbf{x}}|0\rangle_{\mathbf{y}}|0\rangle_{\mathbf{s}} \to |i\rangle_{\mathbf{x}}|f(i)\rangle_{\mathbf{y}}|j(i)\rangle_{\mathbf{s}} \tag{3.6}$$

During the application of F, the register $\mathbf{s}$ is filled with the temporary junk bits $j(i)$. To reclaim $\mathbf{s}$, QCL allocates an auxiliary register $\mathbf{t}$ and translates F into an operator F' which is defined as

$$F'(\mathbf{x}, \mathbf{y}, \mathbf{s}, \mathbf{t}) = F^{\dagger}(\mathbf{x}, \mathbf{t}, \mathbf{s})\, \mathit{Fanout}(\mathbf{t}, \mathbf{y})\, F(\mathbf{x}, \mathbf{t}, \mathbf{s}) \tag{3.7}$$

The *fanout* operator is a quantum function defined as

$$\mathit{Fanout} : |i\rangle|0\rangle \to |i\rangle|i\rangle \tag{3.8}$$

The application of F' restores the scratch register $\mathbf{s}$ and the auxiliary register $\mathbf{a}$ to $|0\rangle$ while preserving the function value in the target register $\mathbf{t}$:

$$|i, 0, 0, 0\rangle \to |i, 0, j(i), f(i)\rangle \to |i, f(i), j(i), f(i)\rangle \to |i, f(i), 0, 0\rangle \tag{3.9}$$

3.3.1.6 Simulation

The interpreter `qcl` can simulate quantum computers with arbitrary numbers of qubits. According to the hybrid architecture as introduced in 3.1.3, the numerical simulations are handled by a library (`libqc`) to separate the classical program state from the quantum machine state. QCL provides special commands for inspecting the simulated machine state.

The `dump` command prints the current machine state in Braket notation. When a quantum expression is given, it prints the probability spectrum instead.

```
qcl> qureg q[2];
qcl> Mix(q);
qcl> dump;
: STATE: 2 / 4 qubits allocated, 2 / 4 qubits free
0.5 |0000> + 0.5 |0010> + 0.5 |0001> + 0.5 |0011>
qcl> dump q[0];
: SPECTRUM q[0]: |...0>
0.5 |0> + 0.5 |1>
```

The current machine-state can be loaded and saved with the `load` and `save` command.

3.3.2 Quantum Variables

Quantum registers bound to a symbolic name are referred to as *quantum variables.*

3.3.2.1 General Registers

A general quantum Register with $n =$ *expr* qubits can be declared with

$$\textit{var-def} \leftarrow \texttt{qureg}\ \textit{identifier}\ \texttt{[}\ \textit{expr}\ \texttt{]}\ \texttt{;}$$

Empty quantum memory is allocated from the heap and bound to the symbol *identifier*.

A global declaration defines a permanent quantum register which is not to prone to scratch space management. This means that — as with classic global variables — there is no way to reclaim allocated qubits within the same scope.

The reseting of the machine state with the `reset` command has no effect on register bindings.

```
[0/4] 1 |0000>
qcl> qureg q[1];             // allocate a qubit
qcl> reset;                  // reset: |Psi> -> |0>
[1/4] 1 |0000>
qcl> list q;                 // register q still exists
: global symbol q = |...0>:
qureg q[1];
```

The quantum types `quvoid` and `quscratch` are restricted to pseudo-classical operators (`qufunct`) and are equivalent to `qureg`, except that they are treated differently by memory management (see 3.3.1.5 for details).

3.3.2.2 Quantum Constants

Registers can be declared constant, by using the register type `quconst`. A quantum constant has to be invariant to all applied operators.

Definition 10 (Invariance of Registers) *A quantum register* $\mathbf{c}$ *is considered invariant to a register operator* $U(\mathbf{s},\mathbf{c})$ *if* U *meets the condition*

$$U : |i,j\rangle = |i\rangle_{\mathbf{s}}|j\rangle_{\mathbf{c}} \to (U_j\,|i\rangle_{\mathbf{s}})\,|j\rangle_{\mathbf{c}} \tag{3.10}$$

Quantum constants have a fixed probability spectrum: Let $|\Psi\rangle = \sum a_{ij}|i,j\rangle$ be the machine state and $|\Psi'\rangle = U(\mathbf{s},\mathbf{c})\,|\Psi\rangle$ and $p(J)$ and $p'(J)$ the probabilities to measure J in register $\mathbf{c}$ before and after the operator is applied, then

$$p(J) = \langle\Psi|P_J|\Psi\rangle = \sum_i a^*_{iJ}a_{iJ} \quad \text{with} \quad P_J = \sum_k |k,J\rangle\langle k,J| \tag{3.11}$$

$$\begin{aligned} p'(J) &= \langle\Psi'|P_J|\Psi'\rangle = \langle\Psi|U^\dagger P_J U|\Psi\rangle = \\ &= \sum_{i',j',i,j} a^*_{i'j'}a_{ij}\ (\langle i'|_{\mathbf{s}}U^\dagger_{j'}\,\langle j'|_{\mathbf{c}})\ P_J\ (U_j\,|i\rangle_{\mathbf{s}}\,|j\rangle_{\mathbf{c}}) = \\ &= \sum_{i',i} a^*_{i'J}a_{iJ}\ \langle i|U^\dagger_J U_J|i\rangle = p(J) \end{aligned} \tag{3.12}$$

If an argument to an operator is declared as `quconst`, the register has to be invariant to all subsequent operator calls within the operator definition.

```
qcl> operator foo(quconst c) { Rot(pi,c); }
! in operator foo: parameter mismatch: quconst used as non-const
argument to Rot
```

When used as an argument type to a quantum function, constant registers aren't swapped out when local scratch registers are uncomputed (see 3.3.1.5).

3.3.2.3 Empty Registers

If an argument **v** to an operator is declared `quvoid`, the quantum register is expected to be empty when the operator is called normally, or to be uncomputed if the operator is called inverted (see 3.4.3.2). So, depending on the adjungation flag of the operator, the machine state $|\Psi\rangle$ has to conform to either

$$U(\mathbf{v},\ldots): |\Psi\rangle = |0\rangle_{\mathbf{v}}|\psi\rangle \to |\Psi'\rangle \quad \text{or} \quad U^{\dagger}(\mathbf{v},\ldots): |\Psi\rangle \to |0\rangle_{\mathbf{v}}|\psi'\rangle \tag{3.13}$$

This can be checked at runtime with simulator the option `--check`.

```
qcl> qureg q[4];
qcl> qureg p[4];
qcl> set check 1;        // turn on consistency checking
qcl> Rot(pi/100,p[2]);   // slightly rotate one target qubit
[8/8] 0.999877 |00000000> + -0.0157073 |01000000>
qcl> Fanout(q,p);        // p is assumed void
! in qufunct Fanout: memory error: void or scratch register not empty
```

When used as an argument type to a quantum function, void registers are swapped out to a temporary register if local scratch registers are uncomputed.

3.3.2.4 Scratch Registers

As an argument **s** to an operator, registers of type `quscratch` are considered to be explicit scratch registers which have to be empty when the operator is called and have to get uncomputed before the operator terminates, so operator and machine state have to satisfy the condition

$$U(\mathbf{s},\ldots): |\Psi\rangle = |0\rangle_{\mathbf{s}}|\psi\rangle \to |0\rangle_{\mathbf{s}}|\psi'\rangle = |\Psi'\rangle \tag{3.14}$$

If a scratch register is defined within the body of a quantum function, Bennet's method of "uncomputing" temporary registers (see 3.3.1.5) is used to free the register again.

Quantum functions using local scratch registers may not take general (`qureg`) registers as arguments.

```
qcl> qufunct nop(qureg q) { quscratch s[1]; }
! invalid type: local scratch registers can't be used with
qureg arguments
```

3.3.2.5 Register References

To conveniently address subregisters or combined registers (see below), quantum expressions can be named by declaring a register reference.

$$\textit{def} \leftarrow \textit{type identifier} \; [\, \texttt{=} \; \textit{expr} \,] \; \texttt{;}$$

The quantum expression *expr* is bound to the register *identifier* of the quantum type *type* which can be `qureg` or `quconst`.

```
qcl> qureg q[8];
qcl> qureg oddbits=q[1]&q[3]&q[5]&q[7];
qcl> qureg lowbits=q[0:3];
qcl> list q,oddbits,lowbits;
: global symbol q = |........76543210>:
qureg q[8];
: global symbol oddbits = |........3.2.1.0.>:
qureg oddbits;
: global symbol lowbits = |...........3210>:
qureg lowbits;
```

References can also be used to override type-checking by redeclaring a `quconst` as `qureg`, which can be useful if a constant argument should be temporarily used as scratch space but is restored later.

3.3.3 Quantum Expressions

A quantum expression is an anonymous register reference, which can be used as an operator argument or to declare named references (see above).

Expr.	Description	Register
`a`	reference	$\langle a_0, a_1 \ldots a_n \rangle$
`a[i]`	qubit	$\langle a_i \rangle$
`a[i:j]`	substring	$\langle a_i, a_{i+1} \ldots a_j \rangle$
`a[i\l]`	substring	$\langle a_i, a_{i+1} \ldots a_{i+l-1} \rangle$
`a&b`	concatenation	$\langle a_0, a_1 \ldots a_n, b_0, b_1 \ldots b_m \rangle$

Table 3.5: quantum expressions

3.3.3.1 Subregisters

Subregisters can be addressed with the subscript operator `[...]`. Depending on the syntax (see table 3.5), single qubits are specified by their zero-based offset and substrings are specified by the offset of the first qubit and either the offset of the last qubit (syntax `[·:·]`) or the total length of the subregister (syntax `[·\·]`).

```
qcl> qureg q[8];
qcl> print q[3],q[3:4],q[3\4];
: |....0...> |...10...> |.3210...>
```

Indices can be arbitrary expressions of type `int`. Invalid subscripts trigger an error.

```
qcl> int i=255;
qcl> print q[floor(log(i,2))];
: |0.......>
qcl> print q[floor(log(i,2))\2];
! range error: invalid quantum subregister
```

3.3.3.2 Combined Registers

Registers can be combined with the concatenation operator `&`. If the registers overlap, an error is triggered.

```
qcl> print q[4:7]&q[0:3];
: |32107654>
qcl> print q[2]&q[0:3];
! range error: quantum registers overlap
```

3.4 Quantum Operations

3.4.1 Non-unitary Operations

As pointed out in 3.1.3, any quantum computation must be a composition of initializations, unitary operators and measurements. A typical probabilistic quantum algorithm usually runs an evaluation loop like this:

```
{
  reset;            // R: |Psi> -> |0>
  myoperator(q);    // U: |0> -> |Psi'>
  measure q,m;      // M: |Psi'> -> |m>
} until ok(m);      // picked the right m ?
```

The `reset` command resets the machine-state $|\Psi\rangle$ to $|0\rangle$, which is also the initial state when `qcl` is started. The quantum heap and the binding of quantum variables are unaffected.

$$\textit{stmt} \leftarrow \texttt{measure}\ \textit{expr}\ [\ ,\ \textit{identifier}]\ ;$$

The `measure` command measures the quantum register *expr* and assigns the measured bit-string to the `int` variable *identifier*. If no variable is given, the value is discarded.

The outcome of the measurement is determined by a random number generator, which — by default — is initialized with the current system time. For reproducible behavior of the simulation, a seed value can be given with the option `--seed`.

Since `reset` and `measure` operations are irreversible, they must not occur within operator definitions.

3.4.2 Subroutines

3.4.2.1 Hierarchy of Subroutines

Besides the classical subroutine type `procedure` and `function`, QCL provides two quantum routine types for general unitary operators (`operator`) and pseudo-classical operators (`qufunct`). QCL allows to invert operators and can perform scratch-space management for quantum functions, thus allowed side effects on the classical program state as well as on the quantum machine state have to be strictly specified.

routine type	**program state**	**machine state**	**recursion**
`procedure`	all	all	yes
`operator`	none	unitary	no
`qufunct`	none	pseudo-classical	no
functions	none	none	yes

Table 3.6: hierarchy of QCL Subroutines and allowed side-effects

The 4 QCL routine types form a call hierarchy, which means that a routine may invoke only subroutines of the same or a lower level (see table 3.6).

The mathematical semantic of QCL operators and functions requires that every call is reproducible. This means, that not only the program state must not be changed by these routines, but also that their execution may in no way depend on the global program state which includes global variables, options and the state of the internal random number generator.

3.4.2.2 External Routines

While QCL incorporates a classical programming language, to provides all the necessary means to change the program state, there is no hardwired set of elementary operators to manipulate the quantum machine state, since this would require assumptions about the architecture of the simulated quantum computer.

An elementary `operator` or `qufunct` can be incorporated by declaring it as `extern`.

$$\begin{array}{lcl} \textit{def} & \leftarrow & \texttt{extern operator}\ \textit{identifier arg-list}\ \texttt{;} \\ & \leftarrow & \texttt{extern qufunct}\ \textit{identifier arg-list}\ \texttt{;} \end{array}$$

External operators have no body since they are not executed within QCL, but merely serve as a hook for a binary function which implements the desired operation directly by using the numeric QC-library and is linked to the interpreter.

Section 3.4.4 and 3.4.7 describe the elementary unitary and pseudo classic gates which are provided by the integrated simulator of `qcl`.

3.4.3 General Operators

The routine type `operator` is used for general unitary operators. Conforming to the mathematical notion of an operator, a call with the same parameters has to result in exactly the same transformation, so no global variable references, random elements or dependencies on input are allowed.

Since the type `operator` is restricted to reversible transformations of the machine state, `reset` and `measure` commands are also forbidden.

3.4.3.1 Operator Arguments

Operators work on one or more quantum registers so a call of an m qubit operator with a total quantum heap of n qubits can result in $\frac{n!}{(n-m)!}$ different unitary transformations.

In QCL, this polymorphism is even further extended by the fact, that quantum registers can be of different sizes, so for every quantum parameter $\mathbf{s}$, the register size $\#\mathbf{s} = |\mathbf{s}|$ is an implicit extra parameter of type `int`. An addition to that, operators can take an arbitrary number of explicit classical arguments.

If more than one argument register is given, their qubits may not overlap.

```
qcl> qureg q[4];
qcl> qureg p=q[2:3];
qcl> CNot(q[1\2],p);
! runtime error: quantum arguments overlapping
```

3.4.3.2 Inverse Operators

Operator calls can be inverted by the adjungation prefix '`!`'. The adjoint operator to a composition of unitary operators is[3]

$$\left(\prod_{i=1}^{n} U_i\right)^\dagger = \prod_{i=n}^{1} U_i^\dagger \tag{3.15}$$

Since the sequence of applied suboperators is specified using a procedural classical language which cannot be executed in reverse, the inversion of the composition, is is achieved by the delayed execution of operator calls.

When the adjungation flag is set, the operator body is executed and all calls of suboperators are pushed on a stack which is then processed in reverse order with inverted adjungation flags.

3.4.3.3 Local Registers

As opposed to pseudo-classical operators, it is in general impossible to uncompute a unitary operator in order to free a local register again without also destroying the intended result of the computation. This is a fundamental limitation of QC known as the *non cloning theorem* which results from the fact that a cloning operation i.e. a transformation with meets the condition

$$U : |\psi\rangle|0\rangle \to |\psi\rangle|\psi\rangle \tag{3.16}$$

for an arbitrary[4] $|\psi\rangle$ cannot be unitary if $|\psi\rangle$ is a composed state because

$$U\,(a|0,0\rangle + b|1,0\rangle) = a^2|0,0\rangle + ab\,|0,1\rangle + ba\,|1,0\rangle) + b^2|1,1\rangle \tag{3.17}$$

$$\neq a\,U\,|0,0\rangle + b\,U\,|1,0\rangle = a^2|0,0\rangle + b^2|1,1\rangle \tag{3.18}$$

U can only be unitary if $|\psi\rangle$ is a basis state, i.e. $|\psi\rangle = |i\rangle$, in which case $U = \mathit{Fanout}$.

Due to the lack of a unitary copy operation for quantum states, Bennet-style scratch space management is impossible for general operators since it is based on cloning the result register.

Despite this limitation, it is possible in QCL to allocate temporary quantum registers but it is up to the programmer to properly uncompute them again. If the option `--check` is set, proper cleanup is verified by the simulator.

---

[3] To avoid ambiguities with non-commutative matrix products, we use the convention $\prod_{i=1}^{n} f_i = f_n f_{n-1} \cdots f_1$

[4] For any particular $|\psi\rangle$ an infinite number of unitary "cloning" operators trivially exists, as e.g. $U_\psi = \sum_{i,j,k} |i, j \oplus k\rangle\langle k|\psi\rangle\langle i, j|$

```
qcl> set check 1
qcl> operator foo(qureg q) { qureg p[1]; CNot(p,q); }
qcl> qureg q[1];
qcl> Mix(q);
[1/4] 0.707107 |0000> + 0.707107 |0001>
qcl> foo(q);
! in operator foo: memory error: quantum heap is corrupted
[1/4] 0.707107 |0000> + 0.707107 |0011>
```

Local registers are useful if an operator contains some intermediary pseudo-classical operations which require scratch space.

3.4.4 Unitary Gates

3.4.4.1 Unitary Matrices

The most general form for specifying a unitary operator (or any other linear transformation) is by defining it's matrix elements: An n qubit unitary operator U describes a transformation $U : \mathbf{C}^{2^n} \to \mathbf{C}^{2^n}$ and therefore corresponds to a $2^n \times 2^n$ matrix in $\mathbf{C}$

$$U = \sum_{i,j=0}^{2^n} |i\rangle u_{ij} \langle j| = \begin{pmatrix} u_{0,0} & \cdots & u_{0,2^n-1} \\ \vdots & \ddots & \vdots \\ u_{2^n-1,0} & \cdots & u_{2^n-1,2^n-1} \end{pmatrix} \tag{3.19}$$

Since for a unitary transformation $U^\dagger U = (U^*)^{\mathrm{T}} U = I(n)$, the Matrix U unitary if and only if

$$\bigwedge_{i,j=0}^{2^n-1} \sum_{k=0}^{2^n-1} u^*_{ki} u_{kj} = \delta_{ij} \tag{3.20}$$

QCL provides external operators for general unitary 2×2, 4×4 and 8×8 matrices, which the programmer can use to directly implement a custom set of 1, 2 and 3 qubit gates.

```
extern operator Matrix2x2(
  complex u00,complex u01,
  complex u10,complex u11,
qureg q);
extern operator Matrix4x4(...,qureg q);
extern operator Matrix8x8(...,qureg q);
```

Matrix operators are checked for unitarity before they are applied:

```
qcl> const i=(0,1);
qcl> qureg q[1];
qcl> Matrix2x2(i*cos(pi/6),i*sin(pi/6),(0,0),(1,0),q);
! external error: matrix operator is not unitary
```

3.4.4.2 Qubit Rotation

The rotation of a single qubit is defined by the transformation matrix $U(\theta)$

$$U(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & \sin\frac{\theta}{2} \\ -\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix} \tag{3.21}$$

The factor $-\frac{1}{2}$ to θ is set in analogy to spin rotations, which can be shown to be of the form $\mathcal{D} = \mathrm{e}^{-\frac{\mathrm{i}}{2}\delta_j\sigma_j}$ and thus have a period of 4π.

```
extern operator Rot(real theta,qureg q);
```

In contrast to most other external Operators, `Rot` is not generalized to work with arbitrary register sizes.

```
qcl> Rot(pi/2,q);
! external error: Only single qubits can be rotated
```

3.4.4.3 Hadamard Gate

The *Hadamard Gate* is a special case of a generalized qubit Rotation and defined by the transformation matrix H

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{3.22}$$

For the case of n-qubit registers, H can be generalized to

$$H : |i\rangle \to 2^{-\frac{n}{2}} \sum_{j \in \mathbf{B}^n} (-1)^{(i,j)} |j\rangle \tag{3.23}$$

The vectors $\mathcal{B}' = \{i \in \mathbf{B}^n \mid |i'\rangle = H\,|i\rangle\}$ form the *Hadamard base* or *dual base* or *parity base* to $\mathcal{B} = \{i \in \mathbf{B}^n \mid |i\rangle\}$.

The Hadamard Transformation is self adjoint (i.e. $H^\dagger = H$), which, for unitary operators, implies that $H^2 = I$.

Since $\mathcal{B}'$ only contains uniform superpositions that just differ by the signs of the base-vectors, the external implementation of H is called `Mix`.

```
extern operator Mix(qureg q);
```

3.4.4.4 Conditional Phase Gate

The *conditional phase gate* is a pathological case of a conditional operator (see 2.2.2.6), for the zero-qubit phase operator $\mathrm{e}^{\mathrm{i}\phi}$.

$$V(\phi) : |\epsilon\rangle \to \begin{cases} \mathrm{e}^{\mathrm{i}\phi}\,|\epsilon\rangle & \text{if } \epsilon = 111\ldots \\ |\epsilon\rangle & \text{otherwise} \end{cases} \tag{3.24}$$

The conditional phase gate is used in the quantum Fourier transform (see 4.2.3).

```
extern operator CPhase(real phi,qureg q);
```

3.4.5 Pseudo-classical Operators

The routine type `qufunct` is used for pseudo-classical operators and quantum functions, so all transformations have to be of the form

$$|\Psi\rangle = \sum_i c_i |i\rangle \to \sum_{i,j} c_i \delta_{j\pi_i} |j\rangle = |\Psi'\rangle \tag{3.25}$$

with some permutation π. All n-qubit pseudo-classical operators F therefore have the common eigenstate

$$|\Psi\rangle = 2^{-\frac{n}{2}} \sum_{i=0}^{2^n-1} |i\rangle \iff F\,|\Psi\rangle = |\Psi\rangle \tag{3.26}$$

3.4.5.1 Bijective Functions

The most straightforward application for pseudo-classical operators is the direct implementation of bijective functions (see 2.2.2.4)

```
qufunct inc(qureg x) {
  int i;
  for i = #x-1 to 1 {
    CNot(x[i],x[0:i-1]);
  }
  Not(x[0]);
}
```

The operator `inc` shifts the base-vectors of it's argument. In analogy to boson states, where the increment of the eigenstate corresponds to the generation of a particle, `inc` is a *creation operator*.[5]

```
qcl> qureg q[4];
qcl> inc(q);
[4/4] 1 |0001>
qcl> inc(q);
[4/4] 1 |0010>
qcl> inc(q);
[4/4] 1 |0011>
qcl> inc(q);
[4/4] 1 |0100>
```

3.4.5.2 Conditional Operators

When it comes to more complicated arithmetic operations, it is often required to apply a transformation to a register **a** in dependence on the content of another register **e**.

[5] In fact, this is not quite correct, since other than bosons, an n qubit register is limited to 2^n states, so `inc` $|2^n-1\rangle = |0\rangle$ whereas $a^\dagger\,|2^n-1\rangle = |2^n\rangle$

If all qubits of **e** are required to be set, for the transformation to take place, the operator is a conditional operator with the invariant (`quconst`) enable register **e** (see 2.2.2.6).

A simple example for a conditional operator is the Toffoli gate

$$T : |x, y, z\rangle \to |x \oplus (y \wedge z), y, z\rangle \tag{3.27}$$

or it's generalization, the controlled not gate (see 3.4.7.4). A conditional version of the above increment operator is also easy to implement:

```
qufunct cinc(qureg x,quconst e) {
  int i;
  for i = #x-1 to 1 step -1 {
    CNot(x[i],x[0:i-1] & e);
  }
  CNot(x[0],e);
}
```

Now, only base-vectors of the form $|i\rangle|11\ldots\rangle_{\mathbf{s}}$ are incremented:

```
qcl> qureg q[4]; qureg e[2]; Mix(e);
[6/6] 0.5 |000000> + 0.5 |100000> + 0.5 |010000> + 0.5 |110000>
qcl> cinc(q,e);
[6/6] 0.5 |000000> + 0.5 |100000> + 0.5 |010000> + 0.5 |110001>
qcl> cinc(q,e);
[6/6] 0.5 |000000> + 0.5 |100000> + 0.5 |010000> + 0.5 |110010>
qcl> cinc(q,e);
[6/6] 0.5 |000000> + 0.5 |100000> + 0.5 |010000> + 0.5 |110011>
```

3.4.6 Quantum Functions

As defined in 2.2.2.5, a quantum function F is a pseudo-classical operator with the characteristic

$$F : |x\rangle_{\mathbf{x}}|0\rangle_{\mathbf{y}} \to |x\rangle_{\mathbf{x}}|f(x)\rangle_{\mathbf{y}} \quad \text{with} \quad f : \mathbf{B}^n \to \mathbf{B}^m \tag{3.28}$$

If we require the argument register **x** to be invariant to F by declaring **x** as `quconst`, this leaves us with $((2^m - 1)!)^{2^n}$ possible pseudo-classical implementations of F for any given f. To reflect the fact that $F\,|x, y \neq 0\rangle$ is undefined, the target register has to be of type `quvoid`. (see 3.3.2.3).

Since, according to the above definition, quantum functions are merely ordinary pseudo-classical operators, whose specification is restricted to certain types of input states, they also use the same QCL routine type `qufunct`.

The following example calculates the parity of **x** and stores it to **y**:

```
qufunct parity(quconst x,quvoid y) {
  int i;
  for i = 0 to #x-1 {
    CNot(y,x[i]);
  }
}

qcl> qureg x[2]; qureg y[1]; Mix(x);
[3/3] 0.5 |000> + 0.5 |010> + 0.5 |001> + 0.5 |011>
qcl> parity(x,y);
[3/3] 0.5 |000> + 0.5 |110> + 0.5 |101> + 0.5 |011>
```

3.4.6.1 Scratch parameters

We can extend the notion of quantum functions, by also allowing an explicit scratch register $\mathbf{s}$ (see 3.3.2.4) as an optional parameter to F, so we end up with an operator $F(\mathbf{x}, \mathbf{y}, \mathbf{s})$ with the characteristic

$$F : |x\rangle_\mathbf{x}|0\rangle_\mathbf{y}|0\rangle_\mathbf{s} \to |x\rangle_\mathbf{x}|f(x)\rangle_\mathbf{y}|0\rangle_\mathbf{s} \tag{3.29}$$

Using the `parity` and the `cinc` operator form the above examples, we can implement an "add parity" function $f(x) = x + \mathrm{parity}(x)$ by providing a scratch qubit:

```
qufunct addparity(quconst x,quvoid y,quscratch s) {
  parity(x,s);      // write parity to scratch
  x -> y;           // Fanout x to y
  cinc(y,s);        // increment y if parity is odd
  parity(x,s);      // clear scratch
}

qcl2> qureg x[2]; qureg y[2]; qureg s[1]; Mix(x);
[5/8] 0.5 |00000> + 0.5 |00010> + 0.5 |00001> + 0.5 |00011>
qcl2> addparity(x,y,s);
[5/8] 0.5 |00000> + 0.5 |01110> + 0.5 |01001> + 0.5 |01111>
```

Instead of providing a explicit scratch parameter, we can, of course, also use a local register of type `qureg`, which is functionally equivalent:

```
qufunct addparity2(quconst x,quvoid y) {
  qureg s[1];
  parity(x,s);
  x -> y;
  cinc(y,s);
  parity(x,s);
}

qcl2> qureg x[2]; qureg y[2]; Mix(x);
[4/8] 0.5 |00000> + 0.5 |00010> + 0.5 |00001> + 0.5 |00011>
qcl2> addparity2(x,y);
[4/8] 0.5 |00000> + 0.5 |01110> + 0.5 |01001> + 0.5 |01111>
```

Explicit scratch parameters are useful to save memory, if a quantum function F is to be used by another operator U, which still has empty scratch registers at the moment, the suboperator is called, which would e.g. be the case if U is of the form

$$U(\mathbf{x},\mathbf{y},\mathbf{s},\ldots) = \left(\prod_{i=2}^{l} U_i(\mathbf{x},\mathbf{y},\mathbf{s},\ldots)\right) F(\mathbf{x},\mathbf{y},\mathbf{s})\, U_1(\mathbf{x},\mathbf{y},\ldots) \tag{3.30}$$

Since both, explicit scratch parameters of type `quscratch` and local registers of type `qureg`, have to be uncomputed manually, they are especially useful for quantum functions $U : |x,0,0\rangle \to |x, f(s(x),x),0\rangle$ of the form

$$U(\mathbf{x},\mathbf{y},\mathbf{s}) = S(\mathbf{x},\mathbf{s})F(\mathbf{x},\mathbf{s},\mathbf{y})S^{\dagger}(\mathbf{x},\mathbf{s}) \tag{3.31}$$

if S is invariant to $\mathbf{x}$ and F is invariant to $\mathbf{x}$ and $\mathbf{s}$, because the uncomputation of $\mathbf{s}$ doesn't require an additional register to temporarily save $\mathbf{y}$ (see 3.3.1.5) as would be the case, if a managed local scratch register of type `quscratch` would be used instead (see below).

3.4.7 Pseudo-classical Gates

3.4.7.1 Base Permutation

The most general form for specifying an n qubit pseudo-classical operator U, is by explicitly defining the underlying permutation π of base-vectors:

$$U_{pc.} = \sum_{i=0}^{2^n-1} |\pi_i\rangle\langle i| = \langle \pi_0, \pi_1 \ldots \pi_{2^n-1}\rangle \tag{3.32}$$

QCL provides external operators for vector permutations for $|\pi| = 2$, 4, 8, 16, 32 and 64 which the programmer can use to directly implement a custom set of 1 to 6 qubit pseudo-classical operators:

```
extern qufunct Perm2(int p0 ,int p1 ,qureg q);
extern qufunct Perm4(int p0 ,int p1 ,int p2 ,int p3 ,qureg q);
extern qufunct Perm8(...,qureg q);
extern qufunct Perm16(...,qureg q);
extern qufunct Perm32(...,qureg q);
extern qufunct Perm64(...,qureg q);
```

Base permutations are checked for unitarity before they are applied (i.e. it is verified that the given integer sequence is in fact a permutation)

```
qcl> qureg q[3];
qcl> Perm8(0,0,1,2,3,4,5,6,q);
! external error: no permutation
```

3.4.7.2 Fanout

The *Fanout* operation is a quantum function (see 2.2.2.5) and stands for a class of transformations with the characteristic $Fanout : |x, 0\rangle \to |x, x\rangle$

The external fanout operator of QCL is defined as

$$Fanout : |x, y\rangle \;\to\; |x, x \oplus y\rangle, \tag{3.33}$$

however, it is considered bad programming style to rely on this particular implementation.

```
extern qufunct Fanout(quconst a,quvoid b);
```

QCL also provides the special syntax `a->b` and `a<-b` as abbreviations for `Fanout(a,b)` and `!Fanout(a,b)`.

3.4.7.3 Swap

The *Swap* operator exchanges the qubits of two equal sized registers ($Swap : |x, y\rangle \to |y, x\rangle$). A one to one qubit *Swap* operator has the transformation matrix

$$Swap = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.34}$$

```
extern qufunct Swap(qureg a,qureg b);
```

As with the fanout operator, `a<->b` is syntactic sugar for `Swap(a,b)`.

3.4.7.4 Not and Controlled Not

The not operator C inverts a qubit. Its transformation matrix is

$$C = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{3.35}$$

The controlled-not operator $C_{[[\mathbf{e}]]}$ is the conditional operator (see 2.2.2.6) to C with the enable register $\mathbf{e}$:

$$C_{[[\mathbf{e}]]} : |b\rangle|\epsilon\rangle_{\mathbf{e}} \to \begin{cases} |1-b\rangle\,|\epsilon\rangle_{\mathbf{e}} & \text{if } \epsilon = 111\ldots \\ |b\rangle|\epsilon\rangle_{\mathbf{e}} & \text{otherwise} \end{cases} \tag{3.36}$$

```
extern qufunct Not(qureg q);
extern qufunct CNot(qureg q,quconst c);
```

The QCL versions of `Not` and `CNot` also work on target registers, in which case $C_{[[\mathbf{e}]]}$ is applied to all qubits:

```
qcl> qureg q[4]; qureg p[4];
qcl> Not(q);
[8/8] 1 |00001111>
qcl> CNot(p,q);
[8/8] 1 |11111111>
```

3.5 Programming Techniques

3.5.1 Design of Quantum Algorithms

As has been shown in 3.1.2, quantum computers and probabilistic classical computers are computationally equivalent, but for certain tasks, quantum algorithms can provide a more efficient solution than classical implementations.

In order to achieve any speedup over classical algorithms, it is necessary to take advantage of the unique features of quantum computing namely

- Superpositioning
- Quantum Parallelism
- Interference

3.5.1.1 Superpositioning

A key element in any universal programming language is conditional branching. Any classical program can be modeled as a decision tree where each node corresponds to a binary state s_n and leads to one or more successor states $s_{n+1}^{(i)}$. On a deterministic Turing machine (TM), only one of those transitions $s_n \to s_{n+1}^{(k)}$ is possible, so the computational path $\langle s_0, s_1, \ldots s_n \rangle$ is predetermined.

On a probabilistic TM, the transitions are characterized by probabilities p_i with $\sum_i p_i = 1$ and one of the possible successor states $s_{n+1}^{(i)}$ is chosen accordingly at random.

Since the eigenvectors $|i\rangle$ directly correspond to classical binary states, we might interpret a unitary transformation

$$U : |s\rangle \to \sum_{s'} u_{ss'} |s'\rangle \quad \text{with} \quad s, s' \in \mathbf{B}^n \quad \text{and} \quad u_{ss'} \in \mathbf{C} \tag{3.37}$$

as a probabilistic transition form the classical state s to the successor states s' with the transition probabilities $p_{s'} = |u_{ss'}|^2$, but unless we perform a measurement, the resulting machine state remains in a superposition of all possible classical successor states

$$|\Psi\rangle = |s_n\rangle \xrightarrow{U} |\Psi'\rangle = \sum_i u_{s_n s_{n+1}^{(i)}} |s_{n+1}^{(i)}\rangle \tag{3.38}$$

So from a classical point of view, we can consider a unitary operator which transforms an eigenstate into a superposition of n eigenstates with nonzero amplitudes as a 1–n fork-operation, which enables a quantum computer to follow several classical computational paths at once.

Most non-classical algorithms take advantage of this feature by bringing a register into a even superposition of eigenstates to serve as search space. This can be achieved by applying the *Hadamard* transformation (see 3.4.4.3) to each qubit

```
[0/4] 1 |0000>
qcl> qureg q[2];         // allocate 2-qubit register
qcl> Mix(q[0]);          // rotate first qubit
[2/4] 0.707107 |0000> + 0.707107 |0001>
qcl> Mix(q[1]);          // rotate second qubit
[2/4] 0.5 |0000> + 0.5 |0010> + 0.5 |0001> + 0.5 |0011>
```

Classically, this can be viewed as a binary decision tree with a 50% chance for each bit to flip. For an n-qubit register, this leads to 2^n classical computational paths all of which are followed simultaneously resulting in a superposition of 2^n eigenvectors.

Since the *Hadamard* transforms for each single qubit commute, we can a-posteriori emulate classic probabilistic behavior by performing a measurement on the single qubits; thereby, the temporal order of the measurements is unimportant so we can force a decision on the second qubit before we decide on the the first and reconstruct the classical computational path in reverse

```
qcl> measure q[1];       // second qubits gives 0
[2/4] 0.707107 |0000> + 0.707107 |0001>
qcl> measure q[0];       // first qubit gives 1
[2/4] 1 |0001>
```

3.5.1.2 Quantum Parallelism

If we restrict unitary transformations to pseudo-classical operators (see 2.2.2.4) then the classical decision tree degenerates into a list and we end up with the functionality of a classical reversible computer i.e. for any bijective binary function f there is a corresponding pseudo-classical operator

$$U_f : |s\rangle \to |f(s)\rangle \quad \text{with} \quad s \in \mathbf{B}^n \quad \text{and} \quad f : \mathbf{B}^n \to \mathbf{B}^n \tag{3.39}$$

The restriction to bijective functions is not a severe as it seems, since for any general binary function g a corresponding quantum function

$$U_g : |s, 0\rangle \to |s, g(s)\rangle \quad \text{with} \quad s \in \mathbf{B}^n \quad \text{and} \quad f : \mathbf{B}^n \to \mathbf{B}^n \tag{3.40}$$

can be constructed, which implements g with a maximum overhead of $O(\sqrt{n})$ in space- and $O(2)$ time-complexity, so besides this minor performance penalty, a quantum computer with only pseudo-classical operators is functionally equivalent to a deterministic classical computer.

However, if we use a quantum function on an superposition of eigenstates, the same classical computation is performed on all bit-strings simultaneously.

$$|\Psi\rangle = \sum_s |s, 0\rangle \xrightarrow{U_g} |\Psi'\rangle = \sum_s |s, g(s)\rangle \tag{3.41}$$

In classical terms, this can be described as a SIMD (single instruction, multiple date) vector operation, in quantum terms this feature is referred to as *quantum parallelism*.

As an example, lets consider a full binary adder

$$ADD(\mathbf{a}, \mathbf{b}, \mathbf{s}) : |a\rangle_\mathbf{a} |b\rangle_\mathbf{b} |0\rangle_\mathbf{s} \to |a\rangle_\mathbf{a} |b\rangle_\mathbf{b} |a+b\rangle_\mathbf{s} \tag{3.42}$$

Using the *controlled-not* operator $C_{[[\mathbf{e}]]}$ (see 3.4.7.4), this can be implemented as

$$ADD(\mathbf{a}, \mathbf{b}, \mathbf{s}) = C_{[[\mathbf{ab}]]}(\mathbf{s_1})\, C_{[[\mathbf{b}]]}(\mathbf{s_0})\, C_{[[\mathbf{a}]]}(\mathbf{s_0}) \quad \text{with} \quad \mathbf{s} = \mathbf{s_0 s_1} \tag{3.43}$$

If we put the argument qubits $\mathbf{a}$ and $\mathbf{b}$ into an even superposition of $|0\rangle$ and $|1\rangle$, then we can perform the addition on all possible combinations of inputs simultaneously:

```
qcl> qureg a[1];          // argument a
qcl> qureg b[1];          // argument b
qcl> qureg s[2];          // target register s=a+b
qcl> Mix(a & b);          // bring arguments into superposition
[4/4] 0.5 |0000> + 0.5 |0010> + 0.5 |0001> + 0.5 |0011>
qcl> CNot(s[0],a);        // calculate low bit of sum
[4/4] 0.5 |0000> + 0.5 |0010> + 0.5 |0101> + 0.5 |0111>
qcl> CNot(s[0],b);
[4/4] 0.5 |0000> + 0.5 |0110> + 0.5 |0101> + 0.5 |0011>
qcl> CNot(s[1],a & b);    // calculate high bit of sum
[4/4] 0.5 |0000> + 0.5 |0110> + 0.5 |0101> + 0.5 |1011>
```

3.5.1.3 Interference

While superpositioning and quantum parallelism allow us to perform an exponentially large number of classical computations in parallel, the only way to read out any results is by performing a measurement whereby all but one of the superpositioned eigenstates get discarded. Since it doesn't make any difference if the computational path is determined during the calculation (as with the probabilistic TM) or a-posteriori (by quantum measurement), the use of quantum computers wouldn't provide any advantage over probabilistic classical computers.

Quantum states, however, are not merely a probability distribution of binary values but are vectors i.e. each eigenstate in a superposition isn't characterized by a real probability, but a complex amplitude, so

$$|\psi_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad \text{and} \quad |\psi_2\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{3.44}$$

describe different states, even if they have the same probability spectrum.

So, while on a probabilistic TM, the probabilities of two different computational paths leading to the same final state s simply add up, this is not necessarily the case on a quantum computer since generally

$$|\alpha + \beta|^2 \neq |\alpha|^2 + |\beta|^2 \quad \text{for} \quad \alpha, \beta \in \mathbf{C} \tag{3.45}$$

To illustrate this concept, consider the three states

$$|\psi_1\rangle = |0\rangle, \quad |\psi_2\rangle = |1\rangle \quad \text{and} \quad |\psi_3\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \tag{3.46}$$

If we apply the Hadamard-transform H (see 3.4.4.3) to the eigenstates $|\psi_1\rangle$ and $|\psi_2\rangle$ we get

$$|\psi_1'\rangle = H\,|\psi_1\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \quad \text{and} \quad |\psi_2'\rangle = H\,|\psi_2\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{3.47}$$

Since $|\psi_1'\rangle$ and $|\psi_2'\rangle$ have the same probability distribution and $|\psi_3\rangle$ is merely a superposition of $|\psi_1\rangle$ and $|\psi_2\rangle$, classically we would assume that $|\psi_3'\rangle$ also shows the same probability spectrum, however

$$|\psi_3'\rangle = H\,|\psi_3\rangle = \frac{1}{\sqrt{2}}(|\psi_1'\rangle + |\psi_2'\rangle) = |0\rangle \tag{3.48}$$

so in case of $|0\rangle$ the probabilities added up while in case of $|1\rangle$, the complex amplitudes had opposing signs leading to a partial probability of 0. This phenomenon is referred to as positive or negative *interference*.

So while the computational paths on a probabilistic TM are independent, interference allows computations on superpositioned states to interact and it is this interaction which allows a quantum computer to solve certain problems more efficiently than classical computers. The foremost design principle for any quantum algorithm therefor is to use interference to increase the probability of "interesting" eigenstates while trying to reduce the probability of "dull" states, in order to raise the chance that a measurement will pick one of the former.

Since any unitary operator U can also be regarded as a base-transformation (see 1.3.2.6), the above problem can also be reformulated as finding an appropriate observable for the measurement, thereby effectively replacing the register observable $\mathcal{S}$ (see 2.2.1.5) by an observable $\tilde{\mathcal{S}}$ with the Hermitian operator

$$\tilde{S} = U(\mathbf{s})\, S\, U^\dagger(\mathbf{s}) \tag{3.49}$$

If the whole machine state is measured at once, then the eigenvalues $|\tilde{\imath}\rangle$ of $\tilde{S}$ are the column vectors of U

$$\tilde{S}\,|\tilde{\imath}\rangle = U\,S\,U^\dagger\,|\tilde{\imath}\rangle = i\,|\tilde{\imath}\rangle \quad \text{with} \quad |\tilde{\imath}\rangle = U\,|i\rangle = \sum_j u_{ji}|j\rangle \tag{3.50}$$

Fourier transformations are esp. useful, if global properties of classic functions such as periodicy are of interest for the problem.

3.5.2 Dealing with Reversibility

In 2.2.2.5 we have shown that for any non-reversible boolean function $f : \mathbf{B}^n \to \mathbf{B}^m$ there exists a set of unitary quantum functions

$$F : |x\rangle_{\mathbf{x}}|0\rangle_{\mathbf{y}} \to |x\rangle_{\mathbf{x}}|f(x)\rangle_{\mathbf{y}} \quad \text{with} \quad |\mathbf{x}| = n \quad \text{and} \quad |\mathbf{y}| = m \tag{3.51}$$

which can be used to circumvent the inherent restriction of quantum computers to reversible operations.

3.5.2.1 Register Reuse

While keeping a copy of the argument will allow us to compute non reversible functions, this also forces us to provide extra storage for intermediate results. Since longer calculations usually involve the composition of many quantum functions this would leave us with a steadily increasing amount of "junk" bits which are of no concern for the final result. A straightforward implementation of $f(x) = l(k(h(g(x))))$ already uses 3 additional registers (function values are in prefix notation, O stands for a quantum function $O : |x, 0\rangle \to |x, o(x)\rangle$, indices indicate the registers operated on):

$$|x, 0, 0, 0, 0\rangle \xrightarrow{G_{12}} |x, gx, 0, 0, 0\rangle \xrightarrow{H_{23}} |x, gx, hgx, 0, 0\rangle \xrightarrow{K_{34}} \tag{3.52}$$

$$|x, gx, hgx, khgx, 0\rangle \xrightarrow{L_{45}} |x, gx, hgx, khgx, lkhgx\rangle$$

Generally, a composition of n non-revertible functions would require $n-1$ registers to store intermediary results.

A simple and elegant solution of this problem was proposed by Bennet [8, 9]: If a composition of two non-reversible functions $f(x) = h(g(x))$ is to be computed, the scratch space for the intermediate result can be "recycled" using the following procedure:

$$|x, 0, 0\rangle \xrightarrow{G_{12}} |x, g(x), 0\rangle \xrightarrow{H_{23}} |x, g(x), h(g(x))\rangle \xrightarrow{G_{12}^\dagger} |x, 0, f(x)\rangle \tag{3.53}$$

The last step is merely the inversion of the first step and uncomputes the intermediate result. The second register can then be reused for further computations.

Without scratch-management, the evaluation of a composition of depth d needs d operations and consumes $d-1$ junk registers. Bennet's method of uncomputing can then be used to trade space against time: Totally uncomputing of all intermediate results needs $2d-1$ operations and $d-1$ scratch registers, which is useful, if the scratch can be reused in the further computation.

By a combined use of r registers as scratch and junk space, a composition of depth $d = (r+2)(r+1)/2$ can be evaluated with $2d - r - 1 = (r+1)^2$ operations. An calculation of $f(x) = l(k(j(i(h(g(x))))))$ on a 4-register machine (1 input, 1 output and 2 scratch/junk registers) would run as follows (function values are in prefix notation):

$$|x, 0, 0, 0\rangle \xrightarrow{I_{34}H_{23}G_{12}} |x, gx, hgx, ihgx\rangle \xrightarrow{G_{12}^\dagger H_{23}^\dagger} |x, 0, 0, ihgx\rangle \xrightarrow{J_{42}^\dagger K_{23} J_{42}} \tag{3.54}$$

$$|x, 0, kjihgx, ihgx\rangle \xrightarrow{L_{32}} |x, lkjihgx, kjihgx, ihgx\rangle = |x, fx, kjihgx, ihgx\rangle$$

By using this method, we can reduce the needed space by $O(1/\sqrt{d})$ with a computation overhead of $O(2)$.

3.5.2.2 Junk Registers

If the computation of a function $f(x)$ fills a scratch register with the junk bits $j(x)$ (i.e. $|x, 0, 0\rangle \to |x, f(x), j(x)\rangle$), a similar procedure can free the register again:

$$|x,0,0,0\rangle \xrightarrow{F_{123}} |x,f(x),j(x),0\rangle \xrightarrow{\mathit{Fanout}_{24}} |x,f(x),j(x),f(x)\rangle \xrightarrow{F_{123}^{\dagger}} |x,0,0,f(x)\rangle \tag{3.55}$$

Again, the last step is the inversion of the first. The intermediate step is a *Fanout* operation (see 3.4.7.2) which copies the function result into an additional empty register. Possible implementations are e.g.

$$\mathit{Fanout} : |x,y\rangle \to |x, x \oplus y\rangle \quad \text{or} \quad |x,(x+y) \bmod 2^n\rangle \tag{3.56}$$

3.5.2.3 Overwriting Invertible Functions

As pointed out in 2.2.2.4, every invertible function $f : \mathbf{Z}_{2^n} \to \mathbf{Z}_{2^n}$ has a corresponding pseudo classic operator $F : |i\rangle \to |f(i)\rangle$. While a direct implementation of F is possible with any complete set of pseudo-classical operators[6], the implementation as a quantum function can be substantially more efficient.

If we have efficient implementations of the quantum functions $U_f : |i,0\rangle \to |i,f(i)\rangle$ and $U_{f^{-1}} : |i,0\rangle \to |i,f^{-1}(i)\rangle$, then an overwriting operator F' can be constructed by using an n qubit scratch register.

$$F' : |i,0\rangle \xrightarrow{U_f} |i,f(i)\rangle \xrightarrow{\mathit{Swap}} |f(i),i\rangle \xrightarrow{U_{f^{-1}}^{\dagger}} |f(i),0\rangle \tag{3.57}$$

[6]One example would be the Toffoli gate $T : |x,y,z\rangle \to |x \oplus (y \wedge z), y, z\rangle$ which can be used to implement any pseudo-classical operator for 3 or more qubits

Chapter 4

Quantum Algorithms

This chapter introduces two quantum "killer applications" — Grover's fast quantum search and Shor's factorization algorithm — which both solve traditional problems in computing science and provide substantial speedup over the fastest known classical solutions.

4.1 Grover's Database Search

Many problems in classical computer science can be reformulated as searching a list for a unique element which matches some predefined condition. If no additional knowledge about the search-condition C is available, the best classical algorithm is a brute-force search i.e. the elements are sequentially tested against C and as soon as an element matches the condition, the algorithm terminates. For a list of N elements, this requires an average of $\frac{N}{2}$ comparisons.

By taking advantage of quantum parallelism and interference, Grover found a quantum algorithm which can find the matching element in only $O(\sqrt{N})$ steps. [20]

4.1.1 Formulating a Query

The most straightforward way, albeit not the most convenient for the algorithm, to implement the search condition is as a quantum function

$$\texttt{query} : |x, 0\rangle \to |x, C(x)\rangle \quad \text{with} \quad x \in \mathbf{B}^n \quad \text{and} \quad C : \mathbf{B}^n \to \mathbf{B} \tag{4.1}$$

as this allows us to formulate the problem within the realms of classical boolean logic.

Grover's algorithm can then be used to solve the equation $C(x) = 1$ while besides the fact that a solution exists and that it is unique, no additional knowledge about $C(x)$ is required.

Usually, the implementation of `query` will be complicated enough as not to allow an efficient algebraic solution, but since the inner structure of $C(x)$ doesn't matter for the algorithm, we can easily implement a test query with the solution n as

```
qufunct query(qureg x,quvoid f,int n) {
  int i;

  for i=0 to #x-1 {      // x -> NOT (x XOR n)
    if not bit(n,i) { Not(x[i]); }
  }
  CNot(f,x);             // flip f if x=1111..
  for i=0 to #x-1 {      // x <- NOT (x XOR n)
    if not bit(n,i) { !Not(x[i]); }
  }
}
```

A more realistic application would be the search for an encryption key in a known-plaintext attack. With p being the known plaintext to the ciphertext c, a QCL implementation could look like this:

```
qufunct encrypt(int p,quconst key,quvoid c) { ... }

qufunct query(int c,int p,quconst key,quvoid f) {
  int i;
  quscratch s[blocklength];

  encrypt(p,key,s);
  for i=0 to #s-1 {      // s -> NOT (s XOR p)
    if not bit(p,i) { Not(x[i]); }
  }
  CNot(f,x);             // flip f if s=1111..
}
```

Note that, unlike the example above, this query function uses a local scratch register, so it isn't necessary to explicitely uncompute **s**, as this will be taken care of by QCL's internal scratch space management (see 3.3.1.5).

4.1.2 The Algorithm

4.1.2.1 Setting up the Search Space

The solution space of a n bit query condition C is $\mathbf{B}^n$. On a quantum computer, this search space can be implemented as a superposition of all eigenstates of an n qubit register, i.e.

$$|\Psi\rangle = \frac{1}{\sqrt{N}} \sum_{i=0}^{N} |i\rangle \quad \text{with} \quad N = 2^n \tag{4.2}$$

In 3.5.1.1 we have shown how such a state can be prepared by a n-qubit Hadamard transform

$$H : |i\rangle \to 2^{-\frac{n}{2}} \sum_{j \in \mathbf{B}^n} (-1)^{(i,j)} |j\rangle \tag{4.3}$$

(see 3.4.4.3) of the initial machine state $|0\rangle$.

4.1.2.2 The Main Loop

The main loop of the algorithm consists of two steps

1. Perform a conditional phase shift which rotates the phase of all eigenvectors which match the condition C by π radians.

$$Q : |i\rangle \to \begin{cases} -|i\rangle & \text{if } C(i) \\ |i\rangle & \text{if } \neg C(i) \end{cases} \tag{4.4}$$

2. Apply a diffusion operator

$$D = \sum_{ij} |i\rangle d_{ij} \langle j| \quad \text{with} \quad d_{ij} = \begin{cases} \frac{2}{N} - 1 & \text{if } i = j \\ \frac{2}{N} & \text{if } i \neq j \end{cases} \tag{4.5}$$

Since only one eigenvector $|i_0\rangle$ is supposed to match the search condition C, the conditional phase shift will turn the initial even superposition into

$$|\Psi'\rangle = -\frac{1}{\sqrt{N}} |i_0\rangle + \frac{1}{\sqrt{N}} \sum_{i \neq i_0} |i\rangle \tag{4.6}$$

The effect of the diffusion operator on an arbitrary eigenvector $|i\rangle$ is

$$D\,|i\rangle = -|i\rangle + \frac{2}{N} \sum_{j=0}^{N-1} |j\rangle \tag{4.7}$$

so one iteration on a state of the form

$$|\Psi(k,l)\rangle = k|i_0\rangle + \sum_{i \neq i_0} l|i\rangle \tag{4.8}$$

amounts to

$$|\Psi(k,l)\rangle \xrightarrow{Q} |\Psi(-k,l)\rangle \xrightarrow{D} |\Psi(\frac{N-2}{N}k + \frac{2(N-1)}{N}l, \frac{N-2}{N}l - \frac{2}{N}k)\rangle \tag{4.9}$$

4.1.2.3 Number of Iterations

If the above loop operator DQ is repeatedly applied to the initial superposition

$$|\Psi\rangle = |\Psi(\frac{1}{\sqrt{N}}, \frac{1}{\sqrt{N}})\rangle = \frac{1}{\sqrt{N}} \sum_{i=0}^{N-1} |i\rangle \tag{4.10}$$

then the resulting states is still of the form $|\Psi(k,l)\rangle$ and the complex amplitudes k and l are described by the following system of recursions: [21]

$$k_{j+1} = \frac{N-2}{N} k_j + \frac{2(N-1)}{N} l_j \tag{4.11}$$

$$l_{j+1} = \frac{N-2}{N} l_j - \frac{2}{N} k_j \tag{4.12}$$

Using the substitution $\sin^2 \theta = \frac{1}{N}$ the solution of the above system can be written in closed form.

$$k_j = \sin((2j+1)\theta) \tag{4.13}$$

$$l_j = \frac{1}{\sqrt{N-1}} \cos((2j+1)\theta) \tag{4.14}$$

The probability p to measure i_0 is given as $p = k^2$ and has a maximum at $\theta = \frac{\pi}{2(2j+1)}$. Since for large lists, $\frac{1}{\sqrt{N}} \ll 1$ we can assume that $\sin\theta \approx \theta$ and $\pi \gg 2\theta$ and the number of iterations m for a maximum p is about $m = \lfloor \frac{\pi}{4}\sqrt{N} \rfloor$ with $p > \frac{N-1}{N}$ (due to rounding errors). Alternatively, if we are content with $p > \frac{1}{2}$, then $m = \lceil \frac{\pi}{8}\sqrt{N} \rceil$ iterations will do.

4.1.3 Implementation

4.1.3.1 The Query Operator

If we choose to formulate the query as quantum function with a flag qubit **f** to allow for a strictly classical implementation, as suggested in 4.1.1, then the operator Q can be constructed as

$$Q = \mathtt{query}^\dagger(\mathbf{x}, \mathbf{f})\, V(\pi)(\mathbf{f})\, \mathtt{query}(\mathbf{x}, \mathbf{f}) \tag{4.15}$$

by using the conditional phase gate $V(\phi)$ (see 3.4.4.4) and considering the flag register **f** as temporary scratch space.

4.1.3.2 The Diffusion Operator

Using the Hadamard Transform H (see 3.4.4.3) and a conditional phase rotation $R : |i\rangle = -(-1)^{\delta_{i0}}|i\rangle$, the diffusion operator

$$D = \sum_{i,j} |i\rangle \left(\frac{2}{N} - \delta_{ij}\right) \langle j| \tag{4.16}$$

can also be written as $D = HRH$ since

$$HRH = -\frac{1}{N} \sum_{i,k,j} |i\rangle \, (-1)^{(i,k)}(-1)^{\delta_{k0}}(-1)^{(k,j)} \, \langle j| \quad \text{and} \tag{4.17}$$

$$\sum_{k=0}^{N-1} (-1)^{(i,k)}(-1)^{\delta_{k0}}(-1)^{(k,j)} = -2 + \sum_{k=0}^{N-1} (-1)^{(i,k)(k,j)} = N\delta_{ij} - 2 \tag{4.18}$$

Using the `not` operator from 3.4.7.4 and a conditional phase gate $V(\phi)$ we can implement the diffusion operator as

```
operator diffuse(qureg q) {
  Mix(q);               // Hadamard Transform
  Not(q);               // Invert q
  CPhase(pi,q);         // Rotate if q=1111..
  !Not(q);              // undo inversion
  !Mix(q);              // undo Hadamard Transform
}
```

In fact, the above operator implements $-D$, but since overall phases make no physical difference, this doesn't matter.

4.1.3.3 The Procedure grover

By using the above, we can now give a QCL implementation of the complete algorithm:

```
procedure grover(int n) {
  int l=floor(log(n,2))+1;        // no. of qubits
  int m=ceil(pi/8*sqrt(2^l));     // no. of iterations
  int x;
  int i;
  qureg q[l];
  qureg f[1];

  {
    reset;
    Mix(q);               // prepare superposition
    for i= 1 to m {       // main loop
      query(q,f,n);       // calculate C(q)
      CPhase(pi,f);       // negate |n>
      !query(q,f,n);      // undo C(q)
      diffuse(q);         // diffusion operator
    }
    measure q,x;          // measurement
    print "measured",x;
  } until x==n;
}
```

The procedure argument n is the number to be found; the size of the quantum registers as well as the numbers of iterations are set accordingly:

```
qcl> grover(500);
: 9 qubits, using 9 iterations
: measured 500
qcl> grover(123);
: 7 qubits, using 5 iterations
: measured 74
: measured 123
qcl> grover(1234);
: 11 qubits, using 18 iterations
: measured 1234
```

4.2 Shor's Algorithm for Quantum Factorization

4.2.1 Motivation

In contrast to finding and multiplying of large prime numbers, no efficient classical algorithm for the factorization of large number is known. An algorithm is called efficient if its execution time i.e. the number of elementary operations is assymtotically polynomial in the length of its input measured in bits. The best known (or at least published) classical algorithm (the *quadratic* *sieve*) needs $O\left(\exp\left((\frac{64}{9})^{1/3}N^{1/3}(\ln N)^{2/3}\right)\right)$ operations for factoring a binary number of N bits [12] i.e. scales exponentially with the input size.

The multiplication of large prime numbers is therefore a one-way function i.e. a function which can easily be evaluated in one direction, while its inversion is practically impossible. One-way functions play a major roll in cryptography and are essential to public key crypto-systems where the key for encoding is public and only the key for decoding remains secret.

In 1978, Rivest, Shamir and Adleman developed a cryptographic algorithm based on the one-way character of multiplying two large (typically above 100 decimal digits) prime numbers. The RSA method (named after the initials of their inventors) became the most popular public key system and is implemented in many communication programs.

While it is generally believed (although not formally proved) that efficient prime factorization on a classical computer is impossible, an efficient algorithm for quantum computers has been proposed in 1994 by P.W. Shor [11].

4.2.2 The Algorithm

This section describes Shor's algorithm from a functional point of view which means that it doesn't deal with the implementation for a specific hardware architecture. A detailed implementation for the Cirac-Zoller gate can be found in [13], for a more rigid mathematical description, please refer to [15] and for a more detailed dicussion of the QCL implementation, look at [25].

4.2.2.1 Modular Exponentiation

Let $N = n_1 n_2$ with the greatest common divisor $\gcd(n_1, n_2) = 1$ be the number to be factorized, x a randomly selected number relatively prime to N (i.e. $\gcd(x, N) = 1$) and expn the modular exponentiation function with the period r:

$$\mathrm{expn}(k, N) = x^k \bmod N,\ \mathrm{expn}(k + r, N) = \mathrm{expn}(k, N),\ x^r \equiv 1 \bmod N \tag{4.19}$$

The period r is the order of $x \bmod N$. If r is even, we can define a $y = x^{r/2}$, which satisfies the condition $y^2 \equiv 1 \bmod N$ and therefore is the solution of one of the following systems of equations:

$$\begin{aligned} y_1 &\equiv 1 \bmod n_1 &&\equiv 1 \bmod n_2 \\ y_2 &\equiv -1 \bmod n_1 &&\equiv -1 \bmod n_2 \\ y_3 &\equiv 1 \bmod n_1 &&\equiv -1 \bmod n_2 \\ y_4 &\equiv -1 \bmod n_1 &&\equiv 1 \bmod n_2 \end{aligned} \tag{4.20}$$

The first two systems have the trivial solutions $y_1 = 1$ and $y_2 = -1$ which don't differ from those of the quadratic equation $y^2 = 1$ in $\mathbf{Z}$ or a Galois field GF(p) (i.e. $\mathbf{Z}_p$ with prime p). The last two systems have the non-trivial solutions $y_3 = a$, $y_4 = -a$, as postulated by the *Chinese remainder theorem* stating that a system of k simultaneous congruences (i.e. a system of equations of the form $y \equiv a_i \bmod m_i$) with coprime moduli $m_1, \ldots, m_k$ (i.e. $\gcd(m_i, m_j) = 1$ for all $i \neq j$) has a unique solution y with $0 \leq x < m_1 m_2 \ldots m_k$.

4.2.2.2 Finding a Factor

If r is even and $y = \pm a$ with $a \neq 1$ and $a \neq N-1$, then $(a+1)$ or $(a-1)$ must have a common divisor with N because $a^2 \equiv 1 \bmod N$ which means that $a^2 = cN + 1$ with $c \in \mathbf{N}$ and therefore $a^2 - 1 = (a+1)(a-1) = cN$. A factor of N can then be found by using *Euclid's algorithm* to determine $\gcd(N, a+1)$ and $\gcd(N, a-1)$ which is defined as

$$\gcd(a,b) = \begin{cases} b & \text{if } a \bmod b = 0 \\ \gcd(b, a \bmod b) & \text{if } a \bmod b \neq 0 \end{cases} \quad \text{with} \quad a > b \tag{4.21}$$

It can be shown that a random x matches the above mentioned conditions with a probability $p > \frac{1}{2}$ if N is not of the form $N = p^\alpha$ or $N = 2p^\alpha$. Since there are efficient classical algorithms to factorize pure prime powers (and of course to recognize a factor of 2), an efficient probabilistic algorithm for factorization can be found if the period r of the modular exponentiation can be determined in polynomial time.

4.2.2.3 Period of a Function

Let F be quantum function $F : |x, 0\rangle \to |x, f(x)\rangle$ of the integer function $f : \mathbf{Z} \to \mathbf{Z}_{2^m}$ with the unknown period $r < 2^n$.

To determine r, we need two registers, with the sizes of $2n$ and m qubits, which should be reset to $|0, 0\rangle$.

As a first step we produce a homogenous superposition of all base-vectors in the first register by applying an operator U with

$$U|0,0\rangle = \sum_{i=0}^{N-1} c_i |i, 0\rangle \quad \text{with} \quad |c_i| = \frac{1}{\sqrt{N}} \quad \text{and} \quad N = 2^{2n} \tag{4.22}$$

This can e.g. be achieved by the Hadamard transform H. Applying F to the resulting state gives

$$|\psi\rangle = F\,H\,|0,0\rangle = F\,\frac{1}{2^n} \sum_{i=0}^{N-1} |i, 0\rangle = \frac{1}{2^n} \sum_{i=0}^{N-1} |i, f(i)\rangle \tag{4.23}$$

A measurement of the second register with the result $k = f(s)$ with $s < r$ reduces the state to

$$|\psi'\rangle = \sum_{j=0}^{\lceil N/r \rceil - 1} c'_j |rj + s, k\rangle \quad \text{with} \quad c'_j = \left\lceil \frac{N}{r} \right\rceil^{-\frac{1}{2}} \tag{4.24}$$

The post-measurement state $|\psi'\rangle$ of the first register consists only of base-vectors of the form $|rj + s\rangle$ since $f(rj + s) = f(s)$ for all j. It therefore has a discrete, homogenous spectrum.

It is not possible to directly extract the period r or a multiple of it by measurement of the first register because of the random offset s. This problem can be solved by performing a discrete Fourier transform (see 4.2.3)

$$DFT : |x\rangle \to \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} \mathrm{e}^{\frac{2\pi \mathrm{i}}{N} xy} |y\rangle \tag{4.25}$$

on the register, as the probability spectrum of the transformed state is invariant to the offset (i.e. only the phases but not the absolute value of the complex amplitudes are effected).

$$|\tilde{\psi}'\rangle = DFT\,|\psi'\rangle = \sum_{i=0}^{N-1} \tilde{c}'_i |i, k\rangle \tag{4.26}$$

$$\tilde{c}'_i = \frac{\sqrt{r}}{N} \sum_{j=0}^{p-1} \exp\left(\frac{2\pi \mathrm{i}}{N} i(jr + s)\right) = \frac{\sqrt{r}}{N} \mathrm{e}^{\phi_i} \sum_{j=0}^{p-1} \exp\left(\frac{2\pi \mathrm{i}}{N} ijr\right) \tag{4.27}$$

$$\text{with} \quad \phi_i = 2\pi \mathrm{i} \frac{is}{N} \quad \text{and} \quad p = \left\lceil \frac{N}{r} \right\rceil$$

If $N = 2^{2n}$ is a multiple of r then $\tilde{c}'_i = \mathrm{e}^{\phi_i}/\sqrt{r}$ if i is a multiple of N/r and 0 otherwise. But even if r is not a power of 2, the spectrum of $|\tilde{\psi}'\rangle$ shows distinct peaks with a period of N/r because

$$\lim_{n \to \infty} \frac{1}{n} \sum_{k=0}^{n-1} \mathrm{e}^{2\pi \mathrm{i} k\alpha} = \begin{cases} 1 & \text{if } \alpha \in \mathbf{Z} \\ 0 & \text{if } \alpha \notin \mathbf{Z} \end{cases} \tag{4.28}$$

This is also the reason why we use a first register of $2n$ qubits when $r < 2^n$ because it guarantees at least 2^n elements in the above sum and thus a peak width of order $O(1)$.

If we now measure the first register, we will get a value c close to $\lambda N/r$ with $\lambda \in \mathbf{Z}_r$. This can be written as $c/N = c \cdot 2^{-2n} \approx \lambda/r$. We can think of this as finding a rational approximation a/b with $a, b < 2^n$ for the fixed point binary number $c \cdot 2^{-2n}$. An efficient classical algorithm for solving this problem using continued fractions is described in [16] and is implemented in the `denominator` function (appendix B.2).

Since the form of a rational number is not unique, λ and r are only determined by $a/b = \lambda/r$ if $\gcd(\lambda, r) = 1$. The probability that λ and r are coprime is greater then $1/\ln r$, so only $O(n)$ tries are necessary for a constant probability of success as close at 1 as desired.[1]

4.2.3 Quantum Fourier Transform

For a N dimensional vector $|\psi\rangle$, the discrete Fourier transform is defined as

$$DFT : |x\rangle \to \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} \mathrm{e}^{\frac{2\pi\mathrm{i}}{N}xy} |y\rangle \tag{4.29}$$

Since $|\psi\rangle$ is a combined state of n qubits, N is always a power of 2. The classical fast Fourier Transform (FFT) uses a binary decomposition of the exponent to perform the transformation in $O(n2^n)$ steps.

As suggested by Coppersmith [7], the same principle could adapted be to quantum computers by using a combination of Hadamard transformations H (see 3.4.4.3) and conditional phase gates V (see 3.4.4.4). The indices below indicate the qubits operated on:

$$DFT' = \prod_{i=1}^{n-1} \left(H_{n-i-1}(\frac{\pi}{2}) \prod_{j=0}^{i-1} V_{n-i-1,n-j-1}(\frac{2\pi}{2^{i-j+1}}) \right) H_{n-1} \tag{4.30}$$

DFT' iterates the qubits form the MSB to the LSB, "splits" the qubits with the Hadamard transformation and then conditionally applies phases according to their relative binary position $(\mathrm{e}^{\frac{2\pi\mathrm{i}}{2^{i-j+1}}})$ to each already split qubit.

The base-vectors of the transformed state $|\tilde{\psi}'\rangle = DFT' |\psi\rangle$ are given in reverse bit order, so to get the actual DFT, the bits have to be flipped.

```
operator dft(qureg q) { // main operator
  const n=#q;           // set n to length of input
  int i; int j;         // declare loop counters
  for i=0 to n-1 {
    for j=0 to i-1 {    // apply conditional phase gates
      CPhase(2*pi/2^(i-j+1),q[n-i-1] & q[n-j-1]);
    }
    Mix(q[n-i-1]);      // qubit rotation
  }
  flip(q);              // swap bit order of the output
}
```

[1] If the supposed period $r' = b$ derived form the rational approximation $a/b \approx c\,2^{-2m}$ is odd or $\gcd(x^{r'/2} \pm 1, N) = 1$, then one could try to expand a/b by some integer factor k in order to guess the actual period $r = kb$.

4.2.4 Modular Arithmetic

The most difficult part in implementing Shor's algorithm is the construction of an efficient quantum function for modular exponentiation.

$$\texttt{expn}_{a,n}(\mathbf{b},\mathbf{e}) : |b\rangle_{\mathbf{b}}|0\rangle_{\mathbf{e}} \to |b\rangle_{\mathbf{b}}|a^b \bmod n\rangle_{\mathbf{e}} \tag{4.31}$$

Assuming we already have an implementation for modular addition, we could use it to construct modular multiplication and finally exponentiation since

$$ab \bmod n = \sum_{i=0}^{\lceil \mathrm{ld}\, b\rceil} b_i \left(2^i a \bmod n\right) \quad \text{with} \quad b_i \in \mathbf{B} \tag{4.32}$$

$$a^b \bmod n = \prod_{i=0}^{\lceil \mathrm{ld}\, b\rceil} \left(a^{2^i b_i} \bmod n\right) \quad \text{with} \quad b_i \in \mathbf{B} \tag{4.33}$$

4.2.4.1 Modular Addition

The addition modulo n of a classic integer a and a quantum register $\mathbf{b}$ can result in either $a+b$ or $(a-n)+b)$, depending on the particular base-vector $|b\rangle$.

While for $b < n$ the operation is revertible, this is not the case for $b \geq n$, so, if n doesn't happen to be a power of 2, besides the target resister $\mathbf{y}_s$ for the sum, we need an additional flag-qubit $\mathbf{y}_y$ to allow for a quantum function `addn` which is both, unitary and invariant to $\mathbf{b}$:

$$\texttt{addn}_{a,n} : |b\rangle_{\mathbf{b}}|0\rangle_{\mathbf{y}_s}|0\rangle_{\mathbf{y}_f} \to \begin{cases} |b\rangle_{\mathbf{b}}|a+b\rangle_{\mathbf{y}_s}|1\rangle_{\mathbf{y}_{flag}} & \text{if } a+b<n \\ |b\rangle_{\mathbf{b}}|a+b-n\rangle_{\mathbf{y}_s}|0\rangle_{\mathbf{y}_{flag}} & \text{if } a+b \geq n \end{cases} \tag{4.34}$$

The actual implementation of `addn` can be found in appendix B.5.

Since $\texttt{addn}_{n-a,n}$ is a quantum function for modular subtraction and thus implements the inverse function $f_{a,n}^{-1}(b) = b - a \bmod n$ to $f_{a,n} = a + b \bmod n$, we can construct an overwriting version `oaddn` of modular addition, by using the method introduced in 3.5.2.3:

$$F' : |i,0\rangle \xrightarrow{U_f} |i, f(i)\rangle \xrightarrow{Swap} |f(i), i\rangle \xrightarrow{U^\dagger_{f^{-1}}} |f(i), 0\rangle \tag{4.35}$$

$\texttt{addn}_{n-a,n}$ doesn't invert the overflow flag $\mathbf{y}_f$, so we have to switch it manually:

$$U^\dagger_{f^{-1}} = \texttt{addn}_{n-a,n}(\mathbf{b}, \mathbf{y}_s, \mathbf{y}_f) \tag{4.36}$$

The original target registers $\mathbf{y}_s$ and $\mathbf{y}_f$ can now be allocated as unmanaged local scratch.

```
qufunct oaddn(int a,int n,qureg sum,quconst e) {
  qureg j[#sum];
  qureg f[1];

  addn(a,n,sum,f,j,e);              // junk -> a+b mod n
  Swap(sum,j);                      // swap junk and sum
  CNot(f,e);                        // toggle flag
  !addn(n-a,n,sum,f,j,e);           // uncompute b to zero
}
```

The register **e** is an enable register (see 2.2.2.6), so `addn` and `oaddn` are in fact conditional operators which only have an effect on eigenvectors of the form $|x\rangle|111\ldots\rangle_{\mathbf{e}}$.

4.2.4.2 Modular Multiplication

Modular multiplication is merely a composition of conditional additions for each qubit of **b**. The first summand can be slightly optimized, since the accumulator (`prod`) is still empty.

```
qufunct muln(int a,int n,quconst b,qureg prod,quconst e) {
  int i;

  for i=0 to #prod-1 {
    if bit(a,i) { CNot(prod[i],b[0] & e); }
  }
  for i=1 to #b-1 {
    oaddn(2^i*a mod n,n,prod,b[i] & e);
  }
}
```

As above, we can construct an overwriting version, if an implementation of the inverse function exists. This is the case if $\gcd(a,n)=1$ so a and n are relatively prime, because then the modular inverse a^{-1} with $a^{-1}a \bmod n = 1$ exists. Since we intend to use the operator for the Shor algorithm which demands that $\gcd(a^k,n)=1$, this is good enough for us.

By using two conditional XOR gates defined as

$$\texttt{cxor} : |a\rangle_{\mathbf{a}}|b\rangle_{\mathbf{b}}|\epsilon\rangle_{\mathbf{e}} \to \begin{cases} |a\rangle_{\mathbf{a}}|a\oplus b\rangle_{\mathbf{b}}|\epsilon\rangle_{\mathbf{e}} & \text{if } \epsilon = 111\ldots \\ |a\rangle_{\mathbf{a}}|b\rangle_{\mathbf{b}}|\epsilon\rangle_{\mathbf{e}} & \text{otherwise} \end{cases} \tag{4.37}$$

for swapping the registers[2] we can implement a conditional overwriting version of `muln` defined as

$$\texttt{omuln}_{[[\mathbf{e}]],a,n}|b\rangle \to |ab \bmod n\rangle \tag{4.38}$$

[2]normally, 3 XOR operations are necessary to swap a register, but since one register is empty, 2 XORs suffice.

```
qufunct omuln(int a,int n,qureg b,quconst e) {
  qureg j[#b];

  muln(a,n,b,j,e);
  !muln(invmod(a,n),n,j,b,e);
  cxor(j,b,e);
  cxor(b,j,e);
}
```

4.2.4.3 Modular Exponentiation

As with `muln`, we can construct modular exponentiation by conditionally applying `omuln` with the qubits of the exponents as enable string. Before we can start the iteration, the accumulator (`ex`) has to be initialized by 1.

```
qufunct expn(int a,int n,quconst b,quvoid ex) {
  int i;

  Not(ex[0]);                         // start with 1
  for i=0 to #b-1 {
    omuln(powmod(a,2^i,n),n,ex,b[i]);    // ex -> ex*a^2^i mod n
  }
}
```

4.2.5 Implementation

4.2.5.1 Auxiliary Functions

The implementation of the Shor algorithm uses the following functions:

- `boolean testprime(int n)`
  Tests whether n is a prime number
- `boolean testprimepower(int n)`
  Tests whether n is a prime power[3]
- `int powmod(int x,int a,int n)`
  Calculates $x^a \bmod n$
- `int denominator(real x,int qmax)`
  Returns the denominator q of the best rational approximation $\frac{p}{q} \approx x$ with $p, q < q_{max}$

For the actual implementations of these functions, please refer to appendix B.2.

[3] Since both testfunctions are not part of the algorithm itself, short but inefficient implementations with $O(\sqrt{n})$ have been used

4.2.5.2 The Procedure shor

The procedure `shor` checks whether the integer `number` is suitable for quantum factorization, and then repeats Shor's algorithm until a factor has been found.

```
procedure shor(int number) {
  int width=ceil(log(number,2));   // size of number in bits
  qureg reg1[2*width];             // first register
  qureg reg2[width];               // second register
  int qmax=2^width;
  int factor;                      // found factor
  int m; real c;                   // measured value
  int x;                           // base of exponentiation
  int p; int q;                    // rational approximation p/q
  int a; int b;                    // possible factors of number
  int e;                           // e=x^(q/2) mod number

  if number mod 2 == 0 { exit "number must be odd"; }
  if testprime(number) { exit "prime number"; }
  if testprimepower(number) { exit "prime power"; };

  {
    {                              // generate random base
      x=floor(random()*(number-3))+2;
    } until gcd(x,number)==1;
    print "chosen random x =",x;
    Mix(reg1);                     // Hadamard transform
    expn(x,number,reg1,reg2);      // modular exponentiation
    measure reg2;                  // measure 2nd register
    dft(reg1);                     // Fourier transform
    measure reg1,m;                // measure 1st register
    reset;                         // clear local registers
```

```
    if m==0 {                       // failed if measured 0
      print "measured zero in 1st register. trying again ...";
    } else {
      c=m*0.5^(2*width);            // fixed point form of m
      q=denominator(c,qmax);        // find rational approximation
      p=floor(q*m*c+0.5);
      print "measured",m,", approximation for",c,"is",p,"/",q;
      if q mod 2==1 and 2*q<qmax { // odd q ? try expanding p/q
        print "odd denominator, expanding by 2";
        p=2*p; q=2*q;
      }
      if q mod 2==1 {               // failed if odd q
        print "odd period. trying again ...";
      } else {
        print "possible period is",q;
        e=powmod(x,q/2,number);     // calculate candidates for
        a=(e+1) mod number;         // possible common factors
        b=(e+number-1) mod number; // with number
        print x,"^",q/2,"+ 1 mod",number,"=",a,",",
              x,"^",q/2,"- 1 mod",number,"=",b;
        factor=max(gcd(number,a),gcd(number,b));
      }
    }
  } until factor>1 and factor<number;
  print number,"=",factor,"*",number/factor;
}
```

4.2.5.3 Factoring 15

15 is the smallest number that can be factorized with Shor's algorithm, as it's the product of the smallest odd prime numbers 3 and 5. Our implementation of the modular exponentiation needs $2l + 1$ qubits scratch space with $l = \lceil \mathrm{ld}(15+1) \rceil = 4$. The algorithm itself needs $3l$ qubits, so a total of 21 qubits must be provided.

```
$ qcl -b21 -i shor.qcl
qcl> shor(15)
: chosen random x = 4
: measured zero in 1st register. trying again ...
: chosen random x = 11
: measured 128 , approximation for 0.500000 is 1 / 2
: possible period is 2
: 11 ^ 1 + 1 mod 15 = 12 , 11 ^ 1 - 1 mod 15 = 10
: 15 = 5 * 3
```

The first try failed because 0 was measured in the first register of $|\psi'\rangle$ and $\lambda/r = 0$ gives no information about the period r.

One might argue that this is not likely to happen, since the first register has 8 qubits and 256 possible base-vectors, however, if a number n is to be factored, one might expect a period about $\sqrt{n}$ assuming that the prime factors of n are of the same order of magnitude. This would lead to a period $\frac{q}{\sqrt{n}}$ after the DFT and the probability $p = \frac{1}{\sqrt{n}}$ to accidentally pick the basevector $|0\rangle$, would be $p = 25.8\%$.

In the special case of a start value $x = 4$ the period of the modular exponentiation is 2 since $4^2 \bmod 15 = 1$, consequently the Fourier spectrum shows 2 peaks at $|0\rangle$ and $|128\rangle$ and $p = 1/2$.

The second try also had the same probability of failure since $11^2 \bmod 15 = 1$, but this time, the measurement picked the second peak in the spectrum at $|128\rangle$. With $128/2^8 = 1/2 = \lambda/r$, the period $r = 2$ was correctly identified and the factors $\gcd(11^{2/2} \pm 1\,,\, 15) = \{3, 5\}$ to 15 have been found.

Bibliography

[1] Paul Benioff *1997 Models of Quantum Turing Machines, LANL Archive quant-ph/9708054*

[2] J.I. Cirac, P. Zoller *1995 Quantum Computations with Cold trapped Ions, Phys. Rev. Lett. 74, 1995 , 4091*

[3] D. Deutsch, *1985 Quantum theory, the Church-Turing principle and the universal quantum computer. Proceedings of the Royal Society London A 400, 97-117*

[4] J. Gruska, *1998 Foundations of Computing, chap. 12: "Frontiers - Quantum Computing"*

[5] R. W. Keyes *1988 IBM J. Res. Develop. 32, 24*

[6] D. Deutsch *1989 Quantum computational networks. Proceedings of the Royal Society London A 439, 553-558*

[7] D. Coppersmith *1994 An Approximate Fourier Transform Useful in Quantum Factoring, IBM Research Report No. RC19642*

[8] C. H. Bennet *1973 Logical Reversibility of Computation. IBM J. Res. Develop. 17, 525*

[9] C. H. Bennet *1989 SIAM J.Comput. 18, 766*

[10] Johannes Buchmann *1996 Faktorisierung großer Zahlen. Spektrum der Wissenschaft 9/96, 80-88*

[11] P.W. Shor. *1994 Algorithms for quantum computation: Discrete logarithms and factoring*

[12] Samuel L. Braunstein *1995 Quantum computation: a tutorial*

[13] David Beckman et al. *1996 Efficient networks for quantum factoring*

[14] F.D. Murnaghan *1962 The Unitary and Rotation Groups, Spartan Books, Washington*

[15] Artur Ekert and Richard Jozsa. *1996 Shor's Quantum Algorithm for Factoring Numbers, Rev. Modern Physics 68 (3), 733-753*

[16] G.H. Hardy and E.M. Wright *1965 An Introduction to the Theory of Numbers (4th edition OUP)*

[17] B. Jack Copeland *1996, The Church-Turing Thesis. Stanford Encyclopedia of Philosophy ISSN 1095-5054*

[18] E.L. Post *1936. 'Finite Combinatory Processes - Formulation 1'. Journal of Symbolic Logic, 1, 103-105.*

[19] A.M. Turing *1948 Intelligent Machinery. National Physical Laboratory Report. In Meltzer, B., Michie, D. (eds) 1969. Machine Intelligence 5. Edinburgh: Edinburgh University Press., 7*

[20] Lov K. Grover *1996 A fast quantum mechanical algorithm for database search. Proceeding of the 28th Annual ACM Symposium on Theory of Computing*

[21] Michel Boyer, Gilles Brassard, Peter Hoyer, Alain Tapp *1996 Tight bounds on quantum searching. Proceedings PhysComp96*

[22] Hilary Putnam *1965 A philosopher looks at quantum mechanics*

[23] W. Kummer and R. Trausmuth *1988 Skriptum zur Vorlesung 131.869 - Quantentheorie*

[24] Bernhard Ömer *1996 Simulation of Quantum Computers [unpublished]*

[25] Bernhard Ömer *1998 A Procedural Formalism for Quantum Computing, master-thesis, Technical University of Vienna*

List of Figures

List of Tables

Appendix A

QCL Syntax

A.1 Expressions

$$
\begin{array}{rcl}
\textit{complex-coord} & \leftarrow & [\,\texttt{+}\,|\,\texttt{-}\,]\ \textit{digit}\ \{\,\textit{digit}\,\}\ [\ \texttt{.}\ \{\,\textit{digit}\,\}] \\
\textit{const} & \leftarrow & \textit{digit}\ \{\,\textit{digit}\,\}\ [\ \texttt{.}\ \{\,\textit{digit}\,\}] \\
 & \leftarrow & \texttt{(}\ \textit{complex-coord}\ \texttt{,}\ \textit{complex-coord}\ \texttt{)} \\
 & \leftarrow & \texttt{true}\,|\,\texttt{false} \\
 & \leftarrow & \texttt{"}\ \{\,\textit{char}\,\}\ \texttt{"} \\
\textit{expr} & \leftarrow & \textit{const} \\
 & \leftarrow & \textit{identifier}\ [\ \texttt{[}\ \textit{expr}\ [(\ \texttt{:}\ |\ \texttt{..}\ )\ \textit{expr}]\ \texttt{]}\ ] \\
 & \leftarrow & \textit{identifier}\ \texttt{(}\ [\,\textit{expr}\ \{\ \texttt{,}\ \textit{expr}\,\}]\ \texttt{)} \\
 & \leftarrow & \texttt{(}\ \textit{expr}\ \texttt{)} \\
 & \leftarrow & \texttt{\#}\ \textit{expr} \\
 & \leftarrow & \textit{expr}\,\texttt{\^{}}\ \textit{expr} \\
 & \leftarrow & \texttt{-}\ \textit{expr} \\
 & \leftarrow & \textit{expr}\,(\,\texttt{*}\,|\,\texttt{/}\,)\,\textit{expr} \\
 & \leftarrow & \textit{expr}\ \texttt{mod}\ \textit{expr} \\
 & \leftarrow & \textit{expr}\,(\,\texttt{+}\,|\,\texttt{-}\,|\,\texttt{\&}\,)\,\textit{expr} \\
 & \leftarrow & \textit{expr}\,(\,\texttt{==}\,|\,\texttt{!=}\,|\,\texttt{<}\,|\,\texttt{<=}\,|\,\texttt{>}\,|\,\texttt{>=}\,)\,\textit{expr} \\
 & \leftarrow & \texttt{not}\ \textit{expr} \\
 & \leftarrow & \textit{expr}\ \texttt{and}\ \textit{expr} \\
 & \leftarrow & \textit{expr}\,(\,\texttt{or}\,|\,\texttt{xor}\,)\,\textit{expr}
\end{array}
$$

A.2 Statements

block ← `{` *stmt* { *stmt* } `}`

option ← *letter* { *letter* | `-` }

stmt ← [`!`] *identifier* `(` [*expr* { `,` *expr* }] `)` `;`

← *identifier* `=` *expr* `;`

← *expr* (`->` | `<-` | `<->`) *expr* `;`

← `for` *identifier* `=` *expr* `to` *expr* [`step` *expr*] *block*

← `while` *expr* *block*

← *block* `until` *expr* `;`

← `if` *expr* *block* [`else` *block*]

← `return` *expr* `;`

← `input` [*expr*] `,` *identifier* `;`

← `print` *expr* [`,` *expr*] `;`

← `exit` [*expr*] `;`

← `measure` *expr* [`,` *identifier*] `;`

← `reset` `;`

← `dump` [*expr*] `;`

← `list` [*identifier* { `,` *identifier* }] `;`

← (`load` | `save`) [*expr*] `;`

← `shell` `;`

← `set` *option* [`,` *expr*] `;`

← *stmt* `;`

A.3 Definitions

type ← `int` | `real` | `complex` | `string`

← `qureg` | `quvoid` | `quconst` | `quscratch`

const-def ← `const` *identifier* `=` *expr* `;`

var-def ← *type* *identifier* `[` *expr* `]` `;`

← *type* *identifier* [`=` *expr*] `;`

arg-def ← *type* *identifier*

arg-list ← `(` [*arg-def* { `,` *arg-def* }] `)`

body ← `{` { *const-def* | *var-def* } { *stmt* } `}`

def ← *const-def* | *var-def*

$\leftarrow$ *type identifier arg-list body*
$\leftarrow$ `procedure` *identifier arg-list body*
$\leftarrow$ `operator` *identifier arg-list body*
$\leftarrow$ `qufunct` *identifier arg-list body*
$\leftarrow$ `extern operator` *identifier arg-list* `;`
$\leftarrow$ `extern qufunct` *identifier arg-list* `;`

Appendix B

The Shor Algorithm in QCL

B.1 default.qcl

```
extern qufunct Fanout(quconst a,quvoid b);

extern qufunct Swap(qureg a,qureg b);

extern operator Matrix2x2(
  complex u00,complex u01,
  complex u10,complex u11,
qureg q);

extern operator Matrix4x4(
  complex u00,complex u01,complex u02,complex u03,
  complex u10,complex u11,complex u12,complex u13,
  complex u20,complex u21,complex u22,complex u23,
  complex u30,complex u31,complex u32,complex u33,
qureg q);

extern operator Matrix8x8(
  complex u00,complex u01,complex u02,complex u03,
  complex u04,complex u05,complex u06,complex u07,
  complex u10,complex u11,complex u12,complex u13,
  complex u14,complex u15,complex u16,complex u17,
  complex u20,complex u21,complex u22,complex u23,
  complex u24,complex u25,complex u26,complex u27,
  complex u30,complex u31,complex u32,complex u33,
  complex u34,complex u35,complex u36,complex u37,
  complex u40,complex u41,complex u42,complex u43,
  complex u44,complex u45,complex u46,complex u47,
  complex u50,complex u51,complex u52,complex u53,
  complex u54,complex u55,complex u56,complex u57,
  complex u60,complex u61,complex u62,complex u63,
```

```
  complex u64,complex u65,complex u66,complex u67,
  complex u70,complex u71,complex u72,complex u73,
  complex u74,complex u75,complex u76,complex u77,
qureg q);

extern qufunct Perm2(int p0 ,int p1 ,qureg q);

extern qufunct Perm4(int p0 ,int p1 ,int p2 ,int p3 ,qureg q);

extern qufunct Perm8(
  int p0 ,int p1 ,int p2 ,int p3 ,int p4 ,int p5 ,int p6 ,int p7 ,
qureg q);

extern qufunct Perm16(
  int p0 ,int p1 ,int p2 ,int p3 ,int p4 ,int p5 ,int p6 ,int p7 ,
  int p8 ,int p9 ,int p10,int p11,int p12,int p13,int p14,int p15,
qureg q);

extern qufunct Perm32(
  int p0 ,int p1 ,int p2 ,int p3 ,int p4 ,int p5 ,int p6 ,int p7 ,
  int p8 ,int p9 ,int p10,int p11,int p12,int p13,int p14,int p15,
  int p16,int p17,int p18,int p19,int p20,int p21,int p22,int p23,
  int p24,int p25,int p26,int p27,int p28,int p29,int p30,int p31,
qureg q);

extern qufunct Perm64(
  int p0 ,int p1 ,int p2 ,int p3 ,int p4 ,int p5 ,int p6 ,int p7 ,
  int p8 ,int p9 ,int p10,int p11,int p12,int p13,int p14,int p15,
  int p16,int p17,int p18,int p19,int p20,int p21,int p22,int p23,
  int p24,int p25,int p26,int p27,int p28,int p29,int p30,int p31,
  int p32,int p33,int p34,int p35,int p36,int p37,int p38,int p39,
  int p40,int p41,int p42,int p43,int p44,int p45,int p46,int p47,
  int p48,int p49,int p50,int p51,int p52,int p53,int p54,int p55,
  int p56,int p57,int p58,int p59,int p60,int p61,int p62,int p63,
qureg q);

extern qufunct Not(qureg q);

extern qufunct CNot(qureg q,quconst c);

extern operator CPhase(real phi,qureg q);

extern operator Rot(real theta,qureg q);

extern operator Mix(qureg q);

extern qufunct ModExp(int n,int x,quconst a,quvoid b);

boolean bit(int n,int b) {
```

```
  return n/2^b mod 2 == 1;
}

qufunct set(int n,qureg q) {
  int i;
  for i=0 to #q-1 {
    if bit(n,i) { Not(q[i]); }
  }
}

const pi=3.141592653589793238462643383279502884197;
```

B.2 functions.qcl

```
set allow-redefines 1;

// returns the smallest factor > 1 of n or 1 if n is prime

int findfactor(int n) {
  int i;
  if n<=0 { exit "findfactor takes only positive args"; }
  for i=2 to floor(sqrt(n)) {
    if n mod i == 0 { return i; }
  }
  return 1;
}

// test if n is a prime number

boolean testprime(int n) {
  int i;
  if n<=1 { return false; }
  for i=2 to floor(sqrt(n)) {
    if n mod i == 0 { return false; }
  }
  return true;
}

// test if n is a prime power

boolean testprimepower(int n) {
  int i;
  int f;
  i=2;
  while i<=floor(sqrt(n)) and f==0 {
    if n mod i == 0 { f=i; }
    i=i+1;
  }
```

```
  for i=2 to floor(log(n,f)) {
    if f^i==n { return true; }
  }
  return false;
}

// returns x^a mod n

int powmod(int x,int a,int n) {
  int u=x;
  int y=1;
  int i;

  for i=0 to 30 {
    if a/2^i mod 2 == 1 { y=y*u mod n; }
    u=u^2 mod n;
  }
  return y;
}

// return the modular inverse to a mod n or 0 if gcd(a,n)>1

int invmod(int a,int n) {
  int b=a;
  int i;

  if gcd(a,n)>1 { return 0; }
  for i=1 to n {
    if b*a mod n == 1 { return b; }
    b=b*a mod n;
  }
  return 0;
}

// finds the denominator q of the best rational approximation p/q
// for x with q<qmax

int denominator(real x,int qmax) {
  real y=x;
  real z;
  int q0;
  int q1=1;
  int q2;

  while true {
    z=y-floor(y);
    if z<0.5/qmax^2 { return q1; }
    y=1/z;
    q2=floor(y)*q1+q0;
```

```
    if q2>=qmax { return q1; }
    q0=q1; q1=q2;
  }
}

set allow-redefines 0;
```

B.3 qufunct.qcl

```
set allow-redefines 1;

// pseudo classic operator to swap bit order

qufunct flip(qureg q) {
  int i;                // declare loop counter
  for i=0 to #q/2-1 {   // swap 2 symmetric bits
    Swap(q[i],q[#q-i-1]);
  }
}

// Conditional Xor

qufunct cxor(quconst a,qureg b,quconst e) {
  int i;
  for i=0 to #a-1 {
    CNot(b[i],a[i] & e);
  }
}

// Conditional multiplexed binary adder for one of 2 classical
// bits and 1 qubit.
// Full adder if #sum=2, half adder if #sum=1.

qufunct muxaddbit(boolean a0,boolean a1,quconst sel,
                  quconst b,qureg sum,quconst e) {
  qureg s=sel;                          // redeclare sel as qureg

  if (a0 xor a1) {                      // a0 and a1 differ?
    if a0  { Not(s); }                  // write a into sect qubit
    if #sum>1 {                         // set carry if available
      CNot(sum[1],sum[0] & s & e);
    }
    CNot(sum[0],s & e);                 // add a
    if a0  { Not(s); }                  // restore sect qubit
  } else {
    if a0 and a1 {
      if #sum>1 {                       // set carry if available
        CNot(sum[1],sum[0] & e);
```

```
      }
      CNot(sum[0],e);                  // add a
    }
  };
                                       // Add qubit b
  if #sum>1 {                          // set carry if available
    CNot(sum[1],b & sum[0]);
  }
  CNot(sum[0],b);                      // add b
}

// conditional multiplexed binary adder for one of 2 integers
// and 1 qureg. No output carry.

qufunct muxadd(int a0,int a1,qureg sel,quconst b,quvoid sum,quconst e) {
  int i;
  for i=0 to #b-2 {                    // fulladd first #b-1 bits
    muxaddbit(bit(a0,i),bit(a1,i),sel,b[i],sum[i:i+1],e);
  }
                                       // half add last bit
  muxaddbit(bit(a0,#b-1),bit(a1,#b-1),sel,b[#b-1],sum[#b-1],e);
}

// Comparison operator. flag is toggled if b<a.
// b gets overwritten. Needs a #b-1 qubit junk register j
// as argument which is left dirty.

qufunct lt(int a,qureg b,qureg flag,quvoid j) {
  int i;
  if bit(a,#b-1) {                 // disable further comparison
    CNot(j[#b-2],b[#b-1]);         // and set result flag if
    Not(b[#b-1]);                  // MSB(a)>MSB(b)
    CNot(flag,b[#b-1]);
  } else {
    Not(b[#b-1]);                  // disable further comparison
    CNot(j[#b-2],b[#b-1]);         // if MSB(a)<MSB(b)
  }
  for i=#b-2 to 1 step -1 {        // continue for lower bits
    if bit(a,i) {                  // set new junk bit if undecided
      CNot(j[i-1],j[i] & b[i]);
      Not(b[i]);                   // honor last junk bit and
      CNot(flag,j[i] & b[i]);      // set result flag if a[i]>b[i]
    } else {
      Not(b[i]);
      CNot(j[i-1],j[i] & b[i]);
    }
  }
  if bit(a,0) {
    Not(b[0]);                     // if still undecided (j[0]=1)
```

```
    CNot(flag,j[0] & b[0]);      // result is LSB(a)>LSB(b)
  }
}

set allow-redefines 0;
```

B.4 dft.qcl

```
operator dft(qureg q) { // main operator
  const n=#q;           // set n to length of input
  int i; int j;         // declare loop counters
  for i=0 to n-1 {
    for j=0 to i-1 {    // apply conditional phase gates
      CPhase(2*pi/2^(i-j+1),q[n-i-1] & q[n-j-1]);
    }
    Mix(q[n-i-1]);      // qubit rotation
  }
  flip(q);              // swap bit order of the output
}
```

B.5 modarith.qcl

```
set allow-redefines 1;

include "functions.qcl";
include "qufunct.qcl";

// conditional addition mod n for 1 integer and 1 qureg
// flag is set if a+b<n for invertability

qufunct addn(int a,int n,quconst b,quvoid flag,quvoid sum,quconst e) {
  qureg s=sum[0\#b-1];
  qureg f=sum[#b-1];
  qureg bb=b;                     // "abuse" sum and b as scratch
  lt(n-a,bb,f,s);                 // for the less-than operator
  CNot(flag,f & e);               // save result of comparison
  !lt(n-a,bb,f,s);                // restore sum and b
  muxadd(2^#b+a-n,a,flag,b,sum,e); // add either a or a-n
}

// Conditional overwriting addition mod n: sum -> (a+sum) mod n

qufunct oaddn(int a,int n,qureg sum,quconst e) {
  qureg j[#sum];
  qureg f[1];

  addn(a,n,sum,f,j,e);            // junk -> a+b mod n
```

```
  Swap(sum,j);                    // swap junk and sum
  CNot(f,e);                      // toggle flag
  !addn(n-a,n,sum,f,j,e);         // uncompute b to zero
}

// Conditional Multiplication mod n of an integer a by the qureg b,
// prod <- ab mod n.

qufunct muln(int a,int n,quconst b,qureg prod,quconst e) {
  int i;

  for i=0 to #prod-1 {
    if bit(a,i) { CNot(prod[i],b[0] & e); }
  }
  for i=1 to #b-1 {
    oaddn(2^i*a mod n,n,prod,b[i] & e);
  }
}

// Conditional Overwriting multiplication mod n: b-> ab mod n

qufunct omuln(int a,int n,qureg b,quconst e) {
  qureg j[#b];

  if gcd(a,n)>1 {
    exit "omuln: a and n have to be relativly prime";
  }
  muln(a,n,b,j,e);
  !muln(invmod(a,n),n,j,b,e);
  cxor(j,b,e);
  cxor(b,j,e);
}

// Modular exponentiation: b -> x^a mod n

qufunct expn(int a,int n,quconst b,quvoid ex) {
  int i;

  Not(ex[0]);                           // start with 1
  for i=0 to #b-1 {
    omuln(powmod(a,2^i,n),n,ex,b[i]);   // ex -> ex*a^2^i mod n
  }
}

set allow-redefines 0;
```

B.6 shor.qcl

```
include "modarith.qcl";
include "dft.qcl";

procedure shor(int number) {
  int width=ceil(log(number,2));   // size of number in bits
  qureg reg1[2*width];             // first register
  qureg reg2[width];               // second register
  int qmax=2^width;
  int factor;                      // found factor
  int m; real c;                   // measured value
  int x;                           // base of exponentiation
  int p; int q;                    // rational approximation p/q
  int a; int b;                    // possible factors of number
  int e;                           // e=x^(q/2) mod number

  if number mod 2 == 0 { exit "number must be odd"; }
  if testprime(number) { exit "prime number"; }
  if testprimepower(number) { exit "prime power"; };

  {
    {                              // generate random base
      x=floor(random()*(number-3))+2;
    } until gcd(x,number)==1;
    print "chosen random x =",x;
    Mix(reg1);                     // Hadamard transform
    expn(x,number,reg1,reg2);      // modular exponentiation
    measure reg2;                  // measure 2nd register
    dft(reg1);                     // Fourier transform
    measure reg1,m;                // measure 2st register
    reset;                         // clear local registers
    if m==0 {                      // failed if measured 0
      print "measured zero in 1st register. trying again ...";
    } else {
      c=m*0.5^(2*width);           // fixed point form of m
      q=denominator(c,qmax);       // find rational approximation
      p=floor(q*c+0.5);
      print "measured",m,", approximation for",c,"is",p,"/",q;
      if q mod 2==1 and 2*q<qmax { // odd q ? try expanding p/q
        print "odd denominator, expanding by 2";
        p=2*p; q=2*q;
      }
      if q mod 2==1 {              // failed if odd q
        print "odd period. trying again ...";
      } else {
        print "possible period is",q;
        e=powmod(x,q/2,number);    // calculate candidates for
        a=(e+1) mod number;        // possible common factors
```

```
        b=(e+number-1) mod number; // with number
        print x,"^",q/2,"+ 1 mod",number,"=",a,",",
              x,"^",q/2,"- 1 mod",number,"=",b;
        factor=max(gcd(number,a),gcd(number,b));
      }
    }
  } until factor>1 and factor<number;
  print number,"=",factor,"*",number/factor;
}
```