Simulation of Quantum Computers

Bernhard Ömer

4th October 1996

E-mail: `oemer@tph.tuwien.ac.at`

Supervisor: Doz. Dr. Karl Svozil

Department of Theoretical Physics

Technical University of Vienna

Abstract

The steady process of computer miniaturisation will soon come to a scale where quantum effects on computation can no longer be ignored. Hardware development will finally reach a point where boolean logic will no longer be applicable and the classical concept of a universal deterministic computer with the Turing machine as mathematical model will have to be replaced by a quantum theory of computation.

The direct encoding of information as quantum states forces the programmer to deal with new restrictions (e.g. the limitation to unitary operators) but also provides massive parallelism virtually for free, which can be used to efficiently solve problems (e.g. prime factorisation) for which no efficient classical algorithm is known.

This paper gives a general introduction to quantum computing and deals with the problems of simulating quantum computers on classical hardware. As an example, a simulation of Shor's factorisation algorithm is presented.

Contents

1 Introduction

1.1 Computation and Computers

1.1.1 Models of Computation

From an abstract point of view, computation is a process of manipulating a finite set of symbols (data) by applying a series of formal transformations (program). The initial set is called the input, the result of the final transformation the output of the program.

A computer is a physical device, which is able to carry out certain types of operations, which are not only determined by the limitations of the employed hardware, but primarily by the concept of computation used to construct the machine itself and to interpret its results.

The theoretic concept behind nowadays computers is that of the abstract *universal computer* whose most popular representative is the Turing machine named after Alan Turing, one of the pioneers of modern computer science. It can be shown that all deterministic abstract machines with unlimited memory capacity and a minimal set of basic instructions for reading, writing and conditional branching are equivalent in the sense, that every machine can be programmed to simulate and thus execute the programs of any machine of the class including itself. All functions, which can be computed on a Turing machine are called partial recursive or Turing computable.

However it is necessary to stress, that the universal computer is by no means the only applicable concept of computation and that many problems can be (and are actually) solved by using less powerful models like cellular automatons. Anyway, according to Church's theorem, any function which can be described by an algorithm or calculated by any mechanical process is partial recursive, and in this sense the Turing machine is in fact universal.

1.1.2 Computers as Physical Devices

To implement a computational model in a physical device, this computer must be able to adept different internal states and provide means to perform the necessary transformations on them and to extract the output information. The correlation between the physical and the logical state of the machine is arbitrary (as long it is consistent with the desired transformations) and requires interpretation.

In an ordinary RAM module, the common quantum state of thousands of electrons is interpreted as only one bit, thus either as 0 or 1. This abstraction is possible, because the great number of particles statistically washes away the principal uncertainty of measurement inherent to any quantum system. This allows us to implement the concept of a deterministic universal computer in non deterministic hardware.

However, with some problems (e.g. testing for prime numbers), indeterministic behaviour can drastically reduce the average number of necessary computational steps. Algorithms which contain random elements (e.g. Monte Carlo method for numerical integration) are called probabilistic. The computational concept of a (classical) probabilistic algorithm is that of a Turing machine which can "throw coins" i.e. can make random decisions, which of two (or more) computational paths to follow. A random Turing machine can also simulate any quantum system to arbitrary precision.

1.1.3 Limitations of the Classical Concept

The development of integrated circuits during the last decades shows a strong trend toward miniaturisation reducing the number of electrons representing one bit by a factor of 100 every ten years [1]. An extrapolation of this trend suggests, that an atomic scale might be reached within the next two decades, where quantum effects on register measurements can no longer be ignored.

The developers will be forced to either accept this limitation or to drastically alter their concept of computation by creating computers which rely on quantum effects rather than trying to avoid them.

1.2 Classical vs. Quantum Computers

This section introduces the most basic differences between classical and quantum computers in a phenomenologic manner. For a more rigid and formal explanation, please refer to section 2.

1.2.1 Quantum Bits

In a classical computer, the logical state is determined by the expectation value of its register contents (e.g. tension of a capacitor). The interpretation as (classical) bits is performed by comparing the measured value to a defined threshold, while the great number of particles guarantees that the uncertainty of the measurement is small enough to make errors practically impossible.

In a quantum computer, information is represented as the common quantum state of many subsystems. Each subsystem is described by a combination of two "pure" states interpreted as $|0\rangle$ and $|1\rangle$ (quantum bit, qubit). This can e.g. be realised by the spin of a particle, the polarisation of a photon or by the ground state and an excited state of an ion.

For a single qubit, this state can be described by the complex amplitudes a and b of each of the two states ($a|0\rangle + b|1\rangle$) with the condition $aa^* + bb^* = 1$.

It is obvious, that this interpretation stands in contradiction to classic boolean logic, where intermediate states between 0 and 1 are not possible.

1.2.2 Entanglement of States

The logical state of a classical register is determined by the states of all bits this register contains. Those bits can be changed locally i.e. independently form one another. The state of an n bit register, can therefore be described by n binary values.

A quantum register containing more than one qubit can not be described by simply listing the states of each qubit, moreover it is not even possible to define the state of an isolated qubit:

Given an isolated system of two qubits, its state can be described by four complex amplitudes $a|0,0\rangle + b|1,0\rangle + c|0,1\rangle + d|1,1\rangle$. You can define the expectation value for the first qubit, which is $\sqrt{bb^* + dd^*}$ but there is no isolated state for the first qubit anymore like e.g. $(a+c)|0\rangle + (b+d)|1\rangle$ since $|a|^2 + |b|^2 + |c|^2 + |d|^2 = 1$ does not implicate that $|a+c|^2 + |b+d|^2 = 1$.

Therefore, manipulations on a single qubit effect the complex amplitudes of the overall state and have a global character. To describe the combined state of n entangled qubits, 2^n complex numbers are necessary.

1.2.3 Measurement

In a classical computer, the formal description of the inner state and the measurement of this state (i.e. the output of the program) is the same and given by the binary values of the concerned bits. Moreover, the inner state is not effected by the process of measurement (non destructive measurement).

According to the Kopenhagner interpretation of quantum physics, the outcome of measurements on quantum systems (qubits) must be formulated in classical terms (binary bits). The quantum state of the system is thereby reduced: If the first bit in the above mentioned 2 qubit state is measured, and a value of 1 is observed, then the state will be reduced to $b'|1,0\rangle + d'|1,1\rangle$ with $|b'|^2 + |d'|^2 = 1$, thus all basevectors which 0 in the first bit ($|0,0\rangle$ and $|0,1\rangle$) will be set to an amplitude of zero.

Therefore it is principally not possible to measure the state of a quantum register itself; it is however possible, to estimate the expectation value of a qubit by repeated measurements after the same calculation.

1.2.4 Reversibility of Computation

Heat dissipation is one of the major problems with the miniaturisation of classical computers and constant cooling of all components is required. This is achieved by the thermic coupling of the circuits to a heat reservoir like e.g. the surrounding air.

For a quantum computer, cooling by heat coupling is no option since its logical state is directly represented by the common quantum state of its registers. Any heat coupling would necessarily result in the entanglement of this state with the outside world and destroy the coherence of the computation.

The second law of thermodynamics postulates that any non-reversible state change of a system must dissipate heat. Many common logical operations like AND, OR or resetting a bit to 0 or 1 are non-reversible is the sense that the input cannot be calculated from the output. Therefore, these operations cannot directly be implemented in a quantum computer.

2 Principles of Quantum Computation

2.1 Quantum States and Operators

2.1.1 The Hilbert Space

The state of a quantum computer with n qubits is a point in a 2^n-dimensional Hilbert space $\mathcal{H} = \mathbf{C}^{2^n}$. The theoretical storage capacity therefore increases exponentially with the number of qubits.

Any computational step can be described as an operator $O : |\psi\rangle \to |\phi\rangle$ over $\mathcal{H}$ or a subspace of $\mathcal{H}$ which transforms the input state $|\psi\rangle$ to the output state $|\phi\rangle = |O\,\psi\rangle$.

2.1.2 Unitary Operators

As pointed out in (section 1.2.4), quantum computers can only perform reversible operations. Every reversible operation can be described by a *unitary* operator U which matches the condition $U^{-1} = U^\dagger$. Compositions of unitary operators are also unitary since $(UV)^{-1} = V^\dagger U^\dagger$.

A general unitary transformation in the two dimensional Hilbert space $\mathbf{C}^2$ can be defined as follows:

$$U2(\theta, \delta, \sigma, \tau) = \begin{pmatrix} e^{i(\delta+\sigma+\tau)}\cos(\frac{\theta}{2}) & e^{-i(\delta+\sigma-\tau)}\sin(\frac{\theta}{2}) \\ -e^{i(\delta-\sigma+\tau)}\sin(\frac{\theta}{2}) & e^{i(\delta-\sigma-\tau)}\cos(\frac{\theta}{2}) \end{pmatrix} \quad \text{with} \quad \theta, \delta, \sigma, \tau \in \mathbf{R}$$

If this operator can be applied to arbitrary 2-dimensional subspaces of $\mathcal{H}$, than any unitary transformation can be constructed by composition. If only subspaces corresponding to a subset of qubits are allowed, which is the case for many proposed architectures, among them also the linear ion trap (*Cirac-Zoller device*), then an additional 4-dimensional 2-qubit operator is needed to obtain a mixing between separate qubits [2].

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

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

A quantum computer which is capable of performing *U2* and *XOR* operations can therfore perform any possible operation and is in this sense universal.

2.2 Input and Output

2.2.1 Initial State

To set a quantum computer to the desired input state $|\psi\rangle$, it suffices to provide means to initially "cool" all qubits to $|0\rangle$ and then apply a unitary transformation U which matches the condition $U|0\rangle = |\psi\rangle$. One might think of U as a base transformation which trivially exists for any desired $|\psi\rangle$.

2.2.2 Measuring States

Measuring n qubits reduces the dimensionality of $\mathcal{H}$ by a factor of 2^n. The outcome of the measurement is biased by the probability amplitude for a certain bit configuration.

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

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

The basevectors $|i,j\rangle$ are interpreted as a pair of binary numbers with $i < 2^n$ and $j < 2^m$. 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

$$p(I) = \sum_{j=0}^{2^m-1} c_{I,j} c^*_{I,j}, \quad \text{and} \quad |\psi'_I\rangle = \frac{1}{\sqrt{p(I)}} \sum_{j=0}^{2^m-1} c_{I,j} |I,j\rangle$$

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

2.3 Quantum Programming

2.3.1 Quantum Parallelism

Since all unitary transformations are linear operators, any operation performed on a quantum state is simultaneous applied to all its basevectors, thus

$$U \sum_i c_i |i\rangle = \sum_i c_i U |i\rangle$$

This unique feature of quantum computers is called *quantum parallelism*.

Since the number of basevectors exponentially increases with the number of qubits, it is possible to solve certain problems (e.g. prime factorisation of large numbers, see section 4) in polynomial time (i.e. the number of elementary operations is a polynomial in the length of the input) where a classical computer would need an exponential number of steps.

2.3.2 Handling of Non-Reversible 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 $\mathit{DIV2}'|0\rangle = \mathit{DIV2}'|1\rangle$.

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

Generally it can be said that for any function $f : \mathbf{B}^n \to \mathbf{B}^n$ (or equivalently $f : \mathbf{Z}_{2^n} \to \mathbf{Z}_{2^n}$) there exists a unitary operator $F : \mathbf{C}^{2^{2n}} \to \mathbf{C}^{2^{2n}}$ (thus working on two n qubits registers) with $F|x,0\rangle = |x, f(x)\rangle$.

2.3.3 Scratch Space Management

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. In longer calculations this would leave us with a steadily increasing amount of "junk" bits which are of no concern for the final result.

A simple and elegant solution of this problem was proposed by Bennet [3, 4]. 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:

$$|x,0,0\rangle \to |x,g(x),0\rangle \to |x,g(x),h(g(x))\rangle \to |x,0,h(g(x))\rangle = |x,0,f(x)\rangle$$

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.

[1] In this special case, just one additional qubit to hold the lowest bit of the argument would suffice to extend *DIV2'* to a unitary operator.

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:

$$|x,0,0,0\rangle \to |x,f(x),j(x),0\rangle \to |x,f(x),j(x),f(x)\rangle \to |x,0,0,f(x)\rangle$$

Again, the last step is the inversion of the first. The intermediate step is a *FANOUT* operation which copies the function result into an additional empty (i.e. in substate $|0\rangle$) register.

3 The Quantum Class Library

3.1 General Information

3.1.1 About the Program

QULIB is a C++ library for the simulation of quantum computers on an abstract functional level. That means that, although the simulation of a certain hardware architecture (like e.g. the linear ion trap) is also possible, its primary purpose is the verification of high-level quantum algorithms especially in the field of number theory. The main features are

- Basevectors of arbitrary length (no limitation by hardware wordlength)
- Efficient representation of quantum states by hashtables
- Nesting of substates and arbitrary combination of qubits
- Composition and cross product of quantum operators
- Easy extendibility by class inheritance

3.1.2 Hard- and Software

QULIB has been developed on an Intel 486DX4 PC with 100 MHz and 16 MB RAM under Linux using the GNU C/C++ Compiler 2.7.2 but should compile under any operating system with an ANSI C++ compiler.

3.1.3 Sourcefiles

`Makefile`	Makefile for the QULIB library file `libqu.a` and the program `shor`.

`bitvec.h, bitvec.cxx`	Representation and handling of bitvectors
`terms.h, terms.cxx`	Internal types for the representation of quantum states and terms (i.e. a basevector with a complex amplitude)
`qustates.h, qustates.cxx`	User classes for quantum states and substates
`operator.h, operator.cxx`	User classes for quantum operators
`shor.cxx`	Shor's quantum algorithm for prime factorisation

3.2 Simulation of Quantum Computers

3.2.1 Representation of Basevectors

The Hilbert space $\mathcal{H}$ of a quantum computer with n qubits has 2^n dimensions. The basevectors of $\mathcal{H}$ are bitstrings of the length n, which are represented by the class `bitvec`.

If the wordlength N of the computer is smaller than n, then an array is dynamically allocated; if it $N \geq n$, the vector is stored as an `unsigned int`, which is considerably faster and should suffice for most applications.

3.2.2 Representation of Quantum States

A quantum state $|\psi\rangle \in \mathcal{H}$ of n qubits can be described by 2^n complex numbers.

$$|\psi\rangle = \sum_{i=0}^{2^n-1} c_i |i\rangle \quad \text{with} \quad \sum_{i=0}^{2^n-1} c_i c_i^* = 1$$

Only terms with $c_i \neq 0$ must be stored, which can considerably reduce the required amount of memory since registers which are entangled by an arithmetic function (which is normally the case with all scratch registers) don't require additional entries.

The best datastructure for storing those terms is determined by the most common forms of access, which, in our case, are sequential read out and adding of terms, where it is crucial that the time to determine whether a basevector is already in the list (and the amplitudes have to be added) and the time fore adding a new entry (if the basevector in not in the list) are of the order $O(1)$ i.e. don't increase with the size of the list. Removing of single elements is normally not needed an may take longer.

A linear array in combination with a hashtable as index can meet all these requirements. A basevector is thereby mapped onto the hashtable by a hashfunction which provides a pseudo-random distribution. The entry of the hashtable contains a pointer to the corresponding entry in the array or indicates that such an entry doesn't exist. The overhead for the detection and handling of collisions is of order $O(1)$ if the hashtable is considerably longer than the list itself.

The internal class `termlist` provides this datastructure with all necessary access function and the ability to dynamically adapt the size of arrays and hashtables by powers of 2. The class `quBaseState` is the user class representing the common state of all qubits and contains two `termlist` objects which alternately serve as argument or result of operations.

3.2.3 Substates

Since all qubits are entangled, substates can't be represented as an isolated datastructure, but are merely references to certain qubits of an associated basestate. A reference to m qubits of an n qubit basestate represents a subspace $\mathcal{S} = \mathbf{C}^{2^m}$ of $\mathcal{H}$.

The class `quSubState` and all its derived classes make this reference transparent to the user i.e. all manipulations on the substate are correctly mapped onto its basestate:

Let $|\psi\rangle = |\phi\rangle|\chi\rangle$ be the n qubit basestate of the substate $|\phi\rangle$ referring to the first m qubits of $|\psi\rangle$ and $U : \mathcal{S} \to \mathcal{S}$ a unitary operator

$$|\psi\rangle = |\phi\rangle|\chi\rangle = \sum_{i=0}^{2^m-1} \sum_{j=0}^{2^{n-m}-1} c_{i,j}|i,j\rangle, \quad U|i\rangle = \sum_{k=0}^{2^m-1} u_{i,k}|k\rangle$$

Applying U to $|\phi\rangle$ would be equivalent to applying $U \times ID(n-m)$ to $|\psi\rangle$ where $ID(k)$ is the identity operator over $\mathbf{C}^{2^k}$.

$$U \times ID(n-m)|\psi\rangle = \sum_{i=0}^{2^m-1} \sum_{j=0}^{2^{n-m}-1} \sum_{k=0}^{2^m-1} u_{i,k}c_{i,j}|k,j\rangle$$

3.2.4 Operators

In principle, any unitary operator on n qubits can be represented by a complex $2^n \times 2^n$ matrix. Applying this operator to a state with k nonzero terms would require $O(2^n k)$ multiplications. Operators of this kind are represented by the class `opMatrix`, which stores the nonzero elements of each row in an array of linear lists.

Most operators, however, work on limited subspaces of only a few qubits or (as e.g. all arithmetic operators) merely substitute basevectors with or without an additional phase factor. QULIB provides classes for all these special cases:

The class `opEmbedded` represents operators of the kind $OP = ID(n) \times U \times ID(m)$, `opPermutation` are operators of the form $U|i\rangle = c_i|j_i\rangle$ and are stored as a one dimensional array of terms.

The class `opFunction` represents arithmetic functions of the form $U|i,0\rangle \to |i,f(i)\rangle$, where $f(i)$ is defined as a virtual member function. The result of $U|i,j\neq 0\rangle$ is undefined and would lead to an error.

A more general interface for user defined function is the class `opGate` for operators of the form $U|i\rangle \to c(i)|f(i)\rangle$. The functions $f(i)$ and $c(i)$ are provided by the user, who is responsible that the transformation is unitary.

3.3 Class Hierarchy

The following list contains all QULIB classes grouped by their header files. Virtual base classes are signed with an asterix (*).

3.3.1 File bitvec.h

`bitvec`	basevectors of $\mathcal{H}$ (bit strings)

3.3.2 File terms.h

`term`	a basevector multiplied with a complex amplitude
`termlist`	hashtable of terms representing a quantum state
`probtree`	binary tree of vectors with a real amplitude to represent the spectrum of a state

3.3.3 File qustates.h

`quState *`	base class for all quantum states
`  quBaseState`	base state containing the actual state information
`  quSubState *`	substate referring to another base or substate
`    quVar`	container class for the assigning of substates
`    quCombState`	concatenation of 2 substates
`    quSubString`	coherent substring of qubits
`      quWord`	substring interpreted as word
`      quBit`	substring of length 1 i.e. a single qubit

3.3.4 File operator.h

`opOperator *`	base class of all operators
`  opElementary *`	elementary (i.e. not composed) operator
`    opMatrix`	operator stored as array of nonzero matrix elements
`      opU2`	general 1 bit unitary operator
`    opIdentity`	identity operator
`    opSwap`	operator for swapping to substrings

`opPermutation`	operator stored in a linear list of replacing terms
`opFunction *`	operator of the form $\|x, 0\rangle \to \|x, f(x)\rangle$
`opEXPN`	modular exponentiation ($\|a, 0\rangle \to \|a, x^a \bmod N\rangle$, see section 4.2.1)
`opGate *`	operator implemented as C++ function
`opCk`	$(k+1)$-dimensional controlled-NOT gate
`opC0`	inverts one arbitrary bit of state
`opNot`	1 bit NOT-gate
`opC1`	controlled-NOT with 1 arbitrary input
`opXor`	2 bit XOR-gate (controlled-NOT with 1 fixed input)
`opC2`	controlled-NOT with 2 arbitrary inputs
`opToffoli`	3 bit Toffoli-gate (controlled-NOT with 2 fixed inputs)
`opCondPhase`	conditional phase gate with k arbitrary inputs
`opX`	cond. phase gate with 2 inputs and $\phi = \frac{2\pi}{2^n}$
`opComposition`	composition of 2 operators
`opEmbedded`	container class for operators working on substates
`opVar`	container class for assigning and composing of operators

For a description of constructors and member functions please refer to appendix A.1 and A.2.

4 Shor's Algorithm for Quantum Factorisation

4.1 Motivation

In contrast to finding and multiplying of large prime numbers, no efficient classical algorithm for the factorisation 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 [7] 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 cryptosystems 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 (e.g. Netscape, PGP, etc.).

While it is generally believed (although not formally proved) that efficient prime factorisation on a classical computer is impossible, an efficient algorithm for quantum computers has been proposed in 1994 by P.W. Shor [6].

4.2 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 device can be found in [8]. For a more rigid mathematical description, please refer to [9].

4.2.1 Modular Exponentiation

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

$$F_N(k) = x^k \bmod N,\ F_N(k+r) = F_N(k),\ x^r \equiv 1 \bmod N$$

The function F_N performs a modular exponentiation, its 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:

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

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 $\mathrm{GF}(p)$ (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 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* for determing $\gcd(N, a+1)$ and $\gcd(N, a-1)$ which is defined as

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

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 factorise pure prime powers (and of course to recognise a factor of 2), an efficient probabilistic algorithm for factorisation can be found if the period r of the modular exponentiation can be determined in polynomial time.

4.2.3 Period of a Sequence

Let F be an operator of the form $F|x,0\rangle \to |x, f(x)\rangle$ and $f : \mathbf{Z} \to \mathbf{Z}_{2^m}$ a function 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 basevectors in the first register by applying an operator U with

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

This can e.g. be achieved by transforming each qubit with the $U2(\frac{\pi}{2})$ operator (see section 2.1.2). Another possibility is the discrete *fast Fourier transform* (*FFT*) which is defined for $2n$ qubits as

$$FFT|i\rangle = \frac{1}{2^n} \sum j = 0^{2^{2n}-1} \mathrm{e}^{\frac{2\pi \mathrm{i}}{2^{2n}}\, ij} |j\rangle$$

Applying F to the resulting state gives

$$|\psi\rangle = F \cdot FFT|0,0\rangle = F\, \frac{1}{2^n} \sum_{i=0}^{2^{2n}-1} |i,0\rangle = \frac{1}{2^n} \sum_{i=0}^{2^{2n}-1} |i, f(i)\rangle$$

A measurement of the second register with the result $k = f(s)$ with $s < r$ reduces the state to

$$|\psi'\rangle = \sum_{j=0}^{\lceil q/r \rceil - 1} c'_j |rj+s, k\rangle \quad \text{with} \quad q = 2^{2n} \quad \text{and} \quad c'_j = \sqrt{\left\lceil \frac{r}{q} \right\rceil}$$

The post-measurement state $|\psi'\rangle$ of the first register consists only of basevectors 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. The result of a Fourier transform, however, is invariant (except for phase factors which don't effect the probability spectrum) to offsets of a periodic distribution.

$$|\tilde{\psi}'\rangle = FFT|\psi'\rangle = \sum_{i=0}^{q-1} \tilde{c}_i'|i,k\rangle$$

$$\tilde{c}_i' = \frac{\sqrt{r}}{q}\sum_{j=0}^{p-1}\exp\left(\frac{2\pi \mathrm{i}}{q}\,i(jr+s)\right) = \frac{\sqrt{r}}{q}\,\mathrm{e}^{\phi_i}\sum_{j=0}^{p-1}\exp\left(2\pi \mathrm{i}\,\frac{ijr}{q}\right)$$

$$\text{with} \quad \phi_i = 2\pi\mathrm{i}\,\frac{is}{q} \quad \text{and} \quad p = \left\lceil \frac{q}{r} \right\rceil$$

If $q = 2^{2n}$ is a multiple of r then $\tilde{c}_i' = \mathrm{e}^{\phi_i}/\sqrt{r}$ if i is a multiple of q/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 q/r because

$$\lim_{n\to\infty}\frac{1}{n}\sum_{k=0}^{n-1}\mathrm{e}^{2\pi\mathrm{i}k\alpha} = \begin{cases} 1 & \text{if } \alpha \in \mathbf{Z} \\ 0 & \text{if } \alpha \notin \mathbf{Z} \end{cases}$$

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 $\lambda q/r$ with $\lambda \in \mathbf{Z}_r$. This can be written as $c/q = c \cdot 2^{-2n} \approx \lambda/r$. 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 an efficient classical algorithm for solving this problem using continued fractions is described in [10] and also implemented in the program `shor` (see section 4.3).

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.

4.3 The program shor

The program `shor` is an implementation of Shor's algorithm using the library QULIB to simulate an abstract quantum computer. It is written in C++ under Linux and should run on every computer with an ANSI C++ compiler.

For an overview of the used functions and a summary of the main program please refer to appendix B.

4.3.1 Usage

`shor` is started from the command line with

```
shor <number> [options]
```

The first parameter is the number to be factorised. The following options are supported:

-s seed sets the seed value for the pseudo random number generator needed for the simulation of measurements and the selection of the base value x (see section 4.2.1).

Any value from 0 to $2^{31} - 1$ is allowed. If no seed value is provided, the actual system clock is used.

-t maxtries sets the maximum numbers of selections (i.e. measurements without state reduction) from the same $|\psi'\rangle$ state in case of failure.

Of course, this would not be possible on a real quantum computer, but it can be interpreted as repeating the computation with the same x value (see section 4.2.1) and "by chance" measuring the same value in the second register.

The default value for this option is 3. If you prefer a more "realistic" simulation, set the value to 1 and after every failure the whole simulation is restarted from scratch.

-g gates sets the maximum number of conditional phase gates per bit used in the implementation of the fast Fourier transform (see [8] for details).

A lower number means that the FFT is less accurate but carried out faster ($O(n)$ instead of $O(n^2)$ elementary operators). There is a tradeoff between faster execution and higher probability of failure due to less accurate peaks in the spectrum (see section 4.2.3).

This option is realistic in the sense that it could be implemented on a real quantum computer. If no value is given, the FFT is carried out exactly.

-q (operate quietly) No log output is produced and only the result of the factorisation is printed.

-v (operate verbosely) Every step of the algorithm commented.

-l (log spectrums) After each operation or measurement the spectrum of the quantum registers is printed.

-d (dump states) After each operation or measurement the state of the quantum registers is printed

4.3.2 Error Messages

When called with an illegal syntax, `shor` produces a USAGE message. When a number is cannot be factorised with Shor's algorithm, the program terminates with an explaining message:

```
number must be odd !

81 is a prime power of 3 !

59 is a prime number !
```

4.3.3 Factoring 15

15 is the smallest number which can be factorised with Shor's algorithm. The command `shor 15 -v -t1 -s7` starts the simulation in verbose mode with a random seed value of 7 and immediate recalculation in case of failure. The following output is produced:

```
factoring 15: random seed = 7, tries = 1.
allocating 12 quBits with 256 terms.

RESET:   reseting state to |0,0>
FFT:     performing 1st Fourier transformation.
EXPN:    trying x = 2. |a,0> --> |a,2^a mod 15>
MEASURE: 2nd register: |*,1>
FFT:     performing 2nd Fourier transformation.
MEASURE: 1st register: |0,1>
<failed> measured zero in 1st register. trying again ...

RESET:   reseting state to |0,0>
FFT:     performing 1st Fourier transformation.
EXPN:    trying x = 8. |a,0> --> |a,8^a mod 15>
MEASURE: 2nd register: |*,4>
FFT:     performing 2nd Fourier transformation.
MEASURE: 1st register: |64,4>
rational approximation for 64/2^8 is 1/4, possible period: 4
8^2 mod 15 = 4. possible common factors of 15 with 5 and 3.
15 = 5 * 3.
program succeeded after 1 s and 2 iterations.
```

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 which would suggest a probability of $p = 1/q = 1/256$ to measure 0. In fact, 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 $q/\sqrt{n}$ after the FFT or $p = 25.8\%$.

In the special case of a start value $x = 2$ the period of modular exponentiation is 4 since $2^4 \bmod 15 = 1$, consequently the Fourier spectrum shows 4 peaks at $|0\rangle$, $|64\rangle$, $|128\rangle$ and $|192\rangle$ and $p = 1/4$ as expected. This can be verified by running `shor` with the option `-l`.

The second try also had the same probability of failure since 8 is the multiplicative inverse to 2 in $\mathbf{Z}_{15}$, but this time, the measurement picked the second peak in the spectrum at $|64\rangle$. With $64/2^8 = 1/4 = \lambda/r$, the period $r = 4$ was correctly identified and the possible common factors $8^2 \pm 1 \bmod 15$ with 15 have been found.

References

[1] R. W. Keyes *1988 IBM J. Res. Develop. 32, 24*

[2] D. Deutsch *1989 Quantum computational networks. Proceedings of the Royal Society London A 439, 553-558*

[3] C. H. Bennet *1973 IBM J. Res. Develop. 17, 525*

[4] C. H. Bennet *1989 SIAM J.Comput. 18, 766*

[5] Johannes Buchmann *1996 Faktorisierung großer Zahlen. Spektrum der Wissenschaft 9/96, 80-88*

[6] P.W. Shor. *1994 Algorithms for quantum computation: Discrete logarithms and factoring*

[7] Samuel L. Braunstein *1995 Quantum computation: a tutorial*

[8] David Beckman et al. *1996 Efficient networks for quantum factoring*

[9] Artur Ekert and Richard Jozsa. *1996 Shor's Quantum Algorithm for Factoring Numbers, Rev. Modern Physics 68 (3), 733-753*

[10] G.H. Hardy and E.M. Wright *1965 An Introduction to the Theory of Numbers (4th edition OUP)*

A QULIB Quick Reference

A.1 Constructors

```
QUANTUM STATES

quBaseState(int bits,int buflen=256);
        new base state with <bits> qubits and an initial buffer for
        <buflen> terms.
quVar(); quVar(quVar& qs); quVar(quState& qs);
        state variable either empty or set to state <qs>
quCombState(quState& head,quState& tail);
        concatenation of <head> and <tail>
quSubString(int bits,int offs,quState& base);
        substate of <bits> bits from state <base>, beginning at bit <offs>
quBit(int offs,quState& base);
        bit <offs> of state <base>
quWord(int bits,int offs,quState& base);
        substate of <bits> bits from state <base>, beginning at bit <offs>.
        interpreted as word

OPERATORS

opMatrix(int n,term **m);
        matrix op. operating on <n> bits, <m> is a 2 dim. array of nonzero
        matrix elements (1st dim. is 2^n, all list terminated with term())
opU2(double theta,double delta=0,double sigma=0,double tau=0);
        U2 = |  e^i(del+sig+tau) cos(th/2)    e^-i(del+sig-tau) sin(th/2) |
             | -e^i(del-sig+tau) sin(th/2)    e^i(del-sig-tau) cos(th/2)  |
opIdentity(int n);
        identity op. on <n> bits
opSwap(int n,int m,int o1,int o2);
        swaps 2 not overlapping substrings of length <m>:
        |a_0, .. a_o1, .. a_o1+m-1, .. a_o2, .. a_o2+m-1, .. a_n-1> -->
        |a_0, .. a_o2, .. a_o2+m-1, .. a_o1, .. a_o1+m-1, .. a_n-1>
opPermutation(int n,term *p);
        replaces an eigenvec. with an other eigenvec. with ampl. |i> --> p[i]
opEXPN(int arg,int fct,word x,word num);
        modular exponentiation |a,0> --> |a,x^a mod num>
opCk(int n,int k,int o,int *i);
        controlled-NOT: inverts bit <o> if all bits i[0] to i[k-1] are set
opC0(int n,int o);
        |a_0, .. a_o, .. a_n-1> --> |a_0, .. NOT a_o, .. a_n-1>
opNot();
        NOT-Gate: |0> --> |1>, |1> --> |0>
opC1(int n,int o,int i);
        |a_0,.. a_o,.. a_n-1> --> |a_0,.. (a_i XOR a_o),.. a_n-1>
opXor();
        XOR-Gate: |i,o> --> |i,i XOR o>
opC2(int n,int o,int i1,int i2);
        |a_0, .. a_o, ..a_n-1> --> |a_0, .. (a_i1 AND a_i2) XOR a_o, .. a_n-1>
```

```
opToffoli();
        Toffoli-Gate: |o,i1,i2> --> |(i1 AND i2) XOR o,i1,i2>
opCondPhase(int n,int k,int *i,double phi);
        cond. phase gate: multiplies bit <o> with e^( i phi) if all bits i[0]
        to i[k-1] are set
opX(int n,int i1,int i2,int pow);
        |a_0, .. a_n-1> --> e^i(2 PI i1 i2 / 2^pow) |a_0, .. a_n-1>
opComposition(const opOperator& in,const opOperator& out);
opComposition(opOperator *in,opOperator *out);
        composition out * in: |psi> --> out in |psi>
opEmbedded(int n,int offs,const opOperator& op);
opEmbedded(int n,int offs,opOperator *op);
        |a_0..>|a_offs..>|..a_n-1> -> |a_0..> (op |a_offs..>) |..a_n-1>
opVar(); opVar(const opVar& op); opVar(const opOperator& op);
        operator variable either empty or set to <op>
```

A.2 Member Functions

This list contains only functions declared in the base classes and no constructors or destructors. Virtual functions are signed with an asterix.

QUANTUM STATES

```
  int mapbits();                                no. of referenced bits
  bitvec measure();                             destructive measurement
  void opterm(const term& t);                   add term to base (op.interface)
  void printvect(ostream& s,const bitvec& v);   print vector to stream s
  quState* newclone();                          generate a copy of the object
  bitvec select();                              non-destructive measurement
  void reduce(const bitvec& v);                 project state on (sub)vector v
  void normalize(double epsilon=EPSILON);       scale state to norm==1
  probtree* newspectrum();                      generate the spectrum of state
* void reset(const bitvec& v=bitvec(0));        reset state to (sub)vector v
* baseid base();                                object ID
* int basebits();                               no. of bits of base state
* quState* newsubstring(int bits,int offs);     generate ref. object to substr.
* int isbasestate();                            test if state is a basestate
* bitvec mapmask();                             bitmask of referenced base bits
* complex ampl(const bitvec& v);                complex amplitude of vector v
* int baseterms();                              no. of corresponding base terms
* term& baseterm(int i);                        get baseterm (stating with 0)
* void opbegin();                               open operator interface
* void opadd(const bitvec& v,const complex& z); add term (v,z) to base
* void opend();                                 close operator interface
* bitvec map(const bitvec& v);                  get ref. bits from base vector
* bitvec unmap(const bitvec& v);                expand ref. bits to base vect.
* void _printvect(ostream& s,const bitvec& v);  print vector w/o parentheses
```

```
ostream& operator << (ostream& s,const quState& qs);
        output operator: prints spectrum of qs to stream s
ostream& printbase(ostream& s,quState& qs,char* prefix=0,char* postfix=0);
```

```
        output function: prints base state with complex amplitudes
quCombState operator / (quState& head,quState& tail);
        concats the states head and tail

OPERATORS

  int bits() { return _bits; };                no. of qubits (dim = 2^bits)
  quState& operator () (quState& qs);          apply op. on qs (shorthand)
* void apply(quState& qs);                     apply op. on state qs
* opOperator *newclone();                      generate a copy of the object

opComposition operator * (const opOperator& out,const opOperator& in);
        produces a composition out*in of the operators out and in
        (out*in)(qs) --> out(in(qs))
opComposition operator / (const opOperator& low,const opOperator& high);
        produces the cross product of the ops. low and high
        (low/high)(qs1/qs2) --> low(qs1)/high(qs2)
```

B shor.cxx

B.1 Functions

```
int factorize(word n,word *a,word *b);
        returns 0 and sets *a and *b if n = (*a) * (*b)
        returns 1 if n is a prime number

int testpower(word p,word b);
        returns 1 if p is a power of b and 0 otherwise

word powmod(word x,word a,word n);
        returns x^a mod n

int gcd(int a,int b);
        returns the greatest common divisor of a and b

int randcoprime(n);
        returns a random number 1 < r < (n-1) coprime to n

void approx(double x,word qmax,word *p,word *q);
        finds the best rational approximation (*p)/(*q) to x with
        denominator < qmax and sets *p and *q accordingly.

opVar opFFT(int n);
        performs a fast Fourier transformation on qs using
        Coppersmith's algorithm
```

B.2 Main Program

```
word number;                    // number to be factored
word factor;                    // found factor
int width=duallog(number);      // length of N in bits
int nreg1=2*width,nreg2=width;  // width of registers
quBaseState qubase(nreg1+nreg2);// basestate
quWord reg1(nreg1,0,qubase);    // register 1
quWord reg2(nreg2,nreg1,qubase);// register 2
word x;                         // base value
word mreg1,mreg2;               // measurements of 1st and 2nd register
word pow;                       // pow^2==1 mod number
word a,b;                       // possible factors
word p,q;                       // fraction p/q for rational approximation
double qmax=1<<width;           // maximal period

while(1) {
  qubase.reset();                        // reseting state
  opFFT(nreg1)(reg1);                    // 1st Fourier transformation
  x=randcoprime(number);                 // selecting random x
  opEXPN(nreg1,nreg2,x,number)(qubase); // modular exponentiation
  mreg2=reg2.measure().get-word();        // measure 2nd register
  opFFT(nreg1)(reg1);                    // 2nd Fourier transformation
  mreg1=reg1.select().getword();         // measure 1st register
  if(mreg1==0) continue;                 // failed if measured zero

  // finding rational approximation for mreg1/rmax^2
  approx((double) mreg1/(qmax*qmax),(int) qmax,&p,&q);
  if(q%2) continue;                      // failed if q is odd.
  pow=powmod(x,q/2,number);              // pow = x^(q/2) mod number
  a=(pow+1)%number;                      // candidates with possible
  b=(pow+number-1)%number;               // common factors with number

  // testing for common factors with number
  if(a>1 && (factor=gcd(number,a))>1) break;
  if(b>1 && (factor=gcd(number,b))>1) break;
};
```