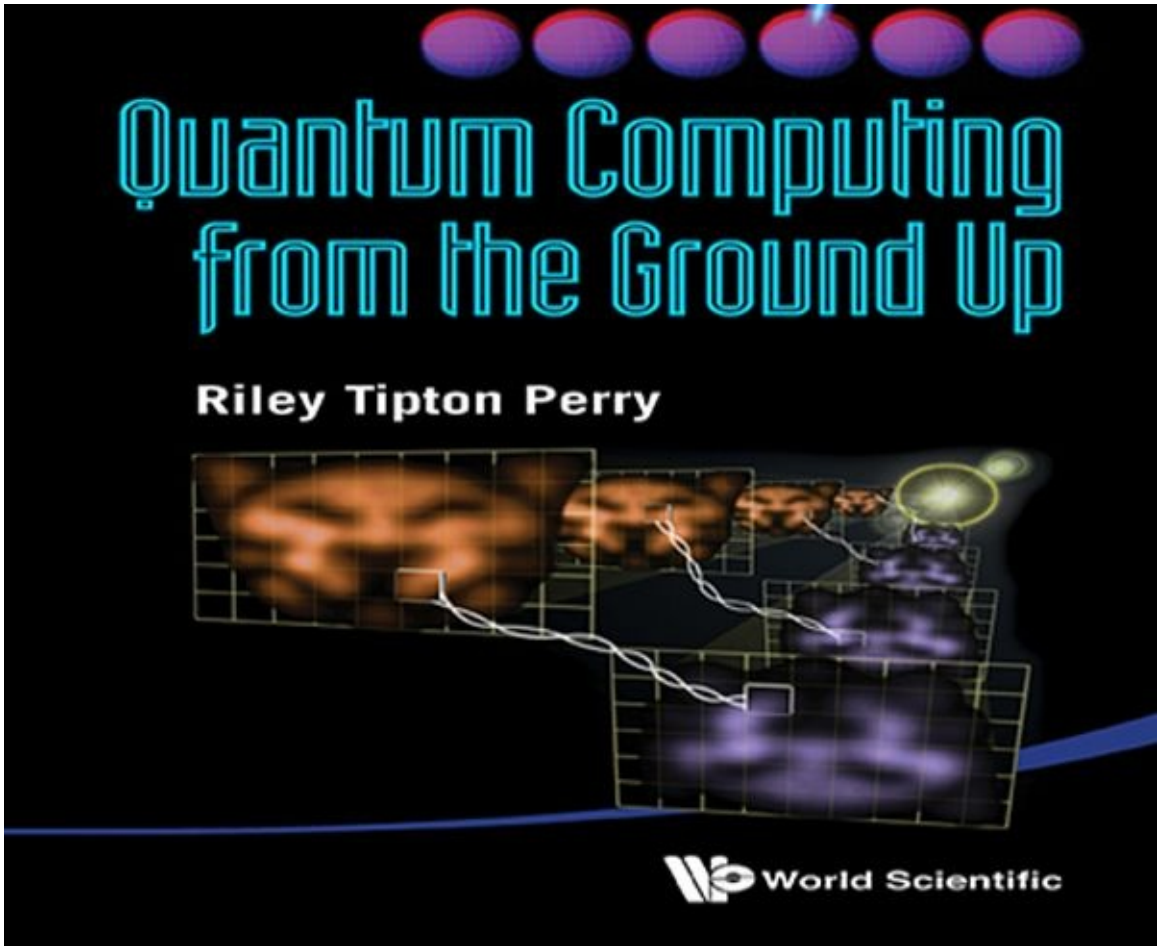
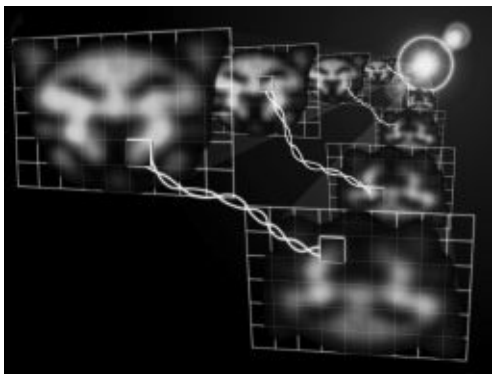

Quantum Computing
from the Ground Up
Riley Tipton Perry
World Scientific

Quantum Computing from the Ground Up

Quantum Computing from the Ground Up



Riley Tipton Perry
University of New South Wales, Australia

World Scientific
NEW JERSEY • LONDON • SINGAPORE • BEIJING • SHANGHAI • HONG KONG • TAIPEI • CHENNAI

Published by

World Scientific Publishing Co. Pte. Ltd.

5 Toh Tuck Link, Singapore 596224

USA office: 27 Warren Street, Suite 401-402, Hackensack, NJ 07601

UK office: 57 Shelton Street, Covent Garden, London WC2H 9HE

British Library Cataloguing-in-Publication Data
A catalogue record for this book is available from the British Library.

QUANTUM COMPUTING FROM THE GROUND UP

ISBN 978-981-4412-11-7 (pbk)

Printed in Singapore.

Contents

Acknowledgments

This tutorial began life as something of an open book and has had many contributors and proof readers. I would like to thank everyone who has contributed. Special thanks go to the following people:

Waranyoo Pulsawat, my dear wife. Waranyoo has been a constant source of support, companionship, and inspiration.

Brian Lederer, who is directly responsible for some parts of this text. He wrote the section on Bell states, some parts of the chapter on quantum mechanics, and a substantial amount of the other chapters. Without his help this work would never have been completed.

Mohamed Barakat and *Massoud Ghias Beygi* who have published their own version of the original book in Arabic!

Andreas Gunnarsson. Andreas' attention to detail is astonishing.

A special thanks to *Xerxes Rånby.* Xerxes had some valuable comments and found a number of errors.

All the members of the QC4Dummies Yahoo group **(http://groups.yahoo.com/group/QC4dummies/)** and administrators *David Morris* and *David Rickman.*

Sean Kaye and *Micheal Nielson* for mentioning this tutorial in their blogs.

The people at *Slashdot* **(http://slashdot.org/)** and *QubitNews* **(http://quantum.fis.ucm.es/)** for posting the tutorial for review.

James Hari, Simon Johnson, James Hollis, Nick Oosterhof, Rad Radish, Karol Bartkiewicz, Robin Kothari, Varun Vaidya, Kennedy Roulston, and Slashdotters *AC, Birdie 1013,* and *s/nemesis.*

Chapter 1

Introduction

1.1 What is Quantum Computing?

In quantum computers we exploit quantum effects to compute in ways that are faster or more efficient than, or even impossible, on conventional computers. Quantum computers use a specific physical implementation to gain a computational advantage over conventional computers. Properties called superposition and entanglement may, in some cases, allow an exponential amount of parallelism. Also, special purpose machines like quantum cryptographic devices use entanglement and other peculiarities like quantum uncertainty.

Quantum computing combines quantum mechanics, information theory, and aspects of computer science [31]. The field is a relatively new one that promises secure data transfer, dramatic computing speed increases, and may take component miniaturisation to its fundamental limit.

This text describes some of the introductory aspects of quantum computing. We'll examine some basic quantum mechanics, elementary quantum computing topics like qubits, quantum algorithms, physical realisations of those algorithms, basic concepts from computer science (like complexity theory, Turing machines, and linear algebra), information theory, and more.

1.2 Why Another Quantum Computing Tutorial?

Most of the books or papers on quantum computing require (or assume) prior knowledge of certain areas like linear algebra or physics so the majority of the current literature is hard to understand for the average computer enthusiast or interested layman. This text attempts to teach basic quantum computing from the ground up in an easily readable way. It contains a lot of the background in mathematics, physics, and computer science that you will need, although it is assumed that you know a little about computer programming.

At certain places in this document, topics that could make interesting research topics have been identified. These topics are presented in the following format:

Question *The topic is presented in bold-italics.*

1.2.1 *Quantum Computation and Quantum Information*

By far the most complete book available for quantum computing is *Quantum Computation and Quantum Information* by Michael A. Nielsen and Isaac L. Chuang, which we'll abbreviate to QCQI. The main references for this work are QCQI and a great set of lecture notes, also written by Nielsen. Nielsen's lecture notes are currently available at **http://www.qinfo.org/people/nielsen/qicss.html**. An honourable mention goes out to Vadim V. Bulitko who has managed to condense a large part of QCQI into fourteen pages!

QCQI may be a little hard to get into at first, particularly for those without a strong background in mathematics. So this tutorial is, in part, a collection of worked examples from various web sites, sets of lecture notes, journal entries, papers, and books which may aid in understanding of some of the concepts in QCQI.

Chapter 2

Computer Science

2.1 Introduction

The special properties of quantum computers force us to rethink some of the most fundamental aspects of computer science. In this chapter we'll see how quantum effects can give us a new kind of Turing machine, new kinds of circuits, and new kinds of complexity classes. This is important as it was thought that these things were not affected by what a computer is built from, but it turns out that they are.

A distinction has been made between computer science and information theory. Although information theory can be seen as a part of computer science it is treated separately in this text with its own dedicated chapter. This is because the quantum aspects of information theory require some of the concepts introduced in the chapters that follow this one.

There's also a little mathematics and notation used in this chapter which is presented in the first few sections of chapter 3 and some basic C and Javascript code for which you may need an external reference.

2.2 History

The origins of computer science can be traced back to the invention of algorithms like Euclid's Algorithm (c. 300 BC), which is an algorithm for finding the greatest common divisor of two numbers. There are also much older sources like early Babylonian cuneiform tablets (c. 2000–1700 BC) that contain clear evidence of algorithmic processes [22]. Up until the 19th century it's difficult to separate computer science from other sciences like mathematics and engineering so we'll say here that computer science began as a separate science in the 19th century.

Fig. 2.1 Charles Babbage and Ada Byron.

An important precursor to early computing machines was the punched card. The Jacquard loom, which was invented by Joseph Marie Jacquard (1752–1834) in 1801 [35] made use of complex patterns stored on punched cards to control a sequence of operations. In the mid 19th century Charles Babbage, 1791–1871 (figure 2.1) designed and partially built several programmable computing machines (see figure 2.4 for the difference engine built in 1822). These machines had many of the features of modern computers. One of these machines called the analytical engine had removable programs on punch cards based on those used in the Jacquard loom. Bab-bage's friend, Ada Augusta King, Countess of Lovelace, 1815–1852 (figure 2.1), the daughter of Lord Byron, is considered by some as the first programmer for her writings on the Analytical engine. Sadly Babbage's work was largely forgotten until the 1930s and the advent of modern computer science. Modern computer science can be said to have started in 1936 when logician Alan Turing, 1912–1954 (figure 2.2) wrote a paper which contained the notion of a *universal computer.*

Fig. 2.2 Alan Turing and Alonzo Church.

The first electronic computers were developed in the 1940s and led Jon Von Neumann, 1903–1957 (figure 2.3) to develop a generic architecture on which many modern computers are based. *Von Neumann architecture* specifies an Arithmetic Logic Unit (ALU), control unit, memory, input/output (IO), a bus, and a computing process. The architecture originated in 1945 in the first draft of a report on EDVAC [10].

Fig. 2.3 Jon Von Neumann.

Computers increased in power and versatility rapidly over the next sixty years, partly due to the development of the transistor in 1947, integrated circuits in 1959, and increasingly intuitive user interfaces. Gordon Moore proposed Moore's law in 1965, the current version of which states that processor complexity will double every eighteen months with respect to cost (in reality it's more like two years). This law still holds but is starting to falter, as components are getting smaller. Soon they will be so

small, only being made up of a few atoms [4], that quantum effects will become unavoidable, possibly ending Moore's law.

There are ways in which we can use quantum effects to our advantage in a classical sense, but by fully utilising those effects we can achieve much more. This approach is the basis for quantum computing.

2.3 Turing Machines

In 1928 David Hilbert, 1862–1943 (figure 2.5) asked if there was a universal algorithmic process to decide whether any mathematical proposition was true. His intuition suggested yes, then, in 1930 he went as far as claiming that there were no unsolvable problems in mathematics. This was promptly refuted by Kurt Gödel, 1908–1976 (figure 2.5) in 1931 by way of his *incompleteness theorem* which can be roughly summed up as follows:

Fig. 2.4 Babbage's difference engine.

> You might be able to prove every conceivable statement about numbers within a system by going outside the system in order to come up with new rules and axioms, but by doing so you'll only create a larger system with its own unprovable statements [24].

Then, in 1936 Alan Turing and Alonzo Church, 1903–1995 (figure 2.2) independently came up with models of computation, aimed at resolving whether or not mathematics contained problems that were *uncomputable*. These were problems for which there were no algorithmic solutions. An algorithm (which is a procedure for solving a mathematical problem) is guaranteed to end after a number of steps. Turing's model, now called a called a *Turing Machine* (TM), is depicted in figure 2.6. It turned out that the models of Turing and Church were equivalent in power. The thesis that any

algorithm capable of being devised can be run on a Turing machine, as Turing's model was subsequently called, was given the names of both these pioneers, the *Church-Turing thesis* [30].

Fig. 2.5 David Hilbert and Kurt Gödel.

2.3.1 *Binary Numbers and Formal Languages*

Before defining a Turing machine we need to say something about *binary numbers*, since this is usually (although not confined to) the format in which data is presented to a Turing machine (see the tape in figure 2.6).

2.3.1.1 *Binary Representation*

Computers represent numbers in binary form, as a series of zeros and ones, because this is easy to implement in hardware compared with other forms, e.g. decimal. Any information can be converted to and from zeros and ones. We call this representation a *binary representation.*

Example Here are some binary numbers and their decimal equivalents:

The binary number 1110 in decimal is 14.
Decimal number 212 when converted to binary becomes 11010100.

The binary numbers (on the left hand side) that represent the decimals 0–4 are as follows:

0 = 0
1 = 1
10 = 2
11 = 3
100 = 4

A binary number has the form $b_{n-1} \ldots b_2 b_1 b_0$ where n is the number of binary digits (or *bits*, with each digit being a 0 or a 1) and b_0 is the *least significant digit*. We can convert the binary string to a decimal number D using the following formula:

$$D = 2^{n-1}(b_{n-1}) + \ldots + 2^2(b_2) + 2^1(b_1) + 2^0(b_0). \qquad (2.1)$$

Here is another example:

Example Converting the binary number 11010100 to decimal:

$$\begin{aligned} D &= 2^7(1) + 2^6(1) + 2^5(0) + 2^4(1) + 2^3(0) + 2^2(1) + 2^1(0) + 2^0(0) \\ &= 128 + 64 + 16 + 4 \\ &= 212 \end{aligned}$$

We call the binary numbers a base 2 number system because they are based on just two symbols, 0 and 1. By contrast, in decimal (which is base 10), we have 0, 1, 2, 3, …, 9.

All data in modern computers is stored in binary format; even machine instructions are in binary format. This allows both data and instructions to be stored in computer memory and it allows all of the fundamental logical operations of the machine to be represented as binary operations.

2.3.1.2 *Formal Languages*

Turing machines and other computer science models of computation use *formal languages* to represent their inputs and outputs. We say a language L has an alphabet Σ. The language is a subset of the set Σ^* of all finite strings of symbols from Σ.

Example If $\Sigma = \{0, 1\}$ then the set of all even binary numbers {0, 10, 100, 110, …} is a langauge over Σ.

It turns out that the power of a computational model (*or automaton*) i.e. the class of algorithm that the model can implement, can be determined by considering a related question:

What type of language can the automaton recognise?

A formal language in this setting is just a set of binary strings. In simple languages the strings all follow an obvious pattern, e.g. with the language:

{01, 001, 0001, …}

the pattern is that we have one or more zeroes followed by a 1. If an automaton, when presented with any string from the language can read each symbol and then halt after the last symbol we say it recognises the language (providing it doesn't do this for strings not in the language). Then the power of the automaton is gauged by the complexity of the patterns for the languages it can recognise.

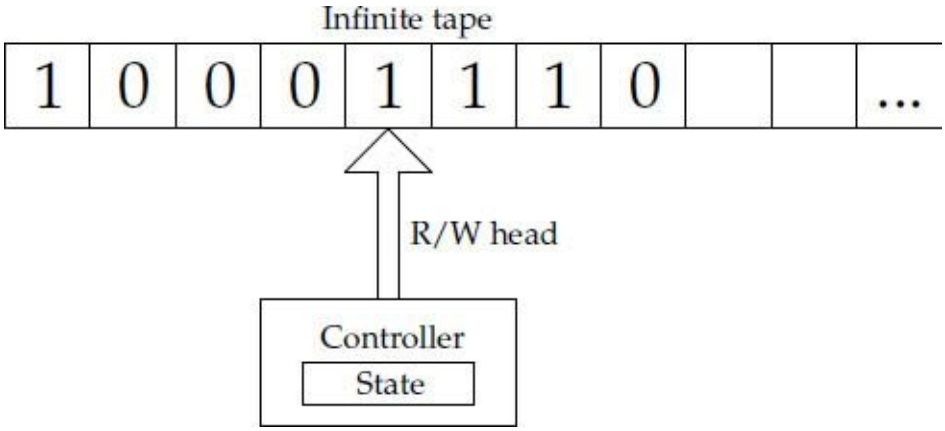


Fig. 2.6 A Turing machine.

2.3.2 Turing Machines in Action

A Turing Machine (which we'll sometimes abbreviate to TM) has the following components [12]:

(1) *A tape* — made up of cells containing 0, 1, or blank. Note that this gives us a alphabet of $\Sigma = \{0, 1, \text{blank}\}$.
(2) *A read/write head* — reads or overwrites the current symbol on each step and moves one square to the left or right.
(3) *A controller* — controls the elements of the machine to do the following:
   (i) Read the current symbol.
   (ii) Write a symbol by overwriting what's already there.
   (iii) Move the tape left or right one square.
   (iv) Change state.
   (v) Halt.
4. *The controller's behaviour* — the way the TM switches states depending on the symbol it's reading, represented by a *Finite State Automata* (FSA).

The operation of a TM is best described by a simple example:

Example Inversion inverts each input bit, for example:

$$001 \rightarrow 110$$

The behaviour of the machine can be represented by a two state FSA.
The FSA is represented below in table form (where the states are labelled 1 and 2: 1 for the running state, and 2 for the halt state).

State	Value	New State	New Value	Direction
1	0	1	1	Move Right
1	1	1	0	Move Right
1	blank	2 - HALT	blank	Move Right
2 - HALT	N/A	N/A	N/A	N/A

2.3.3 *The Universal Turing Machine*

A *Universal Turning Machine* (UTM) is a TM (with an inbuilt mechanism described by a FSA) that is capable of reading, from a tape, a program describing the behaviour of another TM. The UTM simulates the ordinary TM, performing the behaviour generated when the ordinary TM acts upon its data. When the UTM halts its tape contains the result that would have been produced by the ordinary TM (the one that describes the workings of the UTM).

The great thing about the UTM is that it shows that all algorithms (Turing machines) can be reduced to a single algorithm. As stated above, Church, Gödel, and a number of other great thinkers did find alternative ways to represent algorithms, but it was only Turing who found a way of reducing all algorithms to a single one. This reduction in algorithms is a bit like what we have in information theory where all messages can be reduced to zeroes and ones.

2.3.4 *The Halting Problem*

This is a famous problem in computer science. Having discovered the simple but powerful model of computation (essentially the stored program computer), Turing then looked at its limitations by devising a problem that it could not solve.

The UTM can run any algorithm. What, asked Turing, if we have another UTM that, rather than running a given algorithm, looks to see whether that algorithm acting on its data will actually halt in a finite number of steps (rather than looping forever or crashing)? Turing called this hypothetical new Turing machine H (for halting machine). Like a UTM, H can receive a description of the algorithm in question (its program) and the algorithm's data. Then H works on this information and produces a result. When given a number, say 1 or 0 it decides whether or not the given algorithm would then halt. Turing asked if such a machine was possible. The answer he found was no (look below). The very concept of H involves a contradiction! He demonstrated this by taking a variant of H itself as both the algorithm description (program) and data that H should work on. This proof by contradiction only applies to Turing machines and machines that are computationally equivalent. It still remains unproven that the halting problem cannot be solved in *all* computational models.

The next section contains a detailed explanation of the halting problem by means of an example. This can be skipped if you've had enough of Turing machines for now.

2.3.4.1 *The Halting Problem — Proof by Contradiction*

Here we'll look at the halting problem in Javascript [26]. The proof is by contradiction: say we could have a program that can determine whether or not another program will halt.

function Halt(program) {

```
   if (…Code to check if program can halt…) {
      return true;
   } else {
      return false;
   }
}
```

Given two programs, one that halts and one that does not:

```
function Halter(input) {
   alert('finished');
}
   function Looper(input) {
      while (1= =1) {;}
}
```

In our example Halt() would return the following:

```
Halt("function Halter(1){alert('finished');}")
     \\ returns true
Halt("function Looper(1){while (1==1) {;}}")
     \\ returns false
```

So it would be possible given these special cases, but is it possible for all algorithms to be covered in the …**Code to check if program can halt…** section? No — given a new program:

```
function Contradiction(program) {
   if (Halt(program) = = true) {
      while (1 == 1) {;}
   } else {
      alert('finished');
   }
}
```

If **Contradiction()** is given an arbitrary program as an input then:

- If **Halt()** returns true then **Contradiction()** goes into an infinite loop.
- If **Halt()** returns false then **Contradiction()** halts.

If **Halt(\url{Contradiction()})** returns true then:

- **Contradiction()** loops infinitely if **Halt(\url{Contradiction()})** halts.
- **Contradiction()** halts if **Halt(\url{Contradiction() })** goes into an infinite loop.

Contradiction() here does not loop or halt, and we can't decide algorith-mically what the behaviour of **Contradiction()** will be.

2.4 Circuits

Although modern computers are no more powerful than TM's they are a lot more efficient (for more on efficiency see section 2.5). However, what a modern or conventional computer gives in efficiency it loses in transparency (compared with a TM). It is for this reason that a TM is still of value in theoretical discussions, e.g. in comparing the hardness/difficulty of various classes of problems.

We won't go fully into the architecture of a conventional computer, but some of the concepts needed for quantum computing are related, e.g. circuits, registers, and gates. For this reason we'll examine conventional (classical) circuits.

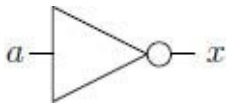
Classical circuits are read left to right and made up of the following:

(1) *Gates* — which perform logical operations on inputs. Given input(s) with values of 0 or 1 they produce an output of 0 or 1 (see below). These operations can be represented by *truth tables* which specify all of the different combinations of the outputs with respect to the inputs.
(2) *Wires* — which carry signals between gates and registers.
(3) *Registers* — which are made up of cells containing 0 or 1, i.e. bits.

2.4.1 *Common Gates*

First off we'll look at the commonly used gates. These are listed below with their respective truth tables.

The **NOT** gate inverts the input bit. The gate part of the diagram is the triangle and circle, The wires are the lines on either side of the gate and the gate reads from left to right (as with all gates and circuits).



NOT	
a	x
0	1
1	0

OR returns a 1 if either of the inputs is 1.