Multiplexor Approximation Method For Quantum Compilers

Robert R. Tucci

P.O. Box 226

Bedford, MA 01730

tucci@ar-tiste.com

December 8, 2005

CROSS REFERENCES TO RELATED APPLICATIONS

Not Applicable

STATEMENT REGARDING FEDERALLY SPONSORED RESEARCH AND DEVELOPMENT

Not Applicable

BACKGROUND OF THE INVENTION

(A)FIELD OF THE INVENTION

The invention relates to an array of quantum bits (qubits) commonly known as a quantum computer. More specifically, it relates to methods for translating an input data-set into a sequence of operations according to which such an array can be manipulated.

(B)DESCRIPTION OF RELATED ART

This invention deals with Quantum Computing. A quantum computer is an array of quantum bits (qubits) together with some hardware for manipulating these qubits. Quantum computers with several hundred qubits have not been built yet. However, once they are built, it is expected that they will perform certain calculations much faster that classical computers. A quantum computer can be made to follow a sequence of elementary operations. The operations are elementary in the sense that they act on only a few qubits (usually 1, 2 or 3) at a time. For example, CNOTs and one-qubit rotations are elementary operations. (A CNOT is a special type of operation that spans two qubits: a control bit and a target bit. The control bit is often called just the control and the target bit just the target.) Henceforth, we will sometimes refer to sequences as products and to operations as operators, instructions, steps or gates. Furthermore, we will abbreviate the phrase "sequence of elementary operations" by "SEO". SEOs are often represented as qubit circuits. For a detailed discussion of quantum computing, see **NieChu00** [M. Nielsen, I. Chuang, *Quantum Computation and Quantum Information*, (Cambridge University Press, 2000)] . Also, one can find at www.arxiv.org some excellent, more recent, free introductions to quantum computing.

We will use the term quantum compiling algorithm to refer to an algorithm that can decompose ("compile") an arbitrary unitary matrix into a SEO, which can then be used to operate a quantum computer. We will use the term quantum compiler to refer to a software program that runs on a classical computer and implements a quantum compiling algorithm (It may do more than this). An early type of quantum compiling algorithm is discussed in **Bar95**[A. Barenco et al, "Elementary Gates for Quantum Computation", quant-ph/9503016] . A different type of quantum compiling algorithm was invented by Tucci and is discussed in **QbtrPat** [US Patent 6,456,994 B1, by R. R. Tucci] , **Tuc99** [R.R. Tucci, "A Rudimentary Quantum Compiler (2cnd Ed.)", quant-ph/9902062] , **Tuc04Nov** [R.R. Tucci, "Qubiter Algorithm Modification, Expressing Unstructured Unitary Matrices with Fewer CNOTs", quant-ph/0411027] , and **Tuc04Dec** [R.R. Tucci, "Quantum Compiling with Approximation of Multiplexors", quant-ph/0412072] . Tucci has also written a quantum compiler program called Qubiter that implements the ideas of **QbtrPat**.

A $U(2)$-multiplexor will be defined precisely later in this document. The quantum compiling algorithm of **QbtrPat** and related work decomposes an arbitrary unitary matrix into a sequence of $U(2)$-multiplexors, each of which is then decomposed into a SEO. (Although **QbtrPat** uses $U(2)$-multiplexors, it does not refer to them by this name, which is of more recent vintage.) After **QbtrPat** was issued, other workers (see **Mich04** [V.V.Shende, S.S.Bullock, I.L.Markov, "A Practical Top-down Approach to Quantum Circuit Synthesis", quant-ph/0406176] and **Hels04** [V. Bergholm, J. Vartiainen, M.Mottonen, M. Salomaa, "Quantum circuit for a direct sum of two-dimensional unitary operators", quant-ph/0410066]) have proposed alternative quantum compiling algorithms that also generate $U(2)$-multiplexors as an intermediate step.

One measure of the inefficiency of a quantum compiler is the number of CNOTs it uses to express an unstructured unitary matrix (i.e., a unitary matrix with no special symmetries). Call this number N_{CNOT}. Although good quantum compilers must also deal with structured matrices, unstructured matrices are certainly an important case worthy of attention. Minimizing N_{CNOT} is a reasonable goal, since a CNOT operation (or any multi-qubit operation) is expected to take more time to perform and to introduce more environmental noise into the quantum computer, than a one-qubit rotation. **Mich03** [V.V.Shende, I.L.Markov, S.S.Bullock, "On Universal Gate Libraries and Generic Minimal Two-qubit Quantum Circuits", quant-ph/0308033] proved that for matrices of dimension 2^{N_B} (where N_B = number of bits), one has $N_{CNOT} \geq \frac{1}{4}(4^{N_B} - 3N_B - 1)$. This lower bound is achieved for $N_B = 2$ by the 3 CNOT circuits first proposed in **Vidal93** [G. Vidal, C.M. Dawson, "A Universal Quantum Circuit for Two-qubit Transformations with 3 CNOT Gates", quant-ph/0307177] . It is not known whether this bound can be achieved for $N_B \geq 3$.

As the table of FIG.**1** illustrates, compiling an unstructured unitary matrix with $N_B > 10$ requires more than a million CNOTs. Thus, we desperately need an approximation method whereby, given any unitary matrix U_{in}, we can find another unitary matrix V such that: (1) V approximates U_{in} well, and (2) V is expressible with fewer CNOTs than U_{in}. This patent proposes one such approximation method.

The use of approximation methods in quantum compiling dates back to the earliest papers in the field. For example, **Copper94** [Don Coppersmith, "An approximate Fourier transform useful in quantum factoring", (1994 IBM Internal Report), quant-ph/0201067] and **Bar95** contain discussions on this issue. The approximation method claimed herein differs substantially from all previously proposed methods. Unlike previous approximation methods, the method propose herein involves approximating U(2)-multiplexors.

The method proposed herein for approximating U_{in} is to approximate some or all of the intermediate $U(2)$-multiplexors whose product equals U_{in}. One can approximate a $U(2)$-multiplexor by another $U(2)$-multiplexor (the "approximant") that has fewer controls, and, therefore, is expressible with fewer CNOTs. We will call the reduction in the number of control bits the bit deficit δ_B. FIG.**2** is emblematic of our approach. It shows a $U(2)$-multiplexor with 3 controls being approximated by either a $U(2)$-multiplexor with 2 controls or one with 1 control.

Tucci has previously published a description of this invention in **Tuc04Dec**. **Tuc04Dec** presents some details about this invention that are not included in this specification. Tucci considers **Tuc04Dec** to be essentially correct and in agreement with this specification, and he wishes **Tuc04Dec** to be considered a part of this specification.

BRIEF SUMMARY OF THE INVENTION

A quantum computer is an array of quantum bits (qubits) together with some hardware for manipulating these qubits.

A quantum compiling algorithm is an algorithm for decomposing ("compiling") an arbitrary unitary matrix into a sequence of elementary operations (SEO), which can then be used to operate a quantum computer. A quantum compiler is a software program that runs on a classical computer and implements a quantum compiling algorithm.

A quantum compiler previously invented by Tucci decomposes an arbitrary unitary matrix U_{in} into a sequence of intermediate $U(2)$-multiplexors, each of which is then decomposed into a SEO.

A preferred embodiment of this invention is a subroutine within a quantum compiler program. The subroutine approximates some or all of the intermediate $U(2)$-multiplexors whose product equals U_{in}. The effect of using the subroutine within the quantum compiler is that we obtain a SEO that doesn't equal U_{in} exactly, but has the virtue of containing fewer CNOTs (or some other type of multi-qubit gate) than an exact SEO.

In a preferred embodiment of the invention, the subroutine approximates a $U(2)$-multiplexor by another $U(2)$-multiplexor that has fewer controls, and, therefore, is expressible with fewer CNOTs.

BRIEF DESCRIPTION OF THE DRAWINGS

FIG. **1** shows, for each N_B, a lower bound on the number of CNOTs required to express an N_B-bit unstructured unitary matrix.

FIG. **2** shows an example of approximating a $U(2)$-multiplexor by another $U(2)$-multiplexor with δ_B fewer controls.

FIG. **3** shows two diagrammatic representations of a $U(2)$-multiplexor.

FIG. **4** shows two possible decompositions of an $R_y(2)$-multiplexor with 1 control.

FIG. **5** shows four possible decompositions of an $R_y(2)$-multiplexor with 2 controls.

FIG. **6** shows one of several possible decompositions of an $R_y(2)$-multiplexor with 3 controls.

FIG. **7** shows one of several possible decompositions of an $R_y(2)$-multiplexor with 4 controls.

FIG. **8** shows the first part of an "out_phis.txt" file.

FIG. **9** shows box **91**, which is the second part of an "out_phis.txt" file (first part in FIG.**8**), and box **92**, which is all of an "out_error.txt" file.

FIG. **10** shows a block diagram of a classical computer feeding data to a quantum computer.

DETAILED DESCRIPTION OF THE INVENTION

(A)Theory Behind New Method

Notation

First, we will define some notation that is used throughout this patent and in related documents. For additional information about our notation, we recommend that the reader consult **Tuc04Dec** and **Paulinesia**[R.R.Tucci, "QC Paulinesia", quant-ph/0407215] . **Paulinesia** is a review article, written by the author of this patent, which uses the same notation as this patent.

Let $Bool = \{0, 1\}$. As usual, let $\mathbb{Z}, \mathbb{R}, \mathbb{C}$ represent the set of integers (negative and non-negative), real numbers, and complex numbers, respectively. For integers a, b such that $a \leq b$, let $\mathbb{Z}_{a,b} = \{a, a+1, \ldots b-1, b\}$. For Γ equal to $\mathbb{Z}$ or $\mathbb{R}$, let $\Gamma^{>0}$ and $\Gamma^{\geq 0}$ represent the set of positive and non-negative Γ numbers, respectively. For any positive integer r and any set S, let S^r denote the Cartesian product of r copies of S; i.e., the set of all r-tuples of elements of S.

For any (not necessarily distinct) objects $a_1, a_2, a_3, \ldots$, let $\{a_1, a_2, a_3, \ldots\}_{ord}$ denote an ordered set. For some object b, let $b\{a_1, a_2, a_3, \ldots\}_{ord} = \{ba_1, ba_2, ba_3, \ldots\}_{ord}$. Let $\emptyset$ be the empty set. For an ordered set S, let S^R be S in reverse order.

We will use $\theta(S)$ to represent the "truth function"; $\theta(S)$ equals 1 if statement S is true and 0 if S is false. For example, the Kronecker delta function is defined by $\delta_x^y = \delta(x, y) = \theta(x = y)$.

For any positive integer N, we will use $\vec{e}_i$ where $i = 1, 2, \ldots, N$ to denote the standard basis vectors in N dimensions; i.e., $[\vec{e}_i]_j = \delta(i,j)$ for $i,j \in \mathbb{Z}_{1,N}$.

I_r and 0_r will represent the r-dimensional unit and zero matrices.

For any matrix A and positive integer r, let $A^{\otimes r}$ denote the r-fold tensor product of r copies of A. Likewise, let $A^{\oplus r}$ denote the r-fold direct sum of r copies of A.

For any matrix $A \in \mathbb{C}^{r\times s}$ and $p = 1, 2, \infty$, let $\|A\|_p$ represent the p-norm of A, and $\|A\|_F$ its Frobenius norm. See **Golub96**[G.H. Golub and C.F. Van Loan, *Matrix Computations, Third Edition* (John Hopkins Univ. Press, 1996)] for a discussion of matrix norms.

Let $\vec{x} \in \mathbb{C}^{r\times 1}$. As is customary in the Physics literature, $\|\vec{x}\|_2$ will also be denoted by $|\vec{x}|$ and called the magnitude of $\vec{x}$.

The Pauli matrices $\sigma_x, \sigma_y, \sigma_z$ are defined by

$$\sigma_x = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} , \quad \sigma_y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} , \quad \sigma_z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} . \tag{1}$$

Let

$$\vec{\sigma} = (\sigma_x, \sigma_y, \sigma_z) , \tag{2}$$

and

$$\sigma_a = \vec{\sigma} \cdot \vec{a} , \tag{3}$$

for any $\vec{a} \in \mathbb{R}^3$.

Define

$$P_0 = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} , \quad P_1 = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} . \tag{4}$$

P_1 is commonly called the number operator, and represented by n, whereas $P_0 = 1 - n = \overline{n}$. For any $b = (b_{N_B-1}, \ldots, b_1, b_0) \in Bool^{N_B}$, let

$$P_b = P_{b_{N_B-1}} \otimes \ldots \otimes P_{b_1} \otimes P_{b_0} \ . \tag{5}$$

We will use $\Omega(\beta)$ where Ω is a 2-dimensional matrix such as P_0, σ_x, n, etc., to represent the one-qubit operator Ω acting on qubit β. We will usually represent general qubit labels by lower case Greek letters.

Any $x \in \mathbb{R}$ can be expressed as a doubly infinite power series in powers of a base $E \in \mathbb{Z}^{>0}$: $x = \pm\sum_{\alpha=-\infty}^{\infty} x_\alpha E^\alpha$. This expansion can be represented by: $\pm(\cdots x_1 x_0 . x_{-1} x_{-2} \cdots)_{\flat E}$, which is called the base E representation of x. The plus or minus in these expressions is chosen to agree with the sign of x. It is customary to omit the subscript $\flat E$ when $E = 10$. For example, $2.25 = 2 + \frac{1}{4} = (1.01)_{\flat 2}$

Define the action of an overline placed over an $a \in Bool$ by $\overline{0} = 1$ and $\overline{1} = 0$. Call this operation, bit negation. Define the action of an oplus placed between $a, b \in Bool$ by $a \oplus b = \theta(a \neq b)$. Call this operation, bit addition. One can extend the bit negation and bit addition operations so that they can act on non-negative reals. Suppose $x = (\cdots x_1 x_0 . x_{-1} x_{-2} \cdots)_{\flat 2}$, and $y = (\cdots y_1 y_0 . y_{-1} y_{-2} \cdots)_{\flat 2}$ are non-negative real numbers. Then define the action of an overline over x so that it acts on each bit individually; i.e., so that $[\overline{x}]_\alpha = \overline{x_\alpha}$. This overline operation is sometimes called bitwise negation. Likewise, define the action of an oplus placed between x and y by $(x \oplus y)_\alpha = x_\alpha \oplus y_\alpha$. This oplus operation is sometimes called bitwise addition (without carry).

We will often use N_B to denote a number of bits, and $N_S = 2^{N_B}$ to denote the corresponding number of states. We will use the sets $Bool^{N_B}$ and $\mathbb{Z}_{0,N_S-1}$ interchangeably, since any $x \in \mathbb{Z}_{0,N_S-1}$ can be identified with its binary representation $(x_{N_B-1} \cdots x_1 x_0)_{\flat 2} \in Bool^{N_B}$.

For any $x = (x_{N_B-1} \cdots x_1 x_0)_{\flat 2} \in \mathbb{Z}_{0,N_S-1}$, define $x^R = (x_0 x_1 \cdots x_{N_B-1})_{\flat 2}$; i.e., x^R is the result of reversing the binary representation of x.

Suppose $\pi : \mathbb{Z}_{0,N_S-1} \to \mathbb{Z}_{0,N_S-1}$ is a 1-1 onto map. (We use the letter π to remind us that it is a permutation; i.e., a 1-1 onto map from a finite set onto itself). One can define a permutation matrix A with entries given by $A_{yx} = \theta(y = \pi(x))$ for all $x, y \in \mathbb{Z}_{0,N_S-1}$. (Recall that all permutation matrices A arise from permuting the columns of the unit matrix, and they satisfy $A^T A = 1$.) In this patent, we will often represent the map π and its corresponding matrix A by the same symbol π. Whether the function or the matrix is being alluded to will be clear from the context. For example, suppose L is an N_S dimension matrix, and π is a permutation on the set $\mathbb{Z}_{0,N_S-1}$. Then, it is easy to check that for all $i, j \in \mathbb{Z}_{0,N_S-1}$, $(\pi^T L)_{ij} = L_{\pi(i),j}$ and $(L\pi)_{ij} = L_{i,\pi(j)}$.

Suppose $\pi_B : \mathbb{Z}_{0,N_B-1} \to \mathbb{Z}_{0,N_B-1}$ is a 1-1 onto map (i.e., a bit permutation). π_B can be extended to a map $\pi_B : \mathbb{Z}_{0,N_S-1} \to \mathbb{Z}_{0,N_S-1}$ as follows. If $x = (x_{N_B-1} \cdots x_1 x_0)_{\flat 2} \in \mathbb{Z}_{0,N_S-1}$, then let $[\pi_B(x)]_\alpha = x_{\pi_B(\alpha)}$ for all $\alpha \in \mathbb{Z}_{0,N_B-1}$. The function $\pi_B : \mathbb{Z}_{0,N_S-1} \to \mathbb{Z}_{0,N_S-1}$ is 1-1 onto, so it can be used to define a permutation matrix of the same name. Thus, the symbol π_B will be used to refer to 3 different objects: a permutation on the set $\mathbb{Z}_{0,N_B-1}$, a permutation on the set $\mathbb{Z}_{0,N_S-1}$, and an N_S-dimensional permutation matrix. All permutations on $\mathbb{Z}_{0,N_B-1}$ generate a permutation on $\mathbb{Z}_{0,N_S-1}$, but not all permutations on $\mathbb{Z}_{0,N_S-1}$ have an underlying permutation on $\mathbb{Z}_{0,N_B-1}$.

An example of a bit permutation that will arise later is π_R; it maps $\pi_R(i) = i^R$ for all $i \in \mathbb{Z}_{0,N_S-1}$ and $\pi_R(\alpha) = N_B - 1 - \alpha$ for all $\alpha \in \mathbb{Z}_{0,N_B-1}$.

Gray Code

Next, we will review some well known facts about Gray code.

For any positive integer N_B, we define a Grayish code to be a list of the elements of $Bool^{N_B}$ such that adjacent N_B-tuples of the list differ in only one component. In other words, a Grayish code is a 1-1 onto map $\pi_{Gish} : \mathbb{Z}_{0,N_S-1} \to \mathbb{Z}_{0,N_S-1}$ such that, for all $k \in \mathbb{Z}_{0,N_S-2}$, the binary representations of $\pi_{Gish}(k)$ and $\pi_{Gish}(k+1)$ differ in only one component. For any $N_B > 1$, there are many functions π_{Gish} that satisfy this definition.

One can define a particular Grayish code that we shall refer to as "the" Gray code and denote by π_G. The Gray code can be defined recursively as follows. Let $\Gamma_0 = \emptyset$. For $N_B > 0$, let Γ_{N_B} equal the set $Bool^{N_B}$ ordered in the Gray code order. In other words, $\Gamma_{N_B} = \{\pi_G(0), \pi_G(1), \pi_G(2), \ldots, \pi_G(2^{N_B} - 1)\}_{ord}$. Then,

$$\Gamma_{N_B+1} = \{0\Gamma_{N_B}, 1\Gamma^R_{N_B}\}_{ord} \tag{6}$$

for $N_B \in \mathbb{Z}^{\geq 0}$. For example, the Gray code for $N_B = 1$ is:

$$\{0, 1\}_{ord} \ , \tag{7a}$$

the Gray code for $N_B = 2$ is:

$$\{00, 01, 11, 10\}_{ord} \ , \tag{7b}$$

the Gray code for $N_B = 3$ is:

$$\begin{array}{c} \{000, 001, 011, 010, \\ 110, 111, 101, 100\}_{ord} \end{array} . \tag{7c}$$

Suppose π_B represents a permutation on $\mathbb{Z}_{0,N_B-1}$ which generates a permutation on $\mathbb{Z}_{0,N_S-1}$ of the same name. Clearly, $\pi_B \circ \pi_G$ is a Grayish code. Indeed, $\pi_B \circ \pi_G$ is a 1-1 onto map, and permuting bits the same way for all elements of a list of Gray code preserves the property that adjacent N_B-tuples differ in only one component. (Note, however, that it is easy to find π_B's such that $\pi_G \circ \pi_B$ is not a Grayish code. Hence, to preserve Grayishness, one must apply the bit permutation after π_G, not before).

Hadamard and Walsh Matrices

Next, we will review some well-known facts about Hadamard and Walsh matrices.

For $j \in \mathbb{Z}_{0,N_S-1}$, define the "reversal" function $\pi_{R(N_B)}(j) = j^R$, and the "negation" function $\pi_{N(N_B)}(j) = \overline{j}$. The function $\pi_{G(N_B)}$ for N_B-bit Gray code has been defined previously. The functions $\pi_{R(N_B)}$, $\pi_{N(N_B)}$ and $\pi_{G(N_B)}$ are 1-1 onto so they can be used to define permutation matrices of the same name. We will often write π_R, π_N and π_G instead of $\pi_{R(N_B)}$, $\pi_{N(N_B)}$ and $\pi_{G(N_B)}$ in contexts where this does not lead to confusion. Note that π_R and π_N are symmetric matrices but π_G isn't.

For any positive integer N_B, we define the N_B-bit Hadamard matrix by

$$H_1 = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \ , \ \ H_{N_B} = H_1^{\otimes N_B} \ , \tag{8}$$

and the N_B-bit Walsh matrix by

$$W_{N_B} = H_{N_B} \pi_R \pi_G \ . \tag{9}$$

Eq.(9) implies that the N_B-bit Hadamard and Walsh matrices have the same columns, except in different orders. We will often omit the subscript N_B from H_{N_B} and W_{N_B} in contexts where doing this does not lead to confusion. Note that H and W are real symmetric matrices and the square of each of them is one.

The columns of H and W can be conveniently classified according to their constancy (See **Tuc04Dec**).

Definition of $U(2)$-Multiplexors

Next, we will define $U(2)$-multiplexors.

We define a $U(2)$-subset to be an ordered set $\{U_b : \forall b\}$ of 2-dimensional unitary matrices. Let the index b take values in a set S. In this patent, we are mostly concerned with the case that $S = Bool^{N_B-1}$.

Consider a qubit array with N_B qubits labelled $0, 1, \ldots, N_B - 1$. Suppose we choose one of these qubits to be the target, and all other qubits to be controls. Let $\vec{\kappa} = (\kappa_1, \kappa_2, \ldots, \kappa_{N_B-1})$ denote the controls and τ the target. Thus, if τ and $\vec{\kappa}$ are considered as sets, they are disjoint and their union is $\{0, 1, 2, \ldots, N_B - 1\}$. Let $\left\{U_b : \forall b \in Bool^{N_B-1}\right\}$ be a $U(2)$-subset. We will refer to any operator $\mathcal{M}$ of the following form as a $U(2)$-multiplexor:

$$\mathcal{M} = \sum_{b \in Bool^{N_B-1}} P_b(\vec{\kappa}) U_b(\tau) \ , \tag{10}$$

where $P_b(\vec{\kappa})$ acts on the Hilbert space of bits $\vec{\kappa}$ and $U_b(\tau)$ acts on that of bit τ. Note that $\mathcal{M}$ is a function of: the labels $\vec{\kappa}$ of the controls, the label τ of the target, and a $U(2)$-subset $\left\{U_b : \forall b \in Bool^{N_B-1}\right\}$. FIG.**3** shows two possible diagrammatic representations of a $U(2)$-multiplexor. The less explicit representation uses nodes such as **31** that we will call "half moon" nodes.

An example of a $U(2)$-multiplexor is the direct sum

$$\begin{aligned} \mathcal{M} &= \sum_{b \in Bool^{\eta_B}} P_b \otimes U_b & \text{(11a)} \\ &= U_{\eta_S - 1} \oplus \ldots \oplus U_2 \oplus U_1 \oplus U_0 \ , & \text{(11b)} \end{aligned}$$

where $\eta_B = N_B - 1$, $\eta_S = 2^{\eta_B}$, and the U_b are 2-dimensional unitary matrices. In fact, if we label our qubits so that qubit 0 is the target and $1, 2, 3, \ldots N_B - 1$ are the controls, then any $U(2)$-multiplexor takes the form given by Eq.(11b).

An $R_y(2)$-multiplexor is a $U(2)$-multiplexor whose $U(2)$-subset consists solely of one-qubit rotations about the Y axis (i.e., $U_b = e^{i\theta_b \sigma_y}$ for all b).

Decomposition of $R_y(2)$-Multiplexors

Tuc99 and **QbtrPat** give a method for expressing exactly ("decomposing") any $R_y(2)$-multiplexor as a SEO consisting of CNOTs and one-qubit rotations. Next, we will present a brief pictorial summary of this decomposition method.

FIGS. **4**, **5**, **6**, and **7** each shows a quantum circuit. Besides the standard circuit symbol for a CNOT, these figures use the following notation. A square gate (such as **41** in FIG.**4**) with an angle θ below the square represents $\exp(i\theta\sigma_y)$ applied at that "wire". FIGS. **4** and **5** each portrays a SEO consisting of alternating one-qubit rotations and CNOTs, with a one-qubit rotation at one end and a CNOT at the other. The angle for the one-qubit rotation that either begins or ends the SEO is denoted by $\theta_{00\dots0}$. Given two adjacent angles θ_b and $\theta_{b'}$ in the SEO, $(b)_{\flat 2}$ and $(b')_{\flat 2}$ differ only in one component, component α, where α is the position of the control bit of the CNOT that lies between the θ_b and $\theta_{b'}$ gates.

FIG.**4** shows two possible ways of decomposing an $R_y(2)$-multiplexor with one control. The decomposition (a) in FIG.**4** is equivalent to:

$$\exp\left(i\sum_{b\in Bool}\phi_b\sigma_y\otimes P_b\right) = e^{i\theta_0\sigma_y(1)}\sigma_x(1)^{n(0)}e^{i\theta_1\sigma_y(1)}\sigma_x(1)^{n(0)}\ , \tag{12}$$

where

$$\begin{bmatrix}\theta_0\\ \theta_1\end{bmatrix} = \frac{1}{2}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}\begin{bmatrix}\phi_0\\ \phi_1\end{bmatrix}\ . \tag{13}$$

FIG.**5** shows four possible ways of decomposing an $R_y(2)$-multiplexor with two controls. FIG.**5** was obtained by applying the results of FIG. **4**. The decompositions exhibited in FIG.**5** can also be expressed algebraically.

FIG. **6** (respectively, **7**) shows one of several possible decompositions of an $R_y(2)$-multiplexor with 3 (respectively, 4) controls. In general, decompositions for multiplexors with N_K controls can be obtained starting from decompositions for multiplexors with $N_K - 1$ controls.

Approximation of $U(2)$-Multiplexors

Next we will discuss one possible method for approximating $U(2)$-multiplexors.

For simplicity, we will first consider how to approximate $R_y(2)$-multiplexors. Later on, we will discuss how to approximate general $U(2)$-multiplexors.

So far we have used N_B to denote a number of bits, and $N_S = 2^{N_B}$ to denote the corresponding number of states. Below, we will use two other numbers of bits, η_B and $\eta_B{}'$, where $\eta_B = N_B - 1$ and $\eta_B{}' \leq \eta_B$. Their corresponding numbers of states will be denoted by $\eta_S = 2^{\eta_B}$ and $\eta_S{}' = 2^{\eta_B{}'}$.

Define an η_S-dimensional matrix V by

$$V = H\pi_B\pi_G \ , \tag{14}$$

where π_B is an arbitrary bit permutation on η_B bits. Eq.(14) defines a new matrix V by permuting the columns of the Hadamard matrix H. Eq.(14) is a generalization of Eq.(9). In fact, V becomes W if we specialize the bit permutation π_B to π_R. If we denote the columns of V by $\vec{v}_j$ for $j \in Z_{0,\eta_S-1}$, then

$$\vec{v}_j = \vec{h}_{\pi_B \circ \pi_G(j)} \ . \tag{15}$$

In **Tuc99**, the decomposition of an $R_y(2)$ multiplexor starts by taking the following Hadamard transform:

$$\vec{\theta} = \frac{1}{\sqrt{\eta_S}} H_{\eta_B} \vec{\phi} \ , \tag{16}$$

where $\eta_B = N_B - 1$ and $\eta_S = 2^{\eta_B}$. The vectors $\{\vec{v}_i : \forall i\}$ constitute an orthonormal basis for the space $\mathbb{R}^{\eta_S}$ in which $\vec{\phi}$ lives, so $\vec{\phi}$ can always be expanded in terms of them:

$$\vec{\phi} = \sum_{i=0}^{\eta_S-1} \vec{v}_i (\vec{v}_i^\dagger \vec{\phi}) \ . \tag{17}$$

Now suppose that we truncate this expansion, keeping only the first $\eta_S{}'$ terms, where $\eta_S{}' = 2^{\eta_B{}'}$ and $\eta_B{}' \in Z_{0,N_B-1}$. Let us call $\vec{\phi}'$ the resulting approximation to $\vec{\phi}$:

$$\vec{\phi}' = \sum_{i=0}^{\eta_S{}'-1} \vec{v}_i (\vec{v}_i^\dagger \vec{\phi}) \ . \tag{18}$$

Define $\vec{\theta}'$, an approximation to $\vec{\theta}$, as follows:

$$\vec{\theta}' = \frac{1}{\sqrt{\eta_S}} H_{\eta_B} \vec{\phi}' \ . \tag{19}$$

If we let $\{\vec{e}_i : \forall i\}$ denote the standard basis vectors, then

$$H_{\eta_B}\vec{v}_i = \begin{bmatrix} \vec{h}_0^\dagger \\ \vec{h}_1^\dagger \\ \vdots \end{bmatrix} \vec{h}_{\pi_B \circ \pi_G(i)} = \vec{e}_{\pi_B \circ \pi_G(i)} \ . \tag{20}$$

Therefore,

$$\vec{\theta'} = \frac{1}{\sqrt{\eta_S}} \sum_{i=0}^{\eta_S{}'-1} \vec{e}_{\pi_B \circ \pi_G(i)} (\vec{v}_i^\dagger \vec{\phi}) \ . \tag{21}$$

By virtue of Eq.(21), if we list the components $\{\theta'_b : \forall b\}$ of $\vec{\theta'}$ in the Grayish code order specified by the map $\pi_B \circ \pi_G$, then the items in the list at positions from $\eta_S{}'$ to the end of the list are zero. Consider, for example, FIG.**5**, which gives the exact decompositions for a multiplexor with 2 controls. Suppose that in one of those decompositions, the angles θ_b's in the second half (i.e., the half that does not contain θ_{00}) of the decomposition are all zero. Then the one-qubit rotations in the second half of the decomposition become the identity. Then the three CNOTs in the second half of the decomposition cancel each other in pairs except for one CNOT that survives. The net effect is that the decomposition for a multiplexor with 2 controls degenerates into a decomposition for a multiplexor with only 1 control. The number of control bits is reduced by one in this example. In general, we can approximate any $R_y(2)$-multiplexor by another $R_y(2)$-multiplexor (the "approximant") that has fewer controls, and, therefore, is expressible with fewer CNOTs. We will call the reduction in the number of control bits the bit deficit δ_B. Hence, $\delta_B = \eta_B - \eta_B{}'$.

The bit permutation π_B on which this approximation of a multiplexor depends can be chosen according to various criteria. If we choose $\pi_B = \pi_R$, then our approximation will keep only the higher constancy components of $\vec{\phi}$. Such a smoothing, high constancies approximation might be useful for some tasks. Similarly, if we choose $\pi_B = 1$, then our approximation will keep only the lower constancy components of $\vec{\phi}$, giving a low constancies approximation. Alternatively, we could use for π_B a bit permutation, out of all possible bit permutations on η_B bits, that minimizes the distance between the original multiplexor and its approximant. Such a dominant constancies approximation is useful if our goal is to minimize the error incurred by the approximation.

The error incurred by approximating a multiplexor can be bounded above as follows. Let $\left\{e^{i\phi_b\sigma_y} : \forall b \in Bool^{\eta_B}\right\}$ denote the $R_y(2)$-subset of an $R_y(2)$-multiplexor $\mathcal{M}_y$ and $\left\{e^{i\phi'_b\sigma_y} : \forall b \in Bool^{\eta_B}\right\}$ that of its approximant $\mathcal{M}'_y$. We will refer to $\|\mathcal{M}'_y - \mathcal{M}_y\|_2$ as the error of approximating $\mathcal{M}_y$ by $\mathcal{M}'_y$. **Tuc04Dec** shows that

$$\|\mathcal{M}'_y - \mathcal{M}_y\|_2 \leq \max_b |\phi'_b - \phi_b| = \|\vec{\phi}' - \vec{\phi}\|_\infty \ . \tag{22}$$

We will refer to $\|\vec{\phi}' - \vec{\phi}\|_\infty$ as the linearized error, to distinguish it from the error $\|\mathcal{M}'_y - \mathcal{M}_y\|_2$.

A simple picture emerges from all this. The number ν of CNOTs (ν could also be taken to be the number of some other type of elementary operation, or else, the number of control bits) that are required to express the multiplexor, and the error ϵ, are two costs that we would like to minimize. These two costs are fungible to a certain extent. Given a multiplexor $\mathcal{M}$, and an upper bound ϵ_0 on ϵ, we can find the approximant $\mathcal{M}'$ with the smallest ν. Similarly, given a multiplexor $\mathcal{M}$, and an upper bound ν_0 on ν, we can find the approximant $\mathcal{M}'$ with the smallest ϵ.

So far, we have given a method whereby one can approximate any $R_y(2)$-multiplexor $\mathcal{M}_y$ by another $R_y(2)$-multiplexor $\mathcal{M}'_y$ so that $\mathcal{M}'_y$ is expressible with fewer CNOTs than $\mathcal{M}_y$. One can extend this approximation method so that is can be used to approximate general $U(2)$-multiplexors. Here is how. Suppose a general $U(2)$-multiplexor $\mathcal{M}$ can be expressed in the form

$$\mathcal{M} = U_L \mathcal{M}_y U_R \ , \tag{23}$$

where U_L and U_R are unitary matrices and $\mathcal{M}_y$ is an $R_y(2)$-multiplexor. Then one can approximate $\mathcal{M}$ by another $U(2)$-multiplexor $\mathcal{M}'$ given by

$$\mathcal{M}' = U_L \mathcal{M}'_y U_R \ , \tag{24}$$

where $\mathcal{M}'_y$ is an $R_y(2)$-multiplexor that approximates $\mathcal{M}_y$. $\mathcal{M}'_y$ can be obtained from $\mathcal{M}_y$ using the previously described approximation method for $R_y(2)$-multiplexors.

It is always possible to expand a general $U(2)$-multiplexor $\mathcal{M}$ in the form of Eq.(23). Indeed, here are two examples. First example: If we apply the CS decomposition (**Golub96**) to a general $U(2)$-multiplexor $\mathcal{M}$, then we get Eq.(23) with

$$U_L = U_{L1} \oplus U_{L0} \ , \tag{25}$$

$$U_R = U_{R1} \oplus U_{R0} \ , \tag{26}$$

and

$$\mathcal{M}_y = \sum_{b \in Bool^{N_B-1}} e^{i\phi_b \sigma_y} \otimes P_b \ , \tag{27}$$

where U_{L1}, U_{L0}, U_{R1}, and U_{R0} are unitary matrices. Second Example: In **Tuc04Nov**, we prove that if a general $U(2)$-multiplexor $\mathcal{M}$ has a $U(2)$-subset $\{U_b : \forall b \in Bool^{N_B-1}\}$, then each U_b can be expressed as

$$U_b = e^{i\eta_b} e^{i\gamma_b \sigma_z} e^{i(\alpha_b \sigma_{s1} + \beta_b \sigma_{s2})} \sigma_w^{f(b)} \ , \tag{28}$$

where $\eta_b, \gamma_b, \alpha_b, \beta_b$ are real numbers, where (s_1, s_2, w) constitute an orthonormal basis for $\mathbb{R}^3$, and where $f(b) \in Bool$ (more precisely, $f(b) = \theta(b_\mu = 1)$, where μ is a bit position). For each b, one can always find a 2-dimensional unitary matrix V_b, a one-qubit rotation about the w axis, such that

$$V_b^\dagger e^{i(\alpha_b \sigma_{s1} + \beta_b \sigma_{s2})} V_b = e^{i\phi_b \sigma_y} \ . \tag{29}$$

Thus, $\mathcal{M}$ can be expressed in the form of Eq.(23), with

$$U_L = \pi^T D V^\dagger \ , \tag{30}$$

$$U_R = VC\pi \ , \tag{31}$$

and

$$\mathcal{M}_y = \sum_{b \in Bool^{N_B - 1}} P_b \otimes e^{i\phi_b \sigma_y} \ , \tag{32}$$

where π is a permutation matrix that relabels the qubits so that $\mathcal{M}$ becomes a direct sum of 2-dimensional unitary matrices, where D is a diagonal unitary matrix derived from the factors $e^{i\eta_b} e^{i\gamma_b \sigma_z}$, where C represents a single CNOT, and where V is a direct sum of the V_b matrices. In general, there are many ways of expanding a general $U(2)$-multiplexor $\mathcal{M}$ in the form given by Eq.(23), and we do not mean to restrict our method to any particular one.

Note that to calculate the approximant multiplexor $\mathcal{M}'$ defined by Eq.(24), it might not be necessary or advantageous to calculate all the intermediate quantities that we have introduced. Here is an example. To approximate a multiplexor whose $U(2)$-subset is given by Eq.(28), it is not necessary to calculate the rotations V_b explicitly. Instead, one can approximate the parameters α_b and β_b. Define vectors $\vec{\alpha}$ and $\vec{\beta}$ from the ordered sets $\{\alpha_b : \forall b\}$ and $\{\beta_b : \forall b\}$, respectively. Calculate an approximation $\vec{\alpha'}$ of $\vec{\alpha}$ using Eq.(18) with ϕ replaced by α. Likewise, calculate an approximation $\vec{\beta'}$ of $\vec{\beta}$. The two expansions of $\vec{\alpha'}$ and $\vec{\beta'}$ in the $\vec{v}_i$ basis can be truncated at the same η'_S. Now U_b can be approximated by replacing α_b and β_b by α'_b and β'_b. This procedure avoids calculating the V_b but is equivalent to approximating the ϕ_b defined by Eq.(29) by the ϕ'_b defined by Eq.(18).

(B)Computer Implementation of Theory

Next, we will discuss a simple computer program called "my_moo" that verifies and illustrates many of the results of this patent. "my_moo" is written in the Octave language. Octave is an interactive language and environment. Full source code of "my_moo" is given in as an Appendix to this document.

When you run “my_moo”, Octave produces two output files called “out_phis.txt” and “out_error.txt”.

A typical “out_phis.txt” file is shown in FIGS.**8** and **9**; it starts with box **81** and ends with box **91**. The corresponding “out_error.txt” file is shown in FIG.**9**, box **92**.

In this example, $N_B = 4$ so $\eta_B = 3$ and $\eta_S = 8$. The first 8 lines of “out_phis.txt” give the components of $\vec{\phi}$. In this case, the computer picked 8 independent random numbers from the unit interval, and then it sorted them in non-decreasing order. “my_moo” can be easily modified so as to allow the user himself to supply the components of $\vec{\phi}$.

After listing $\vec{\phi}$, “out_phis.txt” lists the $\eta_B!$ permutations π_B of η_B bits. For each π_B, it prints the components of $\vec{\phi}'$, listed as a row, for each value of δ_B(=row label). Note that for $\delta_B = 0$, $\vec{\phi}' = \vec{\phi}$, and for $\delta_B = \eta_B$, all ϕ'_j are equal to the average of all the components of $\vec{\phi}$. Note also that for all values of δ_B and j, one has $\phi'_j \in [\min_k(\phi_k), \max_k(\phi_k)]$.

The second output file, “out_error.txt”, gives a table of the linearized error $\|\vec{\phi}' - \vec{\phi}\|_\infty$ as a function of permutation number(=row label) and δ_B(=column label). As expected, the error is zero when δ_B is zero, and it is independent of the permutation π_B when δ_B is maximum (When the bit deficit δ_B is maximum, the approximant has no control bits, so permuting bits at positions Z_{0,η_B-1} does not affect the error.)

Note that in the above example, the last permutation minimizes the error for all δ_B. This last permutation is $\pi_B = \pi_R =$ (bit-reversal), and it gives a high constancies expansion. Recall that for this example, “my_moo” generated iid (independent, identically distributed) numbers for the components of $\vec{\phi}$, and then it rearranged them in monotonic order. Empirically, one finds that almost every time that “my_moo” is operated in the mode which generates iid numbers for the components of $\vec{\phi}$, the high constancies expansion minimizes the error for all δ_B. However, this need not always occur, as the following counterexample shows. Try running “my_moo” for $N_B = 5$, and for $\vec{\phi}$ with its first 7 components equal to 0 and its 9 subsequent components equal to 1. For this $\vec{\phi}$, and for $\delta_B = 3$, the high constancies expansion yields an error of 7/8 while some of the other expansions yield errors as low as 5/8.

Note that although "my_moo" visits all $\eta_B!$ permutations of the control bits, visiting all permutations is a very inefficient way of finding the minimum error. In fact, the $\eta_B!$ control bit permutations can be grouped into equivalence classes, such that all permutations in a class give the same error. It's clear from FIG.**2** that we only have to visit $\binom{\eta_B}{\delta_B} = \frac{\eta_B!}{\delta_B! \eta_B'!}$ (where $\eta_B' = \eta_B - \delta_B$) equivalence classes of permutations. Whereas $\eta_B! \approx \eta_B^{\eta_B} = e^{\eta_B \ln \eta_B}$ is exponential in η_B, $\binom{\eta_B}{\delta_B}$ is polynomial in η_B for two very important extremes. Namely, when δ_B or η_B' is of order one whereas η_B is very large. Indeed, if $\delta_B = 1$ or $\eta_B' = 1$, then $\binom{\eta_B}{\delta_B} = \eta_B$; if $\delta_B = 2$ or $\eta_B' = 2$, then $\binom{\eta_B}{\delta_B} = \frac{\eta_B(\eta_B-1)}{1\cdot 2}$, etc.

Those well versed in the art will have no difficulty in writing simple variants of "my_moo".

Some "my_moo" variants can be written which differ from "my_moo" in how the permutation π_B in Eq.(21) is chosen.

In Eq.(21), we sum over all $i \in S$ where $S = \mathbb{Z}_{0,\eta_{S'}-1} \subset \mathbb{Z}_{0,\eta_S-1}$. Other "my_moo" variants could be written if we modify Eq.(21) by changing S to some other subset of $\mathbb{Z}_{0,\eta_S-1}$.

Still other "my_moo" variants could approximate a general $U(2)$-multiplexor $\mathcal{M}$ using the method discussed earlier, when we discussed Eqs.(23) and (24).

FIG.**10** is a block diagram of a classical computer feeding data to a quantum computer. Box **100** represents a classical computer. It comprises sub-boxes **101, 102, 103**. Box **101** represents input devices, such as a mouse or a keyboard. Box **102** represents the CPU, internal and external memory units. Box **102** does calculations and stores information. Box **103** represents output devices, such as a printer or a display screen. Box **105** represents a quantum computer, comprising an array of quantum bits and some hardware for manipulating the state of those qubits. For more information about the organization of a present day classical computer, see **CPP** [J. Adams, S. Leestma, L. Nyhoff, "C++, an Introduction to Computing",(Prentice Hall, 1995) pages 19-20.]. A quantum compiler is a software program meant to run inside the classical computer symbolized by box **100**.

A "no-frills" preferred embodiment of this invention would be a quantum compiler that would express an input unitary matrix U_{in} as a product of unitary matrices, one of which was an $R_y(2)$-multiplexor $\mathcal{M}_y$. The quantum compiler would comprise the approximation subroutine "my_moo". "my_moo" would be invoked to approximate $\mathcal{M}_y$ by another $R_y(2)$-multiplexor $\mathcal{M}'_y$ such that $\mathcal{M}'_y$ is expressible with fewer CNOTs (or some other type of multi-qubit gate) than $\mathcal{M}_y$. There are many variants of this no-frills embodiment which those well versed in the art will be able to derive easily. The no-frills embodiment could be enhanced by using as approximation subroutine (i.e., multiplexor approximator) one of the variants of "my_moo" discussed earlier. Some of these variants of "my_moo" can approximate general $U(2)$-multiplexors instead of merely $R_y(2)$-multiplexors.

So far, we have described some exemplary preferred embodiments of this invention. Those skilled in the art will be able to come up with many modifications to the given embodiments without departing from the present invention. Thus, the inventor wishes that the scope of this invention be determined by the appended claims and their legal equivalents, rather than by the given embodiments.

APPENDIX: COMPUTER LISTING

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function my_moo
NB=4;   %NB=positive integer, number of bits
example =1;
if (NB<1)
    error("NB is less than 1");
end
len_phi = 2^(NB-1);
phi=zeros(len_phi, 1);
switch example
    case (1)  %phi in increasing order
        %rand("seed", 1.27);
        phi = sort(rand(len_phi,1));
    case (2)  %phi in decreasing order
        %rand("seed", 1.27);
        phi = flipud(sort(rand(len_phi,1)));
    case (3)
        if(len_phi!=8)
            error("this example requires NB=4");
        end
        phi=[.31; .31;.3100001; .31005; .5;.5;.5; .5];
    case (4)
        if(len_phi!=16)
            error("this example requires NB=5");
        end
        phi=[0;0;0;0;0;0;0;1;1;1;1;1;1;1;1;1];
    otherwise
        error("example number is out of range");
end
max_phi =norm(phi, Inf); % all phi components are non-negative
err_file = fopen ("out_error.txt", "w", "native");
phi_file = fopen ("out_phis.txt", "w", "native");
for i=1:len_phi
    fprintf(phi_file, "phi(%d)=\t%11.9f\n",i, phi(i));
end
fprintf(err_file, "error as function of (permutation\\delta_B)\n");
for del_B=0:(NB-1 )
    fprintf(err_file, "\t%9d", del_B);
end
fprintf(err_file, "\n");
more = false;
pi_B = (1: NB-1);
perm_num=0;
```

```
while (1)
     [ pi_B_new, more_new ] = perm_lex_next ( NB-1, pi_B, more );
    if (more_new)
        pi_B =  pi_B_new;
        more = true;
        perm_num++;
        fprintf(phi_file, "-----------------------\n");
        fprintf(phi_file, "permutation %d = (", perm_num);
        for   i = 1 : NB-2
            fprintf (phi_file, "%d,", pi_B(i) );
        end
        fprintf (phi_file, "%d)\n", pi_B(NB-1));
        fprintf(phi_file, "delta_B, phi_prime=\n");
        fprintf(err_file, "%4d", perm_num);
        for del_B=0:(NB-1 )
            phi_pr = approx_phi(phi, pi_B, del_B, NB);
            fprintf(phi_file, "%d", del_B);
            for i=1:len_phi
                comp= phi_pr(i);
                fprintf(phi_file,"\t%5.3f", comp);
            end
            fprintf(phi_file, "\n");
            err= norm(phi - phi_pr, Inf);
            fprintf(err_file, "\t%.3e", err);
        end
        fprintf(err_file, "\n");
    else
        break;
    end
end
fclose(err_file);
fclose(phi_file);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function phi_pr = approx_phi(phi, pi_B, del_B, NB)
if (NB<1)
    error("NB is less than 1");
end
len_phi = 2^(NB-1);
if (length(phi)!=len_phi)
    error("phi has wrong length");
end
if (length(pi_B)!=NB-1)
    error("pi_B has wrong length");
end
if (del_B>NB-1| del_B<0)
```

```
    error("del_B  is out of range");
end
NS_pr=2^(NB-1-del_B);
h = zeros(len_phi, 1);
h_norma = sqrt(len_phi);
phi_pr=zeros(len_phi, 1);
for j=0:(NS_pr-1)
    j1 = grayish_code(j, pi_B, NB-1);
    for i=0:(len_phi-1)
        h(i+1) = (-1)^( dec_to_bin(j1, NB-1)*
        dec_to_bin(i, NB-1)')./h_norma;
    end
    phi_pr = phi_pr + h *(h'*phi);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function gish = grayish_code(i, pi_B, NB)
if (NB<1)
    error("NB is less than 1");
end
if (i>=2^NB|  i<0)
    error("i is out of range");
end
if (length(pi_B)!=NB)
    error("pi_B has wrong length");
end
x=dec_to_bin(i, NB);
y=zeros(1, NB);
for alp=0:(NB-2)
    index = NB - alp;
    if (x(index-1)==1)
        y(index)=1-x(index);
    else
        y(index)=x(index);
    end
end
y(1)=x(1);
z=zeros(1, NB);
for index=1:NB
    z(index) = y(pi_B(index));
end
gish   =  bin_to_dec(z);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function i = bin_to_dec(x)
NB=length(x);
if (NB<1)
```

```
    error("NB is less than 1");
end
i=0;
for alp=0:(NB-1)
    i =  i + 2^alp*x(NB - alp);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function x = dec_to_bin(i, NB)
if (NB<1)
    error("NB is less than 1");
end
if (i>=2^NB | i<0)
    error("i is out of range");
end
x=zeros(1, NB);
q=i;
for alp=0:(NB-1)
    index = NB-alp;
    rem=q-2*floor(q/2); %rem=remainder
    q=floor(q/2);
    x(index)=rem;
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [ x_new, y_new ] = i_swap ( x, y )
x_new = y;
y_new = x;
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
function [ p_new, more_new ] = perm_lex_next ( n, p, more )
%I got this code from the Internet. Bob Tucci
%Reference: Mok-Kong Shen,
%Com. of the ACM, Vol. 6, Sept. 1963, page 517.
more_new = more;
if ( !more_new )
    %p_new = (1:n);
    p_new = linspace(1,n,n);
    more_new = 1;
else
    p_new(1:n) = p(1:n);
    if ( n <= 1 )
        p_new = [];
        more_new = 0;
        return;
    end
    w = n;
    while ( p_new(w) < p_new(w-1) )
```

```
        if ( w == 2 )
            more_new = 0;
            return;
        end
        w = w - 1;
    end
    u = p_new(w-1);
    for j=n:-1:w
        if ( u < p_new(j) )
            p_new(w-1) = p_new(j);
            p_new(j) = u;
            ff=floor ( ( n - w - 1 ) / 2 );
            for  k= 0 : ff
                [ p_new(n-k), p_new(w+k) ]=i_swap(
                p_new(n-k), p_new(w+k));
            end
            return;
        end
    end
end
```

I claim:

1. A method of operating a classical computer, wherein said method must be stored in the external or internal memory units of said classical computer, to calculate a sequence of operations on qubits with the purpose of applying said sequence of operations to a quantum computer to induce said quantum computer to execute a desired calculation, wherein said classical computer comprises a multiplexor approximator, wherein if said multiplexor approximator is given a prior data-set that fairly directly specifies a prior $U(2)$-multiplexor $\mathcal{M}$, then the approximator will calculate a posterior data-set that fairly directly specifies a posterior $U(2)$-multiplexor $\mathcal{M}'$, wherein $\mathcal{M}'$ approximates $\mathcal{M}$, wherein $\mathcal{M}'$ can be expressed with fewer elementary operations of a particular type than $\mathcal{M}$, said method comprising the steps of:

   storing in said classical computer an initial data-set that fairly directly specifies an $U(2)$-multiplexor $\mathcal{M}_1$, wherein $\mathcal{M}_1$ is an instance of said $\mathcal{M}$,

   applying said multiplexor approximator using as said prior $U(2)$-multiplexor the multiplexor $\mathcal{M}_1$.

2. The method of claim 1, also utilizing a quantum computer, comprising the additional step of:

   manipulating said quantum computer according to said $\mathcal{M}'$ obtained as the output of an application of said multiplexor approximator.

3. The method of claim 1, wherein said elementary operations of a particular type are CNOTs.

4. The method of claim 1, wherein said $\mathcal{M}'$ is chosen by minimization of a measure of the error incurred by approximating said $\mathcal{M}$ by said $\mathcal{M}'$, wherein said minimization is subject to a constraint which generally rules out approximating said $\mathcal{M}$ by itself.

5. The method of claim 4, wherein said error is defined in terms of $\|\mathcal{M} - \mathcal{M}'\|$, for said $\mathcal{M}$, said $\mathcal{M}'$, and a matrix norm $\| \cdot \|$.

6. The method of claim 4, wherein said constraint is an upper bound on the number, used to express said $\mathcal{M}'$, of elementary operations of a particular type.

7. The method of claim 4, wherein said constraint is an upper bound on the number, used to express said $\mathcal{M}'$, of CNOTs.

8. The method of claim 4, wherein said constraint is an upper bound on the number of bits upon which said $\mathcal{M}'$ depends.

9. The method of claim 1, wherein said $\mathcal{M}'$ is chosen by minimization of the number ν of elementary operations of a particular type which are required to express $\mathcal{M}'$, wherein said minimization is subject to an upper bound on a measure of the error incurred by approximating said $\mathcal{M}$ by said $\mathcal{M}'$.

10. The method of claim 9, wherein said ν is the number of CNOTs required to express $\mathcal{M}'$.

11. A method of operating a classical computer, wherein said method must be stored in the external or internal memory units of said classical computer, to calculate a sequence of operations on qubits with the purpose of applying said sequence of operations to a quantum computer to induce said quantum computer to execute a desired calculation, wherein said classical computer comprises a multiplexor approximator, wherein if said multiplexor approximator is given a prior data-set that fairly directly specifies a prior $U(2)$-multiplexor $\mathcal{M}$ that is expressible as $\mathcal{M} = U_L \mathcal{M}_y U_R$, wherein U_L and U_R are matrices, wherein $\mathcal{M}_y$ is an $R_y(2)$-multiplexor, then the approximator will calculate a posterior data-set that fairly directly specifies a posterior $U(2)$-multiplexor $\mathcal{M}'$ that is expressible as $\mathcal{M}' = U_L \mathcal{M}'_y U_R$, wherein $\mathcal{M}'_y$ is an $R_y(2)$-multiplexor, wherein $\mathcal{M}'_y$ approximates $\mathcal{M}_y$,

wherein $\mathcal{M}'_y$ can be expressed with fewer elementary operations of a particular type than $\mathcal{M}_y$, said method comprising the steps of:

storing in said classical computer an initial data-set that fairly directly specifies a $U(2)$-multiplexor $\mathcal{M}_1$, wherein $\mathcal{M}_1$ is an instance of said $\mathcal{M}$,

applying said multiplexor approximator using as said prior $U(2)$-multiplexor the multiplexor $\mathcal{M}_1$.

12. The method of claim 11, also utilizing a quantum computer, comprising the additional step of:

    manipulating said quantum computer according to said $\mathcal{M}'$ obtained as the output of an application of said multiplexor approximator.

13. The method of claim 11, wherein said elementary operations of a particular type are CNOTs.

14. The method of claim 11, wherein said $\mathcal{M}'_y$ is chosen by minimization of a measure of the error incurred by approximating said $\mathcal{M}$ by said $\mathcal{M}'$, wherein said minimization is subject to a constraint which generally rules out approximating said $\mathcal{M}$ by itself.

15. The method of claim 14, wherein said error is defined in terms of $\|\mathcal{M} - \mathcal{M}'\|$, for said $\mathcal{M}$, said $\mathcal{M}'$, and a matrix norm $\|\cdot\|$.

16. The method of claim 14, wherein said constraint is an upper bound on the number, used to express said $\mathcal{M}'_y$, of elementary operations of a particular type.

17. The method of claim 14, wherein said constraint is an upper bound on the number, used to express said $\mathcal{M}'_y$, of CNOTs.

18. The method of claim 14, wherein said constraint is an upper bound on the number of bits upon which said $\mathcal{M}'_y$ depends.

19. The method of claim 11, wherein said $\mathcal{M}'_y$ is chosen by minimization of the number ν of elementary operations of a particular type which are required to express $\mathcal{M}'_y$, wherein said minimization is subject to an upper bound on a measure of the error incurred by approximating said $\mathcal{M}$ by said $\mathcal{M}'$.

20. The method of claim 19, wherein said ν is the number of CNOTs required to express $\mathcal{M}'_y$.

ABSTRACT

A quantum compiler is a software program that runs on a classical computer. It can decompose ("compile") an arbitrary unitary matrix into a sequence of elementary operations (SEO) that a quantum computer can follow. A quantum compiler previously invented by Tucci decomposes an arbitrary unitary matrix U_{in} into a sequence of $U(2)$-multiplexors, each of which is then decomposed into a SEO. A preferred embodiment of this invention is a subroutine within a quantum compiler program. The subroutine approximates some or all of the intermediate $U(2)$-multiplexors whose product equals U_{in}.