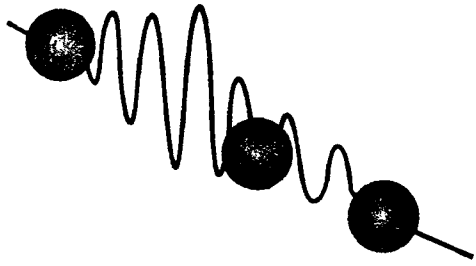
CHAPTER 4



Simulating a Simple Quantum Computer

"Feynman noted that simulating quantum systems on classical computers is hard [...] it is an exponentially difficult problem to merely record the state of a quantum system, let alone integrate its equations of motion."
— Seth Lloyd

In the last chapter, we described mathematical models of various kinds of quantum computers, ranging from quantum implementations of classical reversible Turing machines to quantum circuits and true universal quantum computers. These models have proven to be very useful in anticipating the computational capabilities of hypothetical quantum devices. Nevertheless, it can be quite hard to see beyond the equations to get an intuition of how quantum computers would actually work. To remedy this problem we describe in this chapter a *simulation* of a simple quantum computer that is available in the software supplement that comes with this book. The tools in the supplement allow you to play with the simulator and perhaps try out novel quantum computations of your own.

Of course, as Richard Feynman pointed out, we currently do not know of an efficient way to simulate a quantum computer on a classical computer[Feynman82]. In particular, as quantum computers manipulate, in general, a superposition of an exponential number of possible inputs, and any classical computer must represent each of these inputs explicitly, it is easy to see that a classical computer will

have a hard time keeping track of the complete details of a general quantum mechanical evolution. Consequently, we limit ourselves to simulating a simple quantum computer doing simple computations. Nevertheless, the simulations do provide a more intuitive appreciation of how a real quantum computer might operate.

Of the model quantum computers outlined in the previous chapter, Feynman's is the simplest to simulate as it can be specified using finite-dimensional matrices and involves only time-independent Hamiltonians. So let us take a look at the steps required to build a simulation of Feynman's quantum computer performing a particular computation[Feynman85]. In all cases, the basic sequence of steps is as follows:

1. Choose the computation that you want the computer to perform.
2. Represent that computation as a circuit built out of quantum logic gates.
3. Compute the Hamiltonian H that achieves this circuit.
4. Compute the Unitary evolution operator for H.
5. Determine the size of the memory register (to accommodate cursor + answer qubits).
6. Initialize the memory register (with input to the circuit).
7. Evolve the computer for some time.
8. Test whether the computation is done (read cursor qubits).
9. If so, extract the answer by reading the answer qubits. If not, allow further evolution (from the state projected when the cursor was last read).
10. Put it all together — bundle the previous steps into the full simulator function EvolveQC.

Let us flesh out each of these steps in the context of a fairly simple, but inherently quantum, computation.

Computation Are We Going to Simulate?

The first step is to choose the computation that you want to simulate. Feynman used binary addition with carry in his original paper. We are going to use something simpler and more intrinsically

"quantum." We choose to compute the logical NOT of a single bit, using two special quantum logic gates. You can think of the individual gates as each being "square-root-of-NOT" gates[Deutsch92a], which we denote as $\sqrt{\text{NOT}}$. The name comes from the fact that consecutive applications of two quantum logic gates can be understood mathematically as the dot product of the operators corresponding to those gates. Thus the computation that we want to simulate can be characterized as $\sqrt{\text{NOT}} \cdot \sqrt{\text{NOT}} \equiv \text{NOT}$.

What makes this computation "quantum" is the fact that it is impossible to have a single-input/single-output classical binary logic gate that works this way. Any classical binary $\sqrt{\text{NOT}}$ gate is going to have to output either a 0 or a 1 for each possible input 0 or 1. Suppose you defined the action of a classical $\sqrt{\text{NOT}}$ gate as the pair of transformations,

$$\sqrt{\text{NOT}}_{\text{classical}}(0) = 1$$
$$\sqrt{\text{NOT}}_{\text{classical}}(1) = 1$$

Then two consecutive applications of such a gate could invert a 0 successfully but not a 1. Similarly, if you defined a classical $\sqrt{\text{NOT}}$ gate as the pair of transformations,

$$\sqrt{\text{NOT}}_{\text{classical}}(0) = 1$$
$$\sqrt{\text{NOT}}_{\text{classical}}(1) = 0$$

then two consecutive applications of such a gate would not invert any input! In fact, there no way to define $\sqrt{\text{NOT}}$ classically, using binary logic, so that two consecutive applications of $\sqrt{\text{NOT}}$ reproduce the behavior of a NOT gate.

Thus, although computing NOT is a rather simple computation, the manner in which we do it illustrates several aspects of quantum computations including the use of qubits (superpositions of classical bits), the nature of quantum gates, and Feynman's method for converting a static, circuit-level, description of a computation to a dynamical Schrödinger evolution of a quantum computer that imitates the action of that circuit.

4.2 Representing a Computation as a Circuit

Once you have a particular computation in mind, the next question is how to design a quantum circuit that achieves it. To do so, you need to specify what quantum gates are to be used and how they are to be "wired" together.

We use the term "wired" rather loosely as there are no "wires," as such, at the quantum level. The "wiring" in a quantum circuit is merely a specification of which outputs (i.e., *logical* qubits) from one set of gates feed into which inputs (i.e., *logical* qubits) of another set of gates. Physically, the ports of two gates may "communicate" by either sharing the same *physical* qubit or via direct field interactions. Nevertheless, it is helpful to visualize a quantum circuit as if it were a classical combinatorial logic circuit. Thus a set of quantum gates and their associated wiring diagram define a quantum circuit.

The transformation of inputs to outputs of such circuits must be unitary because, to be realizable quantum mechanically, the circuit must cause the quantum state, that is, the input to the circuit, to evolve in accordance with Schrödinger's equation. However, Schrödinger's equation always predicts that an isolated quantum system will undergo a unitary evolution. Hence, for a hypothetical circuit to have any chance of being realizable quantum mechanically, it has to implement a unitary transformation on its inputs.

To ensure overall unitarity, this means that the quantum gates that comprise the circuit must, themselves, be unitary. However, unitarity implies reversibility. This means that the outputs of any quantum gate can be inferred, uniquely, from its inputs and vice versa. Mathematically, such gates are always describable by unitary matrices.

Just as you can pick different sets of classical logic gates in a classical circuit implementing a classical computation, so too can you pick different sets of quantum gates in a quantum circuit for a quantum computation. We now know that there are many possible choices for 2-input/2-output and 3-input/3-output universal, reversible, quantum logic gates[Barenco95a], [Barenco95]. You can build any quantum circuit out of any universal set of quantum gates. However, there are also other quantum gates, which although not forming a universal set, still provide useful (unitary) transformations for specific computations. This is the kind of gate we use in our $\sqrt{\text{NOT}} \cdot \sqrt{\text{NOT}} \equiv \text{NOT}$ circuit, namely, the square root of the NOT gate, $\sqrt{\text{NOT}}$.

Let us define the action of the $\sqrt{\text{NOT}}$ gate as follows:

$$\sqrt{\text{NOT}} \equiv \begin{pmatrix} \frac{1+i}{2} & \frac{1-i}{2} \\ \frac{1-i}{2} & \frac{1+i}{2} \end{pmatrix}$$

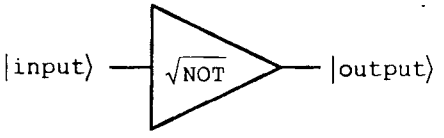


Fig. 4.1 Icon for a $\sqrt{\text{NOT}}$ gate.

This is a gate that acts on a single qubit. In a circuit diagram, we can represent the $\sqrt{\text{NOT}}$ gate as the icon shown in Fig. 4.1. Notice that there is one input and one output.

We can check that the matrix for $\sqrt{\text{NOT}}$ is unitary (and hence reversible) by looking at its product with its own conjugate transpose (denoted by †). For a unitary matrix, such a product ought to be the identity matrix.

$$\sqrt{\text{NOT}}\cdot\left(\sqrt{\text{NOT}}\right)^\dagger=\begin{pmatrix}\frac{1+i}{2} & \frac{1-i}{2}\\ \frac{1-i}{2} & \frac{1+i}{2}\end{pmatrix}\cdot\begin{pmatrix}\frac{1-i}{2} & \frac{1+i}{2}\\ \frac{1+i}{2} & \frac{1-i}{2}\end{pmatrix}=\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix}$$

So the $\sqrt{\text{NOT}}$ operator is, indeed, unitary.

You can see why we chose the name $\sqrt{\text{NOT}}$ by looking at the result of consecutive applications of this operator. We know that the NOT gate is represented by the (unitary) matrix:

$$\text{NOT}\equiv\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$$

The net effect of two $\sqrt{\text{NOT}}$ gates is given by the dot product of the matrices representing each $\sqrt{\text{NOT}}$ operation.

$$\sqrt{\text{NOT}}\cdot\sqrt{\text{NOT}}\equiv\begin{pmatrix}\frac{1+i}{2} & \frac{1-i}{2}\\ \frac{1-i}{2} & \frac{1+i}{2}\end{pmatrix}\cdot\begin{pmatrix}\frac{1+i}{2} & \frac{1-i}{2}\\ \frac{1-i}{2} & \frac{1+i}{2}\end{pmatrix}=\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}\equiv\text{NOT}$$

Thus two consecutive applications of $\sqrt{\text{NOT}}$ achieve the same affect as the NOT operation.

However, the affect of a single $\sqrt{\text{NOT}}$ operator is quite unlike any classical logic gate. Here is the truth table for a single $\sqrt{\text{NOT}}$ gate:

In[]:=

```
TruthTable[ √NOT ]
```

Out[]=

$$\text{ket}[0] \to \left(\frac{1}{2}+\frac{i}{2}\right)\text{ket}[0]+\left(\frac{1}{2}-\frac{i}{2}\right)\text{ket}[1]$$

$$\text{ket}[1] \to \left(\frac{1}{2}-\frac{i}{2}\right)\text{ket}[0]+\left(\frac{1}{2}+\frac{i}{2}\right)\text{ket}[1]$$

Notice that the $\sqrt{\text{NOT}}$ gate has the effect of taking states representing classical bits ($|0\rangle$ and $|1\rangle$) into superpositions of states representing qubits ($\omega_0|0\rangle+\omega_1|1\rangle$). Nothing like this is possible using classical digital logic gates. Thus the $\sqrt{\text{NOT}}$ gate is an inherently quantum computation. Indeed, the $\sqrt{\text{NOT}}$ computation can be put to good use in true random number generation, as we show later in the book.

The $\sqrt{\text{NOT}}$ gate need not only be given inputs that are $|0\rangle$ or $|1\rangle$ however. The truth table for $\sqrt{\text{NOT}}$ also allows us, by the linearity of quantum mechanics, to predict how the gate will transform an input that is a superposition of $|0\rangle$ and $|1\rangle$ i.e. an input of the form $\omega_0|0\rangle+\omega_1|1\rangle$. This becomes important when we connect two $\sqrt{\text{NOT}}$ gates back to back to form a circuit for NOT. The first $\sqrt{\text{NOT}}$ gate converts a bit to a qubit, but the second gate can then convert a qubit back to a bit (albeit a different bit than the original). Thus, although a single $\sqrt{\text{NOT}}$ gate behaves non-classically, two consecutive applications of $\sqrt{\text{NOT}}$ behave as classical logic. For example, here is the truth table for $\sqrt{\text{NOT}}\cdot\sqrt{\text{NOT}}$:

In[]:=

```
TruthTable[ √NOT · √NOT ]
```

Out[]=

```
ket[0] -> ket[1]
ket[1] -> ket[0]
```

which is identical to the truth table for NOT.

In[]:=

```
TruthTable[ NOT ]
```

Out[]=

```
ket[0] → ket[1]

ket[1] → ket[0]
```

Thus our quantum circuit for computing NOT consists of two consecutive applications of $\sqrt{\text{NOT}}$. In other words we build the circuit shown in Fig. 4.2.

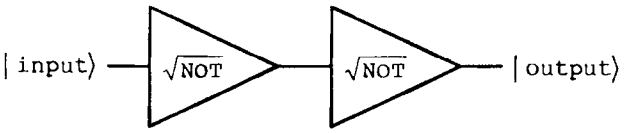


Fig. 4.2 A circuit for NOT as two $\sqrt{\text{NOT}}$ gates connected back to back.

Notice, however, that at the intermediate stage of the computation, after the action of just the first $\sqrt{\text{NOT}}$ gate, the state of the computation is a superposition of bits and hence is highly nonclassical.

4.3 Determining the Size of the Memory Register

At this point you know the computation that you want to perform and the circuit that will effect it. Next you must calculate how many qubits are needed to simulate the operation of this circuit.

The particles comprising the memory register of a (Feynman-like) quantum computer are divided into distinct sets. One set of particles is used to record the position of the cursor (which indicates how many steps of the computation have been performed), and the other set of particles is used to record the answer (or a superposition of answers) to the computation upon which the machine is working. If the computation requires the application of k logical operations (i.e., k gates) to some input pattern of m qubits, then there will be $k + 1$ qubits devoted to monitoring the cursor position and m qubits devoted to representing the answer (or superposition of answers). Thus if your circuit consists of k gate operations applied to a total of m input qubits, you will need $m + k + 1$ qubits to simulate the entire computer. Thus the complete memory register will look something like Fig. 4.3.

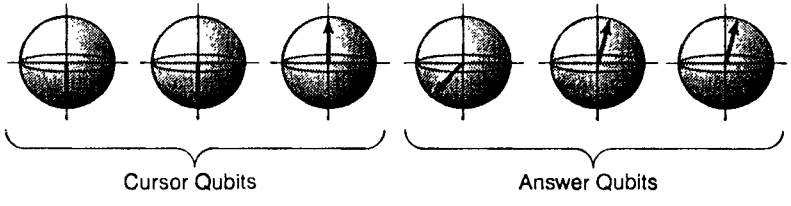


Fig. 4.3 The memory register of Feynman's quantum computer consists of a set of answer qubits that keep track of the evolving state of the computation which the circuit is performing, and a set of cursor qubits that keep track of the progress of the computation.

The states of the cursor qubits and answer qubits are coupled in the sense that, if you observe the position of the cursor and find it to be in its terminal position (i.e., at the $(k+1)$-*th* site), then you can be sure that, at that moment, if you measured the answer qubits, they would reveal a guaranteed correct solution to the computation.

So what would be the required number of cursor bits and answer bits for the "square of the square-root-of-NOT" circuit? For the $\sqrt{\text{NOT}}\cdot\sqrt{\text{NOT}}$ circuit, there is only a single input qubit (so $m = 1$), and there are two quantum gates (so $k = 2$). Hence $m + k + 1 = 4$ and we can represent the complete state of the computer (cursor qubits and answer qubits) as a $2^4 \times 1$ dimensional column vector. Thus you can think of the complete memory register for the quantum computer to be composed of 4 qubits as shown in Fig. 4.4.

Without loss of generality we can suppose that the cursor uses the first three qubits in the register and the answer uses the fourth qubit in the register. Thus our unitary evolution operator is going to have to evolve the state of all four qubits simultaneously. Unfortunately, the $\sqrt{\text{NOT}}$ operator that we constructed earlier works on only a single qubit. So we are going to have to embed our $\sqrt{\text{NOT}}$ operator in an operator that works on four qubits at a time. How do we do this? Simple; just form the direct product of the desired operator with identity operators at the "dead" positions. For example, suppose that we want the $\sqrt{\text{NOT}}$ operator that acts on the 4th of 4 qubits. Let us call this $\sqrt{\text{NOT}}[4,4]$. We form this operator as follows.

$$\sqrt{\text{NOT}}[4,4] \equiv \hat{1}\otimes\hat{1}\otimes\hat{1}\otimes\sqrt{\text{NOT}}\,.$$

You can calculate such products using the function `Direct` found in the electronic supplement. The resulting operator is the following $2^4 \times 2^4$ dimensional tridiagonal matrix:

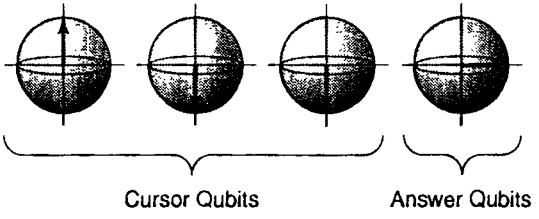


Fig. 4.4 A 4-qubit register sufficient for simulating the square of the square-root-of-NOT computation.

In[]:=

```
Direct[IdentityMatrix[2],
       IdentityMatrix[2],
       IdentityMatrix[2],
       √NOT ]
```

Out[]=

$$\begin{pmatrix}
\frac{1+i}{2} & \frac{1-i}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{1-i}{2} & \frac{1+i}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1-i}{2} & \frac{1+i}{2}
\end{pmatrix}$$

As a check that this larger operator performs the $\sqrt{\text{NOT}}$ operation on the 4th of 4 qubits, we can compute the truth table for two consecutive applications of $\sqrt{\text{NOT}}[4,4]$.

In[]:=

```
TruthTable[ √NOT [4,4] . √NOT [4,4]]
```

Out[]=

```
ket[0, 0, 0, 0] -> ket[0, 0, 0, 1]
ket[0, 0, 0, 1] -> ket[0, 0, 0, 0]
ket[0, 0, 1, 0] -> ket[0, 0, 1, 1]
ket[0, 0, 1, 1] -> ket[0, 0, 1, 0]
ket[0, 1, 0, 0] -> ket[0, 1, 0, 1]
ket[0, 1, 0, 1] -> ket[0, 1, 0, 0]
ket[0, 1, 1, 0] -> ket[0, 1, 1, 1]
ket[0, 1, 1, 1] -> ket[0, 1, 1, 0]
ket[1, 0, 0, 0] -> ket[1, 0, 0, 1]
ket[1, 0, 0, 1] -> ket[1, 0, 0, 0]
ket[1, 0, 1, 0] -> ket[1, 0, 1, 1]
ket[1, 0, 1, 1] -> ket[1, 0, 1, 0]
ket[1, 1, 0, 0] -> ket[1, 1, 0, 1]
ket[1, 1, 0, 1] -> ket[1, 1, 0, 0]
ket[1, 1, 1, 0] -> ket[1, 1, 1, 1]
ket[1, 1, 1, 1] -> ket[1, 1, 1, 0]
```

Notice that the first three qubits are mapped into themselves and the NOT operation is applied to the last qubit. Hence $\sqrt{\text{NOT}}[4,4]$ is applying the $\sqrt{\text{NOT}}$ operation to the 4th of 4 qubits.

ıting the Hamiltonian Operator

You might think that once you have the description of the quantum circuit you have everything you need to simulate the quantum computer. Certainly the circuit level description tells you *what* transformation will be effected on any given input state. However, it does not embody any *dynamics*. A real quantum computer is nothing more than a physical system whose evolution over time can be interpreted as performing some particular computation. But how you arrange for this dynamical evolution is an important issue. The circuit level description tells you what the evolution needs to look like, but you still need to embody the computation in some dynamical process. How do you do this?

Recall that the time evolution of a quantum system is described by Schrödinger's equation. For the case of a time-independent Hamiltonian $\hat{H}$, the Schrödinger equation takes the form

$$i\hbar\frac{\partial|\psi(t)\rangle}{\partial t} = \hat{H}\cdot|\psi(t)\rangle$$

where $|\psi(t)\rangle$ describes the state of the physical system, or, in our case, the state of the memory register of a quantum computer, at time t, $i=\sqrt{-1}$, and $\hbar$ is Planck's constant divided by 2π equal to 1.0545×10^{-34} Js. Thus for a quantum system to perform as a computer, or more specifically a particular quantum circuit, you need to design a Hamiltonian that will cause a given initial state of the memory register $|\psi(0)\rangle$ to evolve in a way that mimics the action of the circuit.

You can push the problem back one step by looking at the solution of Schrödinger's equation, viz.:

$$|\psi(t)\rangle = e^{-i\hat{H}t/\hbar}|\psi(0)\rangle = \hat{U}(t)|\psi(0)\rangle .$$

This implies that, in time t the initial state of the memory register $|\psi(0)\rangle$ evolves into the state $|\psi(t)\rangle$ under the action of the operator $\hat{U}(t)=\exp\left(-i\hat{H}t/\hbar\right)$. Thus if $|\psi(0)\rangle$ represents some input to the circuit we are trying to simulate, then $|\psi(t)\rangle$ represents the output from

the circuit at time t. Clearly, we would like the evolution operator $\hat{U}(t)$ to cause the state to evolve in a way that mimics the action of our desired circuit. So our task becomes that of finding a time-independent Hamiltonian operator $\hat{H}$ such that, when inserted into the matrix exponential, $\exp(-i\hat{H}t/\hbar)$ results in a unitary matrix $\hat{U}(t)$ that mimics the action of our desired circuit.

This is very difficult in general. However, physicist Richard Feynman found a way of making the computation piggyback on another dynamical system, the cursor system.

If the circuit, consisting of m inputs and m outputs, is composed of k logical operations (k gates), then Feynman augmented the input/output qubits with an additional set of $k + 1$ "cursor" qubits. The cursor serves as a kind of program step counter. If the cursor is ever found in the $(k + 1)$th site, then the memory register (of m qubits) is guaranteed to contain a valid answer to the computation at that time.

The form for the Hamiltonian that achieves this coupling between the cursor system and the computation is given by a sum of the (dot) product of creation and annihilation operators with each gate operator and the Hermitian conjugate of this. The idea came from work Feynman had done on the dynamics of spin waves traveling up and down a chain of coupled particles.

In general, a circuit will consist of a set of k gates. Its overall effect can be described by the dot product of the operators for each gate applied in sequence once the gate operators, M, have been embedded within the required number of qubits. For example, M_1 might apply an operation to the 2nd and 3rd of 4 qubits. M_2 might apply an operation to the 4th of 4 qubits, and so on.

$$M = M_k \cdot M_{k-1} \cdot \kappa \cdot M_1.$$

For the $\sqrt{\text{NOT}} \cdot \sqrt{\text{NOT}}$ circuit, there are two gates, M_1 and M_2 that can both be described by operators that act on the 4th of 4 qubits:

$$\hat{M}_2 \cdot \hat{M}_1 \equiv \sqrt{\text{NOT}}[4,4] \cdot \sqrt{\text{NOT}}[4,4].$$

The creation and annihilation operators are used to move the cursor forwards and backwards. To move the cursor from the *ith* site to the $(i + 1)$th site, we first have to annihilate the cursor at the *ith* site and then recreate it at the $(i + 1)$th site. This guarantees that there is only ever one cursor bit set at 1 amongst the cursor sites. This cursor motion must then be coupled to the corresponding gate. So the term for moving the cursor from the *ith* site to the $(i + 1)$th

site and coupling it to the application of the $(i + 1)$th gate operation is

$$c_{i+1} \cdot a_i \cdot \hat{M}_{i+1}.$$

Creation and Annihilation Operators

The creation operator c acting on a single spin state is

$$c = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix}.$$

The effect of this operator is to convert a zero-state to a 1-state and to convert a 1-state to the null state (i.e., a column vector of all zeroes).

Similarly, the annihilation operator a acting on a single spin state is

$$a = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}.$$

The annihilation operator converts a 1-state to a 0-state and converts a 0-state to the null state.

Versions of creation and annihilation operators can be created that act on the ith of four qubits using the Direct product with identity matrices at the "dead" positions that we described earlier. For example, the creation operator that acts on the second of four qubits is given by

$$c_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}.$$

The other operators are defined similarly. We use c_i and a_i to represent the creation and annihilation operators that act on the ith of k qubits.

Now define the Hamiltonian operator to be:

$$\hat{H} = \sum_{i=0}^{k-1} c_{i+1} \cdot a_i \cdot \hat{M}_{i+1} + \left(c_{i+1} \cdot a_i \cdot \hat{M}_{i+1}\right)^{\dagger},$$

where † denotes the conjugate transpose. Notice that this definition uses creation and annihilation operators to move the cursor forwards and backwards and to apply the corresponding gate operator when, and only when, the cursor is in the correct position. Thus the net effect of such a Hamiltonian is to put the quantum computer into a superposition of computations in which different numbers of gate operators have been applied. However, the position of the cur-

sor and the number of operations so far applied are perfectly correlated.

For the SqrtNOT-Squared circuit, the Hamiltonian is given by

In[]:=

```
sqrtNOTcircuit = {√NOT[4, 4], √NOT[4, 4]};
Hamiltonian[1, 2, sqrtNOTcircuit];
```

Out[]=

$$\begin{pmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} & 0 & 0 & 0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} & 0 & 0 & 0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} & 0 & 0 & 0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} & 0 & 0 & 0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1-i}{2} & \frac{1+i}{2} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1+i}{2} & \frac{1-i}{2} & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}$$

4.5 Computing the Unitary Evolution Operator

Given the preceeding form for the Hamiltonian, we can compute the evolution operator $\hat{U}(t)$ from

$$\hat{U}(t) = \left(e^{-i\,\hat{H}/h}\right)^{t}.$$

In[]:=

```
U[1, 2, sqrtNOTcircuit, t]
```

Out[]=

$$\hat{U}(t) = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \beta & 0 & \delta & \varepsilon & 0 & 0 & 0 & \gamma & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \beta & \varepsilon & \delta & 0 & 0 & \gamma & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \varepsilon & \delta & \alpha & 0 & 0 & 0 & \delta & \varepsilon & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \delta & \varepsilon & 0 & \alpha & 0 & 0 & \varepsilon & \delta & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \beta & 0 & 0 & 0 & \delta & \varepsilon & 0 & \gamma & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \beta & 0 & 0 & \varepsilon & \delta & \gamma & 0 & 0 & 0 \\ 0 & 0 & 0 & \gamma & \varepsilon & \delta & 0 & 0 & \beta & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \gamma & 0 & \delta & \varepsilon & 0 & 0 & 0 & \beta & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \varepsilon & \delta & 0 & 0 & \alpha & 0 & \delta & \varepsilon & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \delta & \varepsilon & 0 & 0 & 0 & \alpha & \varepsilon & \delta & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \gamma & 0 & 0 & \varepsilon & \delta & \beta & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \gamma & 0 & 0 & 0 & \delta & \varepsilon & 0 & \beta & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}^t$$

where

$$\alpha = \cos\left(\sqrt{2}\right),$$

$$\beta = \frac{1}{2} + \frac{\alpha}{2},$$

$$\gamma = -\frac{1}{2} + \frac{\alpha}{2},$$

$$\delta = \frac{1}{2\sqrt{2}}(1-i)\ \sin\left(\sqrt{2}\right),$$

$$\varepsilon = \frac{1}{2\sqrt{2}}(-1-i)\ \sin\left(\sqrt{2}\right).$$

ıing the Quantum Computer for a Fixed Length of Time

Once the unitary time evolution operator $\hat{U}(t)$ has been determined and a particular initial state of the memory register of the quantum computer $|\psi(0)\rangle$ has been selected, you can calculate the state of the memory register at any future time t from the equation

$$|\psi(t)\rangle = \hat{U}(t) \cdot |\psi(0)\rangle.$$

The initial state specifies a state for each qubit in the memory register. Recall that the memory register consists of two sets of qubits; the cursor qubits and the answer qubits. The initial state of the $k + 1$ cursor qubits is obligatory: the first cursor qubit must always

be set to state $|1\rangle$ and the rest to state $|0\rangle$. However, the state of the m answer qubits can be initialized to be any valid superposition of m qubits. For the square-root-of-NOT-squared circuit, there is only one answer qubit ($m = 1$) and, as the whole circuit is supposed to perform the NOT operation, it is sensible to set this answer qubit (which also serves as the input to the circuit) to either $|0\rangle$ or $|1\rangle$ initially. In which case we expect, after the computation has taken place, to observe the answer qubit and find it in state $|1\rangle$ or state $|0\rangle$, respectively.

Thus if the input to the circuit is $|0\rangle$, the initial state of the memory register will be $|1\rangle \otimes |0\rangle \otimes |0\rangle \otimes |0\rangle \equiv |1000\rangle$. Likewise, if the input to the circuit is $|1\rangle$, the initial state of the memory register will be $|1\rangle \otimes |0\rangle \otimes |0\rangle \otimes |1\rangle \equiv |1001\rangle$.

The preceding evolution equation is valid for any quantum computation, provided no observations of the state of the memory register are made in the interval $(0, t)$, the computer works perfectly, and the computer does not interact with its environment. The effects of imperfections ("internal errors") are considered in the next chapter and the effects of stray couplings to the environment ("external errors") are considered in Chapter 10. Combating errors is a crucial issue for quantum computing as it is necessary to keep the memory register in a coherent superposition of states long enough for computation to take place.

In the electronic supplement, the preceding operations, which took us from the (timeless) quantum circuit description to the (time-independent) Hamiltonian and hence the (time-dependent) unitary evolution operator, are bundled together in the function SchrodingerEvolution.

```
In[]:=
  ?SchrodingerEvolution
Out[]=
  SchrodingerEvolution[psi0, m, k, circuit, t] evolves
  the given circuit for time t from the initial
  configuration psi0 (a ket vector e.g.,
  ket[1,0,0,0]). You must specify the numbers of gates
  and inputs in your circuit. In general, if your
  circuit contains k gates, there will be k+1 cursor
  qubits. If your circuit has m inputs and m outputs,
  there will be m answer qubits.
```

The first argument to SchrodingerEvolution is the initial state of the memory register. If we specify this state as ket[1,0,0,0] the 4th bit (corresponding to the memory register bit) is set to 0, we are saying that we want to compute $\sqrt{\text{NOT}} \cdot \sqrt{\text{NOT}} \cdot |0\rangle$ (which ought to be

$\text{NOT} \cdot |0\rangle = |1\rangle$). Also, we start the cursor off at the first cursor site. As time evolves, the cursor runs back and forth along its possible positions and the probability of finding the cursor at the $(k + 1)$th cursor position rises and falls repeatedly. Let us simulate this circuit for increasing evolution times (specified in the last argument of SchrodingerEvolution). For simplicity we set h equal to 1 and we evolve the computer for time 0.5 units.

In[]:=

```
sqrtNOTcircuit = {√NOT[4, 4], √NOT[4, 4]};
evoln1 =
    SchrodingerEvolution[ket[1,0,0,0],
                         1, 2, sqrtNOTcircuit, 0.5]
```

Out[]=

```
-0.119878 ket[0, 0, 1, 1] +
(0.229681 - 0.229681 I) ket[0, 1, 0, 0] +
(-0.229681 - 0.229681 I) ket[0, 1, 0, 1] +
0.880122 ket[1, 0, 0, 0]
```

SchrodingerEvolution returned the state of the computer 0.5 time units after the initial state. At this time, the probabilities of finding the memory register (cursor bits and program bit) in each possible configuration are given by the function Probabilities.

In[]:=

```
?Probabilities
```

Out[]=

```
Probabilities[superposition] returns the
probabilities of finding a system in a state
described by superposition in each of its possible
eigenstates upon being measured (observed). If
Probabilities is given the option ShowEigenstates ->
True the function returns a list of {eigenstate,
probability} pairs.
```

In[]:=

```
Probabilities[evoln1, ShowEigenstates->True] //
ColumnForm
```

Out[]=

```
{ket[0, 0, 0, 0], 0}
{ket[0, 0, 0, 1], 0}
{ket[0, 0, 1, 0], 0}
{ket[0, 0, 1, 1], 0.0143707}
{ket[0, 1, 0, 0], 0.105507}
{ket[0, 1, 0, 1], 0.105507}
{ket[0, 1, 1, 0], 0}
{ket[0, 1, 1, 1], 0}
{ket[1, 0, 0, 0], 0.774615}
{ket[1, 0, 0, 1], 0}
{ket[1, 0, 1, 0], 0}
{ket[1, 0, 1, 1], 0}
{ket[1, 1, 0, 0], 0}
{ket[1, 1, 0, 1], 0}
{ket[1, 1, 1, 0], 0}
{ket[1, 1, 1, 1], 0}
```

On the left we see the possible states in which the memory register can be found. On the right, we see the corresponding probability of finding the register in that state at that time if (and we emphasize if) the complete register (cursor qubits and answer qubits) were observed at that time. If we were to run the simulation again for a longer period of time, we would see that the probabilities change, indicating that the odds of finding the register in one state or another change over time.

In[]:=

```
sqrtNOTcircuit = {√NOT[4, 4], √NOT[4, 4]};
evoln2 =
    SchrodingerEvolution[ket[1,0,0,0],
                         1, 2, sqrtNOTcircuit, 2]
```

Out[]=

```
-0.975682 ket[0, 0, 1, 1] +
(0.10892 - 0.10892 I) ket[0, 1, 0, 0] +
(-0.10892 - 0.10892 I) ket[0, 1, 0, 1] +
0.0243184 ket[1, 0, 0, 0]
```

In[]:=

```
Probabilities[evoln2, ShowEigenstates->True] //
ColumnForm
```

```
Out[]=
   {ket[0, 0, 0, 0], 0}
   {ket[0, 0, 0, 1], 0}
   {ket[0, 0, 1, 0], 0}
   {ket[0, 0, 1, 1], 0.951955}
   {ket[0, 1, 0, 0], 0.023727}
   {ket[0, 1, 0, 1], 0.023727}
   {ket[0, 1, 1, 0], 0}
   {ket[0, 1, 1, 1], 0}
   {ket[1, 0, 0, 0], 0.000591386}
   {ket[1, 0, 0, 1], 0}
   {ket[1, 0, 1, 0], 0}
   {ket[1, 0, 1, 1], 0}
   {ket[1, 1, 0, 0], 0}
   {ket[1, 1, 0, 1], 0}
   {ket[1, 1, 1, 0], 0}
   {ket[1, 1, 1, 1], 0}
```

The output indicates that, after the initial state has been allowed to evolve (without any measurements) for 2 time units, there is a 95% chance of finding the cursor at the $(k + 1)$th site (the third qubit set to 1 in ket[0,0,1,1]). Notice also that there is zero probability of finding the register in certain other configurations (such as ket[1,0,1,1]), indicating that certain configurations are forbidden. For example, any configuration that has more than one 1 bit amongst the cursor qubits is forbidden because there can only ever be a single 1 amongst these qubits.

As you can see, the probabilities of obtaining the correct answer rise and fall over time. In Feynman's model of a quantum computer there is uncertainty as to when the computation finishes. However, notice also that whenever the cursor is observed at the $(k + 1)$th position, the answer then in the memory register (the 4th bit) is *always* correct. Hence by periodically observing the state of the $(k + 1)$th bit, as soon as it is found to be in the 1-state, the answer is then known to be in the memory register of the computer.

It is important to understand that when the cursor position is observed, the state of the answer qubits is projected into a state consistent with the observed cursor position. However, the answer bit is not observed in this process, so it is possible for the answer qubits to remain in a superposed (but projected) state after the cursor has been observed.

4.7 Running the Quantum Computer Until the Computation Is Done

In practice, the fact that we cannot predict, with certainty, the time at which the computation will be completed, means that we must periodically observe the cursor position to see if the computation has finished. Although we do not directly observe the state of the answer bits during such observations of the cursor bits, these observations do affect the state of the whole memory register (cursor bits and answer bits). Once we find the cursor at a particular site, say the *i*th cursor site, the state of the complete register is projected into a superposition of states consistent with this observation, that is, a superposition of states in which the *i*th cursor bit is a 1.

If the cursor is found at the $(k + 1)$th site, then further evolution is prevented and the state of the complete memory register (cursor bits plus answer bits) is measured. If the cursor is not yet at the $(k + 1)$th site, the Schrödinger evolution is allowed to resume taking the new projected state as the initial state. Thus the whole computer evolves in a punctuated fashion.

To test for completion, we must measure just the qubits used to encode the cursor in the memory register. We can simulate the act of reading the cursor bits of the memory register using the function ReadCursorBits. This takes two arguments, the number of bits being used to keep track of the cursor position and the superposition of eigenstates that represents the complete state of the memory register at a given time. It returns the position of the cursor and the new state into which the memory register is projected.

In[]:=

```
?ReadCursorBits
```

Out[]=

```
ReadCursorBits[numCursorBits, superposition] reads
the state of the cursor bits of a Feynman-like quan
tum computer. If the computation can be accomplished
in k+1 (logic gate) operations, the cursor will con
sist of a chain of k+1 atoms, only one of which can
ever be in the |1> state. The cursor keeps track of
how many logical operations have been applied to the
program bits thus far. The state of the program bits
of the computer are unaffected by measuring just the
cursor bits. If the cursor is ever found at the
(k+1)th site, then, if you measured the program bits
at that moment, they would be guaranteed to contain
a valid answer to the computation on which the quan
tum computer was working.
```

For example, if the state of the complete memory register is ket[1,0,0,1] and you at that moment read the cursor position, the

answer you would obtain would be {1,0,0} (because the cursor is certainly at site 1).

In[]:=

```
ReadCursorBits[3, ket[1,0,0,1]]
```

Out[]=

```
{{1, 0, 0}, ket[1, 0, 0, 1]}
```

The function ReadCursorBits also returns the state of the register that is projected out after the measurement of the cursor position has been made. You can see this better by supposing that the memory register contains a superposition of possible computational states. For example, assume that the register is in the state 1/2 ket[1,0,0,0] - (1/4 + I/3) ket[0,1,0,0] + Sqrt[83/144] ket[0,0,1,0]. What might we get if we measure the cursor position?

In[]:=

```
ReadCursorBits[3, 1/2           ket[1,0,0,0] -
                  (1/4 + 1/3 I) ket[0,1,0,0] +
                  Sqrt[83/144]  ket[0,0,1,0]]
```

Out[]=

$$\{\{0,\ 1,\ 0\},\ (-(\frac{3}{5}) - \frac{4\ I}{5})\ \text{ket}[0,\ 1,\ 0,\ 0]\}$$

This time when we measured the cursor position we found it at the second of the three possible sites, {0,1,0}. After the measurement was made the state of the memory register was projected into the state (-3/5 - 4/5 I) ket[0,1,0,0].

Now in order to measure the cursor position, you need to measure the state of each cursor bit in turn. If you call ReadCursorBits with the option TraceProgress->True you can see the sequence of steps that takes place during this process.

In[]:=

```
ReadCursorBits[3, 1/2           ket[1,0,0,0] -
                  (1/4 + 1/3 I) ket[0,1,0,0] +
                  Sqrt[83/144]  ket[0,0,1,0],
               TraceProgress->True]
```

Out[]=

$$\{\text{BeforeAnyMeasurements},\ \frac{\text{Sqrt}[83]\,\text{ket}[0,0,1,0]}{12}+\left(-\frac{1}{4}-\frac{I}{3}\right)\text{ket}[0,1,0,0]+\frac{\text{ket}[1,0,0,0]}{2}\}$$

$$\{0,\ \frac{\text{Sqrt}\left[\frac{83}{3}\right]\text{ket}[0,0,1,0]}{6}+\left(-\frac{1}{6}-\frac{2\,I}{9}\right)\text{Sqrt}[3]\,\text{ket}[0,1,0,0]\}$$

$$\{1,\ \left(-\frac{3}{5}-\frac{4\,I}{5}\right)\text{ket}[0,1,0,0]\}$$

$$\{0,\ \left(-\frac{3}{5}-\frac{4\,I}{5}\right)\text{ket}[0,1,0,0]\}$$

"BeforeAnyMeasurements" corresponds to knowing nothing about the cursor position. At that time the state of the computer is a superposition of three states corresponding to the three possible cursor positions. Once the first cursor bit is measured we start to gain information about the cursor location. In the present example we obtain a "0" indicating that the cursor is not at the first location. A side effect of this measurement is that the state of the computer is projected into a state consistent with not finding the cursor at the first position. Next we measure the cursor to see if it is at its second position. This time we read a "1", indicating that the cursor is indeed at the second position, and project the computer into a state consistent with finding the cursor at this location. The third and final measurement is done purely to check that there is no error as there should only ever by a single "1" amongst the cursor qubits. The projected state does not change because the state was already consistent with not finding the cursor at the third location (because we already know it is at the second location). Hence the last measurement yields no new information and so the projected state after the third cursor measurement is identical to the state after the second cursor measurement.

4.8 Extracting the Answer

If the cursor is ever found at the $(k+1)$th site, then the computation is halted and the answer is read off from amongst the program bits (at sites $k+2$ through $k+1+m$). If the cursor is not at the $(k+1)$th site, then the computer is allowed to evolve again according to Schrödinger's equation for some other length of time (which may or may not be fixed) and then the cursor is again observed and the

whole cycle repeats itself. Note that although the cursor bits are observed to determine whether the machine has finished, the program bits are not observed until the very end of the computation. Thus the state of the program bits is never scrambled during the evolution of the quantum computer even though the cursor position is observed. So, in the Feynman model of a quantum computer, there is no doubt as to the correctness of the answer, merely the time at which the answer is available.

tting It All Together

Now that we know how to measure the cursor position without disturbing the program bits, we can envisage a method of evolving the quantum computer whilst periodically checking whether the computation has finished. The CD-ROM contains a simulator that allows you to trace through the steps such a quantum computer would follow. The simulator is invoked using the function EvolveQC.

In[]:=

```
?EvolveQC
```

Out[]=

```
The function EvolveQC[initState, circuit] evolves a
Feynman-like quantum computer, specified as a cir-
cuit of interconnected quantum logic gates, from
some initial state until the computation is com-
plete. The output is a list of snapshots of the
state of the QC at successive cursor-measurement
times. Each snapshot consists of a 4-element list
whose elements are the time at which the cursor is
measured, the state of the register immediately be-
fore the cursor is measured, the result of the meas-
urement, and the state of the register immediately
after the cursor position is measured. The last is
the projected state of the register. EvolveQC can
take the optional argument TimeBetweenObservations
which can be set to a number or a probability dis-
tribution. The default time between observations is
1 time unit.
```

To be concrete let us pick up the computation that we began in the last chapter, and actually see how a quantum computer would perform this calculation.

In[]:=

```
SeedRandom[13377];  (* seed to ensure reproducible
results *)
EvolveQC[ket[1,0,0,0], sqrtNOTcircuit]
```

Out[]=

```
{{0, ket[1, 0, 0, 0],
     {1, 0, 0},
     ket[1, 0, 0, 0]],

   {1, -0.422028 ket[0, 0, 1, 1] +
       (0.349228 - 0.349228 I) ket[0, 1, 0, 0] +
       (-0.349228 - 0.349228 I) ket[0, 1, 0, 1] +
       0.577972 ket[1, 0, 0, 0],

       {0, 1, 0},

       (0.5 - 0.5 I) ket[0, 1, 0, 0] +
       (-0.5 - 0.5 I) ket[0, 1, 0, 1]},

   {2, -0.698456 ket[0, 0, 1, 1] +
       (0.0779718 - 0.0779718 I) ket[0, 1, 0, 0] +
       (-0.0779718 - 0.0779718 I) ket[0, 1, 0, 1] -
       0.698456 ket[1, 0, 0, 0],

       {0, 0, 1},

       -1. ket[0, 0, 1, 1]]}
```

We can give EvolveQC the option Explain->True to produce a mor descriptive explanation of the progress of the computation.

In[]:=

```
SeedRandom[13377];  (* same seed gives same result
as above *)
EvolveQC[ket[1,0,0,0], sqrtNOTcircuit,
         Explain->True]
```

Out[]=

```
Time t=0
State of QC = ket[1, 0, 0, 0]
Cursor observed at position = {1, 0, 0}
Projected state of QC = ket[1, 0, 0, 0]
```

```
Time t=1
State of QC =              -0.422028 ket[0, 0, 1, 1] +
          (0.349228 - 0.349228 I) ket[0, 1, 0, 0] +
         (-0.349228 - 0.349228 I) ket[0, 1, 0, 1] +
                      0.577972 ket[1, 0, 0, 0]
Cursor observed at position = {0, 1, 0}
Projected state of QC =
                    (0.5 - 0.5 I) ket[0, 1, 0, 0] +
                   (-0.5 - 0.5 I) ket[0, 1, 0, 1]


Time t=2
State of QC =              -0.698456 ket[0, 0, 1, 1] +
        (0.0779718 - 0.0779718 I) ket[0, 1, 0, 0] +
       (-0.0779718 - 0.0779718 I) ket[0, 1, 0, 1] -
                      0.698456 ket[1, 0, 0, 0]
Cursor observed at position = {0, 0, 1}
Projected state of QC = -1. ket[0, 0, 1, 1]
```

If you run the same calculation multiple times, without seeding the random number generator, you will find that it is possible to get different computational results each time due to the unavoidable indeterminism of the quantum measurement process used to find the cursor position.

In[]:=

```
EvolveQC[ket[1,0,0,0], sqrtNOTcircuit,
     Explain->True]
```

Out[]=

```
Time t=0
State of QC = ket[1, 0, 0, 0]
Cursor observed at position = {1, 0, 0}
Projected state of QC = ket[1, 0, 0, 0]


Time t=1
State of QC =              -0.422028 ket[0, 0, 1, 1] +
          (0.349228 - 0.349228 I) ket[0, 1, 0, 0] +
         (-0.349228 - 0.349228 I) ket[0, 1, 0, 1] +
                      0.577972 ket[1, 0, 0, 0]
Cursor observed at position = {1, 0, 0}
Projected state of QC = 1. ket[1, 0, 0, 0]


Time t=2
State of QC =              -0.422028 ket[0, 0, 1, 1] +
          (0.349228 - 0.349228 I) ket[0, 1, 0, 0] +
         (-0.349228 - 0.349228 I) ket[0, 1, 0, 1] +
                      0.577972 ket[1, 0, 0, 0]
```

```
Cursor observed at position = {1, 0, 0}
Projected state of QC = 1. ket[1, 0, 0, 0]


Time t=3
State of QC =                -0.422028 ket[0, 0, 1, 1] +
             (0.349228 - 0.349228 I) ket[0, 1, 0, 0] +
            (-0.349228 - 0.349228 I) ket[0, 1, 0, 1] +
                          0.577972 ket[1, 0, 0, 0]
Cursor observed at position = {0, 1, 0}
Projected state of QC =
                       (0.5 - 0.5 I) ket[0, 1, 0, 0] +
                      (-0.5 - 0.5 I) ket[0, 1, 0, 1]


Time t=4
State of QC =                -0.698456 ket[0, 0, 1, 1] +
           (0.0779718 - 0.0779718 I) ket[0, 1, 0, 0] +
          (-0.0779718 - 0.0779718 I) ket[0, 1, 0, 1] -
                          0.698456 ket[1, 0, 0, 0]
Cursor observed at position = {1, 0, 0}
Projected state of QC = -1. ket[1, 0, 0, 0]


Time t=5
State of QC =                 0.422028 ket[0, 0, 1, 1] +
            (-0.349228 + 0.349228 I) ket[0, 1, 0, 0] +
             (0.349228 + 0.349228 I) ket[0, 1, 0, 1] -
                          0.577972 ket[1, 0, 0, 0]
Cursor observed at position = {0, 1, 0}
Projected state of QC =
                      (-0.5 + 0.5 I) ket[0, 1, 0, 0] +
                       (0.5 + 0.5 I) ket[0, 1, 0, 1]


Time t=6
State of QC =                 0.698456 ket[0, 0, 1, 1] +
           (-0.0779718 + 0.0779718 I) ket[0, 1, 0, 0] +
            (0.0779718 + 0.0779718 I) ket[0, 1, 0, 1] +
                          0.698456 ket[1, 0, 0, 0]
Cursor observed at position = {0, 0, 1}
Projected state of QC = 1. ket[0, 0, 1, 1]
```

Here is the output from a different run.

```
In[]:=
   EvolveQC[ket[1,0,0,0], sqrtNOTcircuit,
   Explain->True]
Out[]=
   Time t=0
   State of QC = ket[1, 0, 0, 0]
   Cursor observed at position = {1, 0, 0}
   Projected state of QC = ket[1, 0, 0, 0]


   Time t=1
   State of QC =                -0.422028 ket[0, 0, 1, 1] +
              (0.349228 - 0.349228 I) ket[0, 1, 0, 0] +
             (-0.349228 - 0.349228 I) ket[0, 1, 0, 1] +
                           0.577972 ket[1, 0, 0, 0]
   Cursor observed at position = {1, 0, 0}
   Projected state of QC = 1. ket[1, 0, 0, 0]


   Time t=2
   State of QC =                -0.422028 ket[0, 0, 1, 1] +
              (0.349228 - 0.349228 I) ket[0, 1, 0, 0] +
             (-0.349228 - 0.349228 I) ket[0, 1, 0, 1] +
                           0.577972 ket[1, 0, 0, 0]
   Cursor observed at position = {1, 0, 0}
   Projected state of QC = 1. ket[1, 0, 0, 0]


   Time t=3
   State of QC =                -0.422028 ket[0, 0, 1, 1] +
              (0.349228 - 0.349228 I) ket[0, 1, 0, 0] +
             (-0.349228 - 0.349228 I) ket[0, 1, 0, 1] +
                           0.577972 ket[1, 0, 0, 0]
   Cursor observed at position = {0, 0, 1}
   Projected state of QC = -1. ket[0, 0, 1, 1]
```

In all three cases, the correct answer was returned, but it was returned at different times. Thus in the Feynman quantum computer, the answer returned is always guaranteed to be correct but there is uncertainty as to when the answer will be available.

Graphical Illustration of the Evolution of the Memory Register

We can create a graphical illustration of how the state of the memory register evolves over time using the function PlotEvolution. This

shows the probability with which each eigenstate contributes to the superposition of states in the memory register at the successive times at which the cursor is observed. PlotEvolution takes a single input, which is the output of the function EvolveQC, and returns a graphic that shows the probability (i.e., modulus amplitude squared) of each eigenstate in the superposition of eigenstates at the times when the cursor is observed. For compactness of notation we label the eigenstates of the (in this case 4-bit) memory register, |i>, in base 10 notation; for example, |5> corresponds to the eigenstate of the memory register that is really |0101> and |15> corresponds to the eigenstate of the memory register that is really |1111>. By default, PlotEvolution shows the composition of the superposition of states in the memory register immediately before the cursor is observed (Fig. 4.5).

In[]:=

```
SeedRandom[123457];
evoln = EvolveQC[ket[1,0,0,0], sqrtNOTcircuit];
PlotEvolution[evoln];
```

Out[]=

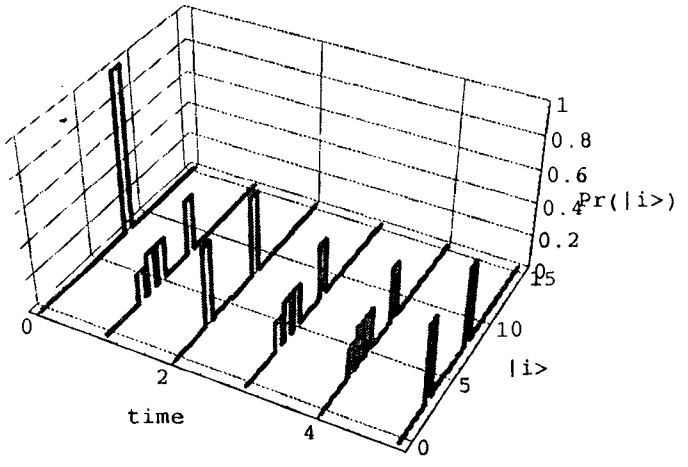


Fig. 4.5 Graphical illustration of the evolution of the state of the cursor and memory register in Feynman's quantum computer. The state, at the indicated times, is shown *before* the cursor has been read.

```
In[]:=
   PlotEvolution[evoln, AfterObservingCursor->True];
Out[]=
```

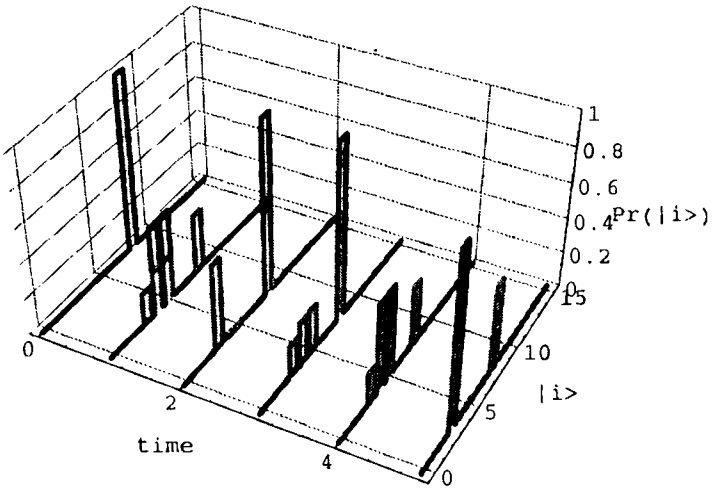


Fig. 4.6 The same evolution as shown in Fig. 4.5 with the introduction of the states the cursor and memory qubits are projected into *after* the cursor has been read.

However, by setting the option AfterObservingCursor->True you can see the effect that observing the cursor has on the composition of the superposition. The light lines in Fig. 4.6 refer to the state of the memory register immediately before the cursor position is observed and the darker lines refer to the state of the register immediately after the cursor has been observed. The projective effect of the cursor observations is quite noticeable. If you measure the cursor position too frequently, then you can sometimes prevent the system from evolving at all! Remember, the computer only evolves during the time it is not being observed (measured). If you do not let it evolve for long enough between measurements, then no appreciable probability amplitude will accumulate in those states representing the cursor anywhere other than at its initial state. In such a situation, each time you measure the cursor, you keep collapsing it back to its initial state and so the computation never advances. In the following example, we keep collapsing the cursor state back to $|100\rangle$. Eventually, we got lucky and projected the state into $|001\rangle$ (the terminal state showing that the computation was done).

In[]:=

```
longCalc = EvolveQC[ket[1,0,0,0], sqrtNOTcircuit,
                    TimeBetweenObservations->0.2];
```

The evolution recorded in longCalc reveals the following sequence of cursor positions (right column) at the indicated times (left column). The cursor result {1,0,0} means the cursor is at the first site (corresponding to state $|100\rangle$), {0,1,0} means it is at the second site (corresponding to state $|010\rangle$), and so on.

```
{0.0, {1, 0, 0}}
{0.2, {1, 0, 0}}
{0.4, {1, 0, 0}}
{0.6, {1, 0, 0}}
{0.8, {1, 0, 0}}
{1.0, {0, 1, 0}}
{1.2, {0, 1, 0}}
{1.4, {0, 1, 0}}
{1.6, {0, 1, 0}}
{1.8, {0, 1, 0}}
{2.0, {0, 1, 0}}
{2.2, {0, 1, 0}}
{2.4, {0, 1, 0}}
{2.6, {0, 1, 0}}
{2.8, {1, 0, 0}}
{3.0, {1, 0, 0}}
{3.2, {1, 0, 0}}
{3.4, {1, 0, 0}}
{3.6, {1, 0, 0}}
{3.8, {1, 0, 0}}
{4.0, {1, 0, 0}}
{4.2, {1, 0, 0}}
{4.4, {1, 0, 0}}
{4.6, {1, 0, 0}}
{4.8, {1, 0, 0}}
{5.0, {1, 0, 0}}
{5.2, {1, 0, 0}}
{5.4, {1, 0, 0}}
{5.6, {1, 0, 0}}
{5.8, {1, 0, 0}}
{6.0, {1, 0, 0}}
{6.2, {1, 0, 0}}
{6.4, {1, 0, 0}}
{6.6, {1, 0, 0}}
{6.8, {1, 0, 0}}
{7.0, {1, 0, 0}}
{7.2, {1, 0, 0}}
{7.4, {1, 0, 0}}
{7.6, {1, 0, 0}}
{7.8, {1, 0, 0}}
{8.0, {1, 0, 0}}
{8.2, {1, 0, 0}}
{8.4, {1, 0, 0}}
{8.6, {1, 0, 0}}
{8.8, {1, 0, 0}}
{9.0, {1, 0, 0}}
{9.2, {1, 0, 0}}
{9.4, {1, 0, 0}}
{9.6, {1, 0, 0}}
{9.8, {1, 0, 0}}
{10.0, {1, 0, 0}}
{10.2, {1, 0, 0}}
{10.4, {1, 0, 0}}
{10.6, {1, 0, 0}}
{10.8, {1, 0, 0}}
{11.0, {1, 0, 0}}
{11.2, {1, 0, 0}}
{11.4, {1, 0, 0}}
{11.6, {1, 0, 0}}
{11.8, {1, 0, 0}}
{12.0, {1, 0, 0}}
{12.2, {1, 0, 0}}
{12.4, {1, 0, 0}}
{12.6, {1, 0, 0}}
{12.8, {0, 1, 0}}
{13.0, {0, 0, 1}}
```