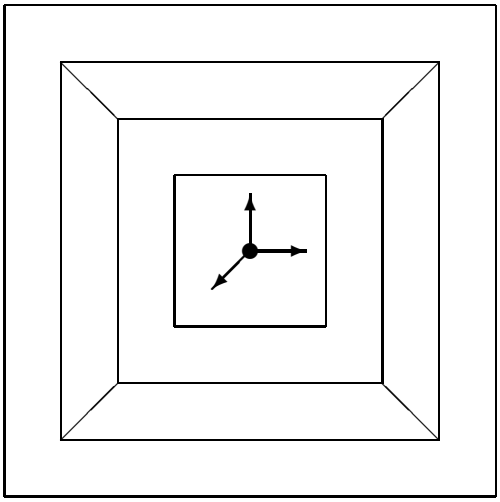
ALGORITHMIC SCIENCE AND ENGINEERING WITHIN SCIENTIFIC COMPUTING

Foundations and Organization

John Lawrence Nazareth



John Lawrence Nazareth
Professor Emeritus
Department of Mathematics
Washington State University, Pullman, WA 99164
and
Affiliate Professor
Department of Applied Mathematics
University of Washington, Seattle, WA 98195.

E-Mail: nazareth@amath.washington.edu
Web: www.math.wsu.edu/faculty/nazareth

Postal address:
Computational Decision Support Systems (CDSS)
P.O. Box 10509, Bainbridge Island, WA 98110.

This e-book is available at www.math.wsu.edu/faculty/nazareth/asebook.pdf
and can be viewed on a computer using the Adobe Acrobat Reader.

.

To the memory of my mother
Monica Isobel Freitas-Nazareth
(1917-2007)
whose gentle and determined spirit lives on
in the gardens of Kirstenbosch, South Africa

PREFACE

During the decade of the nineteen-forties, Presper Eckert and John Mauchly created the world's first electronic digital computer (ENIAC), John von Neumann formalized the underlying stored-program machine architecture that remains relevant to this day, and George Dantzig invented the simplex algorithm for linear programming, one of scientific computing's most widespread applications and an early engine of computing technology. From these beginnings, it seems no less than astonishing that within the span of a single lifetime, digital computers and the modern telecommunications network—evolving rapidly in tandem with their electronic components and the computer software that makes them usable—have permeated and now define the very fabric of our society. Nothing functions without them!

I feel lucky to have been a spectator to this revolutionary development, and also an active participant. This book is a counterpart to my previous monograph on "algorithmic science," namely, *Differentiable Optimization and Equation Solving: A Treatise on Algorithmic Science and the Karmarkar Revolution* (Springer, 2003), and my two previous monographs on "algorithmic engineering," namely, *D_LP and Extensions: An Optimization Model and Decision Support System* (Springer, 2001) and *Computer Solution of Linear Programs* (Oxford University Press, 1987). Whereas these earlier monographs considered specific families of algorithms of scientific computing and their computer implementation, our focus in the present, more concise work is on their "encompassing philosophy," i.e., on complementary issues that embrace the *foundations* and *organizational structure* of an underlying, newly-emerging discipline within scientific computing to which the name algorithmic science & engineering (AS&E) will be attached. The foregoing two italicized headings define the subtitle of our book and its two main parts. The first part contains four chapters and the second part contains three, and these two parts can be perused out of sequence, if desired. There is, however, an overall unity in our presentation, and our book's seven chapters have been written to follow a logical progression. It is therefore preferable to read them in the order given.

Briefly, Chapter 1 surveys well-known models of computation at the foundation of the "discrete" algorithms of computer science. This classical material is presented here in a novel manner that progressively drills down to the essence of a symbol-based algorithm and general-purpose digital computer. In Chapter 2, I introduce new magnitude-based computational models which

I have invented, and which lie at the foundation of the "continuous" algorithms of scientific computing. These new models are the primary contribution of this monograph. They provide a bridge to the Blum-Cucker-Shub-Smale (BCSS) model, currently the most comprehensive, idealized model for computation over the reals (and other number fields). Chapter 3 highlights Stephen Wolfram's recent discoveries concerning the complex behaviour of very simple instances of symbol-based machines introduced in Chapter 1. Chapter 4 describes a fundamental, new algorithmic paradigm for scientific computing that I have developed based on Darwinian principles, so-called population-thinking. Within Part II, Chapters 5 and 6 deal with contextual and organizational issues concerning AS&E, and, finally, Chapter 7 draws on our earlier monographs (referenced above) for illustrative AS&E case studies. *A much more detailed outline of the book* is given in the introduction that follows this preface and the table of contents.

Our book will be of interest to specialists (researchers and educators), administrators, and graduate and advanced undergraduate students, within the fields of scientific computing (including numerical analysis and optimization), computer science, and the mathematics of computation. Specialists in scientific computing often come to the subject via applied mathematics, and, in consequence, they may not be well-versed in the classical models of computation at the foundation of computer science. Likewise, computer scientists, even experts in the theoretical aspects of the field, are frequently unaware of on-going efforts to build a matching foundation for scientific computing. Our book caters simultaneously to both groups of readers by emphasizing the interplay between two basic types of models of computation—symbol-based vis-à-vis magnitude-based—that undergird computer science and scientific computing. Furthermore, we have sought, whenever feasible, to employ visual and verbal forms of presentation instead of mathematical formulae, and to make the discussion entertaining and readable—an extended essay—through the use of motivating quotations and illustrative examples. Thus *researchers and educators* interested in exploring the foundations and organization of computing will find our book to be an accessible entry into the subject. The conceptual framework and taxonomy developed in Part II will be of particular interest to *administrators*, especially at universities where the organizational structure surrounding computing is in a state of flux.

Our book is *not* intended to be a tutorial on algorithms of scientific computing or computer science, but it will be useful to students as a *supplementary reference*, for example, within a course on computing where the main textbook covers the central, technical aspects of the algorithms themselves.

It can also be recommended for purposes of self-study of the foundations and organization of algorithmic science & engineering to *graduate and advanced undergraduate students* with sufficient mathematical maturity and a background in computing. In this regard, note that our writing is concise and will appeal primarily to a student's imagination and desire for independent investigation. Details are often omitted in favor of in-text exercises, and, in some instances, challenging research projects.

Scientific and engineering problems today are often investigated by simulation on a digital computer, the modern *modus operandi* known as computational science and engineering (CSE) that complements the traditional theoretical and experimental modes. CSE relies on the tools furnished by computer science and scientific computing, and they, in turn, are premised on the foundational models of computation and organizational themes considered here. Thus our monograph will also be of interest to *computational scientists and engineers*, albeit only at the margins of their individual fields within the natural sciences and engineering.

It is a pleasure to acknowledge support from the Santa Fe Institute and stimulating interactions with members of the SFI research faculty, in particular, Professors Joe Traub, Cris Moore, David Krakauer, and Chris Wood. I've also received useful feedback and advice from Professors Beresford Parlett and Stuart Dreyfus of the University of California, Berkeley. I thank my colleagues in the Department of Mathematics at Washington State University, Pullman. During my tenure there I benefited from the department's eclectic balance between the pure and applied faces of mathematics and the three key facets of mathematical science & engineering, namely, modeling, algorithmics, and the analysis of models and algorithms. I'm grateful to Professors K.K. Tung and Nathan Kutz, the prior and current chairs of the Department of Applied Mathematics, University of Washington, Seattle, for facilitating this effort. And I thank my wife, Abbey, for her wise counsel from the perspective of the "other culture" (arts & humanities). Without her love, friendship, and joie de vivre, the many hours spent at the keyboard writing the manuscript would have been devoid of meaning.

This book is dedicated to the memory of my dear mother, who sadly did not live long enough to see its completion.

JLN
Bainbridge Island, Washington
June, 2010.

.

CONTENTS

Part II: Context and Organization

INTRODUCTION

This monograph, which addresses computing and algorithms in their broadest sense, is organized into two main parts, the first part comprising four chapters and the second part comprising three, as follows:

Part I describes foundational models and paradigms of computing.

In Chapter 1, we consider the basic concept of an algorithm and associated models of computation that form a theoretical foundation for computer science. By way of introduction, we describe a now-classic algorithm invented by E. Dijkstra for finding a shortest path in a network—it will be used subsequently for purposes of illustration. The main objective of the chapter is to present a sequence of *classical models* that *formally* capture the notion of a *symbol-based* algorithm and programmable computer: Knuth's MMIX machine; random access computational models (RAMs and RASPs); Turing-Post programs (TPPs) and universal TPPs; and, finally, Turing machines (TMs), extending to universal TMs. The first in this list is a realistic computational model. Subsequent models remove inessential detail, progressively, and represent increased levels of abstraction. The last in the sequence, the classical models of Turing and Post, reveals the very *essence* of a symbol-based algorithm and a general-purpose computer.

In Chapter 2, we consider alternative "real-number" models of computation. I introduce the new concept of *magnitude-based* computation, as contrasted with symbol-based computation of Chapter 1, by formulating an idealized model that can compute with real numbers within the restricted context of shortest-path problems on networks. This *special-purpose*, hypothetical computer, which we call a string-node (SN) machine, represents real numbers by means of analog quantities. It is motivated by an observation of Strang [1986] and exhibits an interesting connection with Dijkstra's algorithm. The primary purpose of this chapter is to present an idealized model of magnitude-based computing that I have invented, based on an extension of the random-access computational models of Chapter 1. Continuous-discrete random-access machines (CD-RAMs) and stored programs (CD-RASPs) which result from my model are designed for *general-purpose* computation over the real (and complex) numbers. They are a formalization of the "real-number model" that has been spearheaded by Traub [1999]—see also Traub and Werschulz [1998]—as a foundation for scientific computing. In the concluding section, we overview the BCSS machine model developed

by Blum, Cucker, Shub, and Smale [1998] and place it within the framework of other computational models considered above. BCSS machines are the most comprehensive, idealized formalization, to date, of computation over the reals (and other number fields), and CD-RAMs are a convenient and useful bridge to them.

In Chapter 3, we describe the *complicated dynamical behavior* of very *simple instances* of machines for symbol-based computation introduced in Chapter 1, based on discoveries of Wolfram [2002]. We consider, in turn, the behaviors of register machines (simplified RAMs); simple Turing machines; and related unidimensional mobile and cellular automata (fundamental models of parallel machine computation), including instances that are capable of universal computation.

In Chapter 4, which concludes Part I, we consider two contrasting algorithmic paradigms: the *essentialist* approach, in which the "best" member of a family of algorithms is sought, vis-à-vis the *Darwinian* approach, first proposed and studied in Nazareth [2001a], [2003], which is premised on population-thinking. These ideas are presented within a concrete and practically useful setting provided by the unconstrained nonlinear minimization problem and a classic, direct-search method of Nelder and Mead [1965].

Part II is shorter and it addresses contextual and organizational issues concerning algorithmics and computing.

Chapter 5 presents an overall, "big" picture of the disciplines embraced by the natural sciences, engineering, and mathematics, their interfaces, and their modi operandi. In particular, we create a *visual icon* that helps to clarify the nomenclature associated with these areas and their interrelated disciplines, for example, "pure" vis-à-vis "applied"; "theoretical" vis-à-vis "experimental" vis-à-vis "computational." This provides the context and sets the stage for the discussion in the subsequent two chapters of Part II.

In Chapter 6, we delineate and give a raison d'etre for the emerging discipline of algorithmic science & engineering (AS&E) that is premised on both symbol-based and magnitude-based models of computation, and undergirds the broad field of scientific computing. The latter, in conjunction with the tools furnished by computer science, provides the means whereby challenging problems within the natural sciences and engineering can be investigated via computation—the modern modus operandi known as computational science and engineering (CSE) that today complements the traditional theoretical and experimental modes. We also present a blueprint for creating informally

structured, small-scale, in-house algorithmic science & engineering research centers (ASERCs) for the scientific study of real-number algorithms at a root level, as a practical step towards the nurture of the emerging AS&E discipline.

Finally, in Chapter 7, we provide detailed illustrations of AS&E. These two case studies, within the field of optimization, typify the AS&E discipline as delineated in Chapter 6. The first case study is based on *algorithmic science* research given in Nazareth [2006b], [2003]. We describe a root-level numerical experiment with quasi-Newton algorithms for nonlinear minimization and discuss its broader implications. The second case study is premised on *algorithmic engineering* development in Nazareth [2001b], [1987]. We describe a particular resource-planning application, and use an executable demonstration program for decision-support, provided on CD-ROM in Nazareth [2001b], to present the results of simple, but realistic, planning scenarios.

.

PART I

Foundational Models and Paradigms

Chapter 1

Symbol-Based Computing

This chapter is from the e-book by J.L. Nazareth (2010), *Algorithmic Science and Engineering within Scientific Computing: Foundations and Organization*, 128 pgs., www.math.wsu.edu/faculty/nazareth/asebook.pdf .

What is an algorithm? Stated informally, it is a finitely-terminating, teleological, or goal-seeking, dynamical process that may be defined *conceptually* as a mathematical object, or realized *concretely* in the form of computer software, or even discovered *empirically* within nature. Although now inextricably linked with the electronic computer, the idea of an algorithm can be traced back to antiquity. Classical algorithms include, for example, Euclid's for finding the highest common factor of two positive integers (third century, B.C.), Euler's for finding a cycle that traverses all edges of an undirected graph, Gaussian elimination for solving a system of linear equations, and Newton's method for finding a root of a nonlinear equation. And the word "algorithm" itself is derived from the name of the great ninth century (A.D.) Persian mathematician and scholar, Al-Khowârizm.

It was only relatively recently, however, that the concept of a *general algorithm* was given its *formal* definition. This axiomatic description of computation, in the nineteen-thirties, by a group of distinguished logicians—Alonzo Church, Kurt Gödel, Stephen Kleene, A.A. Markov, Emil Post, and Alan Turing—has been likened in importance to the axiomatization of geometry by Euclid. It provides the theoretical foundation for the field of computer science and is viewed as one of the great intellectual achievements of the twentieth century. The story is well told by the popular science writer, David Berlinski [2000], who characterizes this *symbol-based* conception of an

algorithm in the following poetic manner:

> **In the logician's voice:**
> an algorithm is
> a finite procedure,
> written in a fixed symbolic vocabulary,
> governed by precise instructions,
> moving in discrete steps, 1,2,3, ...,
> whose execution requires no insight, cleverness,
> intuition, intelligence, or perspicuity,
> and that sooner or later comes to an end.

Similarly, the renowned computer scientist, Donald E. Knuth [1996, pg. 5-6], describes the basic idea underlying symbol-based computing as follows (italics ours):

> An *algorithm* is a precisely-defined sequence of *rules* telling how to produce specified output information from given input information in a finite number of steps. A particular representation of an algorithm is called a *program* $\cdots$.
>
> $\cdots$ computing machines (and algorithms) do not compute only with *numbers*. They deal with information of any kind, once it is represented in a precise way. We used to say that a sequence of symbols, such as a name, is represented inside a computer as if it were a number; but it is really more correct to say that a *number is represented inside a computer as a sequence of symbols*.

In this chapter, we will give a *concise overview* and novel perspective on several *classical models of computation* that capture *formally* the foregoing informally-stated notion of a symbol-based algorithm and general-purpose computer: Knuth's MMIX machine; random access computational models (RAMs and RASPs); Turing-Post programs (TPPs) and universal TPPs; and, finally, Turing machines (TMs) and universal TMs. But before embarking on this discussion, we will introduce the algorithmic concept in a concrete manner by describing E.W. Dijkstra's now-classic algorithm for finding a path of shortest length in a network, a problem that arises in a variety of practical applications. The algorithm is useful for illustrative purposes in our present discussion, and it plays a key role in a magnitude-based model of computation developed in Chapter 2.

1.1 Dijkstra's Algorithm

Consider an *undirected* network (N, A), where N denotes the set of nodes and A the set of arcs (pairs of nodes). A function l assigns a *nonnegative* distance, or length, to each arc in A, and l is extended to $N \times N$ by defining $l(x, x) = 0$ for the node x, and $l(x, y) = \infty$ when the corresponding arc $(x, y) \notin A$.

An appealing example of an undirected network can be found in popular road atlases of North America, for example, the widely used Rand McNally atlas, where the main arterial network of highways across the continent is presented in two different ways: a conventional map, drawn to scale, and a depiction of the road information in the form of an undirected network (N, A). The nodes of this network correspond to major cities, and its arcs, along with their associated labels, specify distances between pairs of nodes. To supplement this information, the Rand McNally atlas also provides a tabulation of the distances between pairs of cities. An entry in this table, corresponding to the pair of cities associated with row i and column j, respectively, gives the driving distance between the two cities. These tabulated entries are obtained from the undirected network of roads, (N, A), by using a shortest-path algorithm.

An efficient algorithm for finding the distance from a given source node of an undirected network, say s, to each of its other nodes—the so-called *single-source, shortest-path problem*—was discovered by E.W. Dijkstra. Here we follow the excellent accounts of this algorithm given by Strang [1986, p. 612-617] and Kozen [1992, p. 25-27].

Dijkstra's algorithm builds out from s and gets one node permanently settled, or labeled, at each iteration. Its first iteration is initialized with the node of the network nearest to s—obviously s itself with corresponding shortest distance $D(s) = l(s, s) = 0$; at the second iteration, it finds the second nearest node to s, and so on. Thus, proceeding inductively, let us assume that at iteration k of the algorithm, we know the distances to the k nearest nodes, and that they are identified as members of the subset $X \subseteq N$ with corresponding shortest distances $D(x),\ x \in X$.

A simple version of Dijkstra's algorithm[1] then computes the quantities, or distance numbers, $d(x, y) = D(x) + l(x, y)$ over all nodes $x \in X$ and $y \in Y \equiv N - X$. This yields a total of $k(n - k)$ numbers, i.e., the distances from the settled, or labeled, nodes in X, to the unlabeled ones in Y. The

[1]This is a dynamic programming (DP) approach, see Dreyfus [1965].

$(k+1)$'th nearest node to s will then be the node, say $v \in Y$, corresponding to the least of these numbers, say $d(u, v)$, assumed unique for convenience of discussion. This holds because the best path to this (next nearest) node must be attainable by a *single* step from one of the settled node, now identified as a particular $u \in X$; if it was several steps away then the intermediate nodes would be closer to s, or certainly no further, because of the assumption that arc lengths are non-negative. To complete the $(k+1)$'th iteration, the algorithm adds v to the set X and defines $D(v) = d(u, v)$. Then it begins afresh with the set of permanently settled, or labeled, nodes now containing $(k+1)$ members. The algorithm terminates when all nodes are labeled.

Observe from the foregoing description that the computation is wasteful. During the new iteration $(k+2)$, many of the quantities $d(x, y)$ that are already available, at the end of iteration $(k+1)$, are computed afresh. The only ones that need to be computed are the quantities $d(v, y), y \in N - X$, where v is the node identified and added to X at the end of iteration $(k+1)$. Organizing this calculation suitably leads to the efficient and beautifully compact form of the algorithm, which is defined by Kozen [1992, p. 26] as follows, using a pseudo-language that resembles many computer languages (we have transcribed it using our notation).

Dijkstra's Algorithm:
$X := \{s\}$;
$D(s) := 0$;
for each $y \in N - \{s\}$ do
    $D(y) := l(s, y)$
while $X \neq N$ do
    let $v \in N - X$ such that $D(v)$ is minimum;
    $X := X \bigcup \{v\}$;
    for each edge (v, y) with $y \in N - X$ do
        $D(y) := \min(D(y), D(v) + l(v, y))$;
    end for
end while

It is easy to verify the following claims that serve to establish correctness and efficiency of Dijkstra's algorithm and they are left as exercises for the interested reader:

- for any y, $D(y)$ is the distance from s to y along a shortest path through

only vertices in X;

- for any $x \in X$, $y \in N - X$, $D(x) \leq D(y)$;
- on termination, a subset of arcs that define the shortest paths from s to all other nodes—a subnetwork of (N, A)—form a tree; and
- the algorithm has polynomial-time complexity $O(n^2)$, where n denotes the number of nodes in the network.

Note that there are several variants on Dijkstra's algorithm, in particular, a straightforward extension for the case of a *directed network* where again $l(x, y) \geq 0 \; \forall (x, y)$, and an extension for a directed network where the lengths $l(x, y)$ are of arbitrary sign and subject only to the restriction that the network does not contains a negative directed cycle (Ford's algorithm). For details, see, for example, Lawler [1976] or Evans and Minieka [1992].

1.2 The MMIX Language and Computer

Dijkstra's algorithm is expressed above in a pseudo-language that can easily be translated into an existing programming language for execution on a computer. In a recent fascicle[2], Knuth [2005, pg. iv] makes the following observation concerning a suitable choice of programming language:[3]

> $\cdots$ what language should it be? In the 1960s I would probably have chosen Algol W; in the 1970s, I would then have had to rewrite $\cdots$ using Pascal; in the 1980s, I would surely have changed everything to C; in the 1990s, I would have had to switch to C^{++} and then probably to Java. In the 2000s, yet another language will no doubt be *de rigueur*.[4]

Because high-level languages go in and out of fashion, Knuth [2005] has designed instead a low-level, long-lived computer language that is "powerful enough to allow brief programs to be written for most algorithms, yet simple enough so that its operations are easily learned." It runs on a hypothetical machine MMIX that is "very much like nearly every general-purpose computer designed since 1985, except that it is, perhaps, nicer."

[2] Installment of a larger work.

[3] These remarks were made specifically with reference to the language employed within his landmark book series, Knuth [1968].

[4] In the 2010s, this language may be F$^{\#}$ or Python.

The machine's symbolic vocabulary is based on binary digits. MMIX has 2^{64} cells (8-bit bytes) of addressable *memory* and 2^8 general-purpose *registers*, each capable of holding an octabyte (8 bytes, or 64 bits). In addition, there are 2^5 special-purpose, octabyte registers. Data is transferred from the memory to the machine's registers, transformed in the registers by other instructions, and transferred back to memory from the registers.

The machine uses the RISC, or reduced instruction, architecture with 256 different *operation codes,* or commands, Instructions in MMIX have only a few formats:

1. OP X, Y, Z;
2. OP X, YZ; or
3. OP XYZ;

where OP denotes the operation code, and the remaining quantities denote upto three operands. These operands identify registers of the machine holding data for the instruction and sometimes specify data for direct use within the instruction, i.e., X,Y, and Z identify registers, and YZ and XYZ denote operands that are used directly. The set of instructions covers loading and storing, arithmetic operations, conditional operations, bitwise and bytewise operations, floating-point operations, jumps and branches, etc. Input/output primitives permit the transfer of data between input or output files and memory (via special registers). Their detailed description is not needed here.

The extension of MMIX to a low-level assembly language MMIXAL permits alphabetic names to stand for numbers, and a label field to associate names with memory locations and register numbers. The MMIX language is transparent to its underlying machine and resides at the *opposite* end of the spectrum from high-level programming languages, which shield the user from the computer on which they are implemented and can rapidly become outdated.

MMIX and MMIXAL serve as a powerful and *realistic* model of computation and a means for expressing algorithmic ideas in a precise way. In this setting, *an algorithm takes the form of a legitimate MMIX program*, which is executed in a linear stream under the control of a location counter. The location counter identifies the program instruction to be executed, the "current" instruction. If this is is a branch, or jump, instruction, its execution causes the location counter to be updated so that it points to the chosen instruction

of the program. Otherwise, after execution of the current instruction, the location counter is incremented by 1 in order to identify the next executed instruction in the linear sequence.

An MMIX program (and its associated location counter), the embodiment of an algorithm, is assumed so far to be executed by an unspecified "outside agency." A key idea underlying general-purpose computation is that *MMIX instructions can themselves be held in memory*, and modified within it, if desired. Each instruction is thus defined to be 32-bits (4 bytes) long, where the first of these four 8-bit bytes defines the *operation code*—hence the fact that there are 2^8, or 256, choices—and the remaining three bytes define (upto three) operands, the aforementioned quantities X, Y, Z, YZ, and XYZ. Again, the details need not concern us here. A "fixed hardwired program"—the so-called central processing unit, or CPU, of the machine that corresponds to the "outside agency" mentioned above—incorporates the location counter, the hardware for executing floating-point operations, additional registers, the machine "clock," and so on. The CPU fetches a programmed algorithm's instructions from memory as described earlier, in a linear stream unless a branch is encountered, and decodes and executes each instruction in turn. This is the standard *fetch-execute* cycle of a modern general-purpose, or *universal*, computer.

We will not employ MMIX to describe algorithms here. Rather, it is the *existence* of MMIX as *a realistic programming standard*, or model for machine computation, along with the fact that MMIX captures the key distinction between "algorithm" and "universal computer," that is central to our present discussion. The algorithm vis-à-vis universal computer distinction, an underlying theme of the present section, will repeat itself within each major computational model discussed below.

1.3 Random Access Computational Models

Random access machines (RAMs) and random access stored program (RASP) machines are *idealizations*, or abstractions, of modern computing machines and realistic computational models such as MMIX. The essence of this abstraction involves the following:

- simplifying the addressable memory and registers while simultaneously removing restrictions on their size, or capacity;
- reducing the instruction set; and

- making explicit the distinction between "algorithm" and "universal computer."

1.3.1 Random Access Machines (RAMs)

A detailed formulation of a RAM can be found in Aho, Hopcroft, and Ullman [1974, pgs. 5-14]. Its main components, which are are summarized in Figure 1.1, consist of an addressable *memory*, a read-only *input tape* with associated read head, a write-only *output tape* with associated write head, and a *program* with associated location counter.

The memory is an unbounded set of cells, or 'words,' numbered 1, 2, 3, ..., which define their addresses. Each cell can hold a positive or negative integer, defined by a binary pattern of 0s and 1s, of *arbitrary* size. The relatively large number of registers in MMIX is condensed into a single register, called the *accumulator*. It too can hold an arbitrary integer, or bit pattern, and all computation takes place within this accumulator.

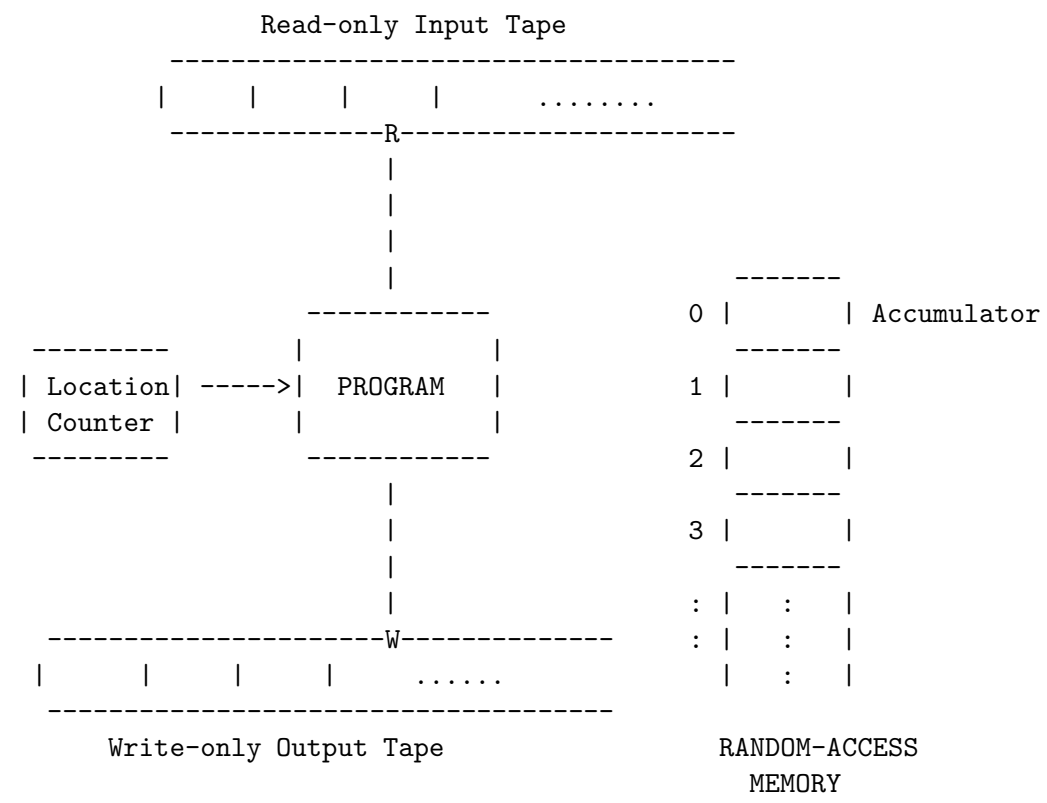
The accumulator corresponds to address 0 of the memory. Cells with addresses 1, 2, ..., are also called machine registers in the RAM model, i.e., the accumulator is simply the first cell of the addressable memory. It is distinguished from them in terms of its function, as will be described in more detail below. Henceforth, the content of register i will be represented by $c(i)$ for $i = 0, 1, 2, \ldots$.

Computation in a RAM is determined by a *program*, which is *not* stored in memory and cannot modify itself. This program consists of a sequence of (optionally) labeled instructions chosen from the list shown in the lower part of Figure 1.1, and they fall into five main groups: load/store between the accumulator and memory cells; arithmetic between the accumulator and memory cells; read/write between input/output tapes and the accumulator; unconditional or conditional jumps to a labeled instruction of the program; stop. Each instruction consists of two fields—an 'operation code' and an address field containing an 'operand'—and there are three choices for the latter:

1. an integer, say i, directly specified,

2. the contents of the word of memory with an address defined by i, i.e., the quantity $c(i)$, or

3. the contents of the word of memory addressed by one level of indirection using the specified integer i, i.e., the quantity $c(c(i))$—the need



Operation Code	Address
1. LOAD	operand
2. STORE	operand
3. ADD	operand
4. SUB	operand
5. MULT	operand
6. DIV	operand
7. READ	operand
8. WRITE	operand
9. JMP	label
10. JGTZ	label
11. JZERO	label
12. STOP	

Figure 1.1: RAM

for this third option within a RAM is explained near the end of this subsection.

These three options are determined by the way that the operand is specified within the address field, namely, $=i$, i, or $*i$, respectively. For example, the program instruction that performs multiplication is defined as follows:

- 'MULT $=i$' means '$c(0) \leftarrow c(0) \times i$';
- 'MULT i' means '$c(0) \leftarrow c(0) \times c(i)$';
- 'MULT $*i$' means '$c(0) \leftarrow c(0) \times c(c(i))$.'

Other arithmetic operations are defined in a similar way.

Consider again the instruction that loads information from memory into the accumulator:

- 'LOAD $=i$' means '$c(0) \leftarrow i$';
- 'LOAD i' means '$c(0) \leftarrow c(i)$';
- 'LOAD $*i$' means '$c(0) \leftarrow c(c(i))$.'

The 'STORE' operation is defined analogously, but note that 'STORE '$=i$' is meaningless.

The input tape is a sequence of cells, each of which can hold an arbitrary bit pattern, or integer. It has a read head that identifies the cell to be read into the memory or accumulator by a 'READ' instruction. Thus, for example, 'READ i' means that the bit pattern, or integer, under the read head is transferred to the cell with address i, which must be nonnegative. (If $i = 0$ then it is transferred to the accumulator.) After its execution the read head always moves one cell to the right. Analogous comments hold for the output tape and its corresponding 'WRITE' instruction.

Finally 'JMP b' indicates an unconditional jump to the instruction labeled b. Substituting 'JGTZ' and 'JZERO' for 'JMP' changes the instruction to a conditional jump governed by $c(0) > 0$ and $c(0) = 0$, respectively.

The execution of a program is governed by the location counter, which points to the current instruction to be carried out by the RAM. If this is a jump instruction then the location counter may be set to point to the instruction determined by the corresponding label. For example, 'JGTZ b' will set the location counter to the program instruction labeled b when the integer in the accumulator is positive. Otherwise it is incremented by 1.

For all non-jump instructions, the location counter is incremented by 1, i.e., the next instruction in the program sequence is pointed to for execution by the machine. Execution ceases when a 'STOP' instruction is encountered or when an instruction is invalid, for example, 'MULT $*i$' with $c(c(i)) < 0$.

We highlight the following characteristics of a RAM:

- Its instruction set corresponds to a small subset of the instructions[5] in MMIX. This RAM instruction set can be augmented with other MMIX-type instructions as desired without any significant alteration of the RAM's computational capability,[6] in principle.

- All computation within a RAM takes place within the single accumulator. Additional accumulator registers could be added to the model to enhance efficiency, but again they would not alter the basic computational power of the model.

- In contrast to MMIX, memory cells and the accumulator of a RAM have unbounded capacity. Furthermore, no limit is placed on the number of memory cells in a RAM.

- A RAM does not store its program in memory or modify it. As such, it models an *algorithm* rather than a general-purpose computer, a distinction that has already been noted within the earlier MMIX model.

- The option of indirect addressing, i.e., the operand $*i$ within an instruction, is necessary in order to allow addresses to vary dynamically within a computation. Without this option, all addresses within a program would be predetermined integers and they could not be chosen at the time of execution, for example, by reading them from the input tape.

- A RAM can easily be extended to (an equivalent) *rational arithmetic machine*, henceforth denoted by RaM, where the data consists of rational numbers, or ratios of integers, represented implicitly by holding the numerator and denominator in memory. For instance, when the arc lengths on the network of section 1.1 are restricted to integers or rationals, Dijkstra's algorithm can be implemented as a RaM program.

[5]Note that the 'DIV' operation uses the celing function to ensure that the result is an integer.

[6]By capability, or basic computational power, we mean the ability to compute the class of (so-called) partial recursive functions.

1.3.2 Random Access Stored Program Machines (RASPs)

A random access stored program, or RASP, machine is identical in memory design and instruction set to a RAM and, in addition, employs an encoding scheme to translate each instruction into a bit sequence that can be be stored in memory. Each instruction uses two memory registers: the first holds the operation code of the instruction encoded as an integer, and the second holds the operand. The encoding of each operation code has two options, indicating whether the associated operand is to be used directly or as an address, i.e, $=i$ or i. The third option of indirect addressing within a RAM, namely $*i$, is no longer needed, because an instruction in memory can now be modified at run, or execution, time. In other words, a RASP can *simulate* the RAM operation of indirect addressing. Details of the encoding can be found in Aho, et al. [1974, p. 15-17].

Analogously to a RAM, the location counter points to the first of the two registers holding the current instruction. After it is executed, the location counter is incremented by 2 for all non-jump instructions. The operand in a jump instruction specifies the register to which the location counter may be reset, for example, JMP i or JGTZ i. (Note that the choice of operand $=i$ is meaningless.) The RASP machine executes a program stored in memory under the control of a "hardwired program," or central processing unit (CPU), analogously to the situation discussed earlier for the MMIX machine. Additional detail can be found in Aho et al. [1974]. Again, a key point to note is that *a RASP machine is a model of a general-purpose stored program, or universal, computer*, but this distinction and the universal-machine aspect of the RASP formulation is not emphasized in the foregoing reference.

1.4 Turing-Post and Turing Models

The computational models of Alan Turing and Emil Post were formulated in the 1930s, well before the advent of the electronic digital computer. Although they preceded the MMIX and RAM models in the historical development of computing, it is convenient to view Turing-Post programs and Turing machines through the lens of these more recent models, i.e., as *idealized* machines that are obtainable by *further abstraction* of a random-access model. Turing-Post and Turing models have the simplest structure of all, yet they do not sacrifice basic computational power, i.e., they retain the ability to compute, albeit less efficiently, the same class of functions as the RAM and RASP models, the so-called *partial recursive functions*.

Instruction	Meaning
WRITE σ	Write the symbol σ in the current cell (under the head).
DOWN	Move head down by one cell
UP	Move head up by one cell
IF σ GOTO L	Goto instruction labeled L if the current cell is σ
STOP	Terminate execution

Table 1.1: Turing-Post Instruction Set with σ = '0', '1', 'b', or '*'

In the approaches of both Turing and Post, the random-access memory of a RAM or RASP, along with its input and output tapes, which are depicted in Figure 1.1, are condensed into a *single* tape that is infinite in one direction, say downward, and divided into individual cells. Instead of an arbitary integer, or bit pattern, a cell on this tape can store only a single binary digit, i.e., the symbols '0' or '1', or it is left unwritten, or blank, denoted by 'b'. In addition, the uppermost cell is identified with the symbol '$*$'. (These four symbols comprise the alphabet of the computational model.) One particular cell on this tape is scanned by a *tape head*, which can *read* or *write* to this cell and then move to the *adjacent* cell immediately above or below, or remain stationary. In contrast to a RAM, the luxury of a randomly-addressable memory of cells, each of which has unbounded capacity, is no longer available. Furthermore, as we shall soon see, the instruction set of a RAM is further simplified within a Turing-type model.

1.4.1 Turing-Post Programs (TPPs)

The Turing-Post program model, which is depicted in Figure 1.2, is premised on the original formulation of Post. It is closer, in *spirit*, to the modern digital computer, but derives its impetus from the fundamental *analysis* of Turing; see Davis and Weyuker [1983]. The RAM location counter and program of Figure 1.1 are retained, but this program now consists of instructions of an even simpler variety shown in Table 1.1.

In this table, $\sigma \in \{0, 1, b, *\}$, i,e,. there are four choices for the 'WRITE' and 'IF' instructions, making for a total of eleven instructions within the table. A *Turing-Post* program is an (optionally) labeled sequence of instructions from this table. As in a RAM, they are executed in a linear sequence, unless a branch is encountered, with the current instruction being identified

by the location counter. *An algorithm, in this approach to computability, can again be viewed as just such a program that terminates on all inputs.* Davis and Weyuker [1983, Chapter 5] provide an excellent, detailed description of this approach, and a very accessible account can be found in Stewart [1987, Chapter 19].

Analogously to the transition from a RAM to a RASP machine, a universal TPP machine is obtained by using an encoding scheme for instructions and program labels that enables a Turing-Post program itself to be stored on the tape of the machine. Four bits are needed in order to encode an operation code (not all 2^4 patterns are needed) and the label within an IF instruction, or position in the program, can be encoded in a *unary* representation. For an example of an encoding scheme along these lines, see Stewart [1987, p. 215]. A *fixed* Turing-Post program can then be devised that is capable of emulating the action of any other TPP, when the former is furnished on tape with the encoded version of the latter, along with its data. This fixed Turing-Post program defines the associated *universal TPP machine*. Again, we can view it as the equivalent of a "hardwired CPU" of a general-purpose computer.

1.4.2 Turing Machines (TMs)

In a Turing machine (TM) as depicted in Figure 1.3, the location counter of a TPP, which points to the current instruction to be executed, is replaced by a "composite pointer" called the *finite-state control*. This can assume one of a finite number of states, which can be viewed as positions in a "primitive program" that consists of *quintuples*, each of which is of the following form:

$$(q, \sigma, \sigma', q', s)$$

where

- q is a state from the set of states that the finite-state control can assume.
- σ is taken from the alphabet as in the Turing-Post case, i.e., $\sigma \in \{0, 1, b, *\}$.
- σ' is also a member of the alphabet $\{0, 1, b, *\}$.
- q' is a state from the set of states.
- s is a member of the set $\{-1, +1, 0\}$.

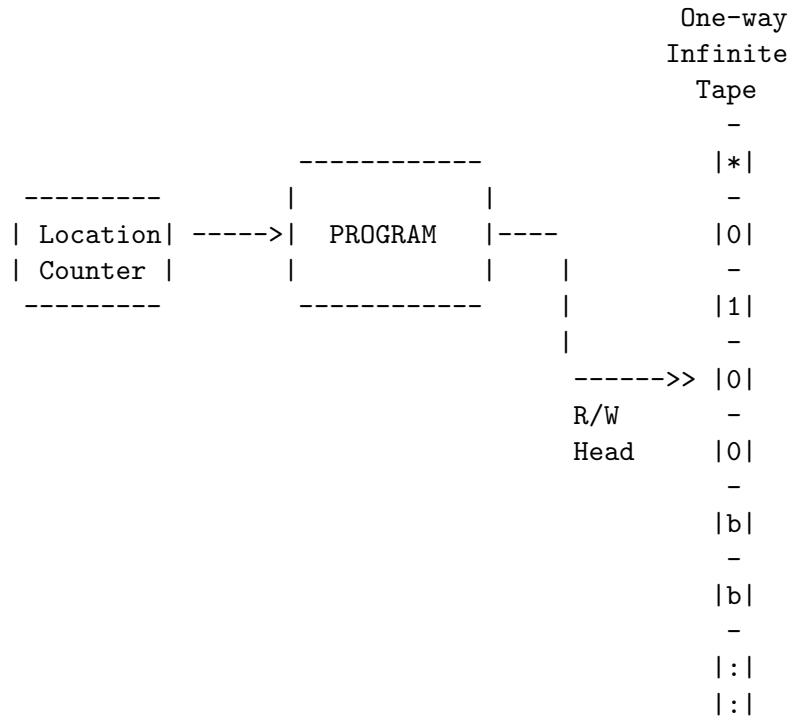


Figure 1.2: Turing-Post Machine

The set of quintuples can be specified *in any order* and each is a *primitive-program line* that is executed by the Turing machine as follows: when the finite state control is in state q and the symbol under the R/W head is σ then overwrite the content of the current cell (under the R/W head) with the symbol σ', change the TM's control state to q', and move the R/W head one cell up if $s = -1$, down if $s = +1$, and leave it in the same position if $s = 0$. The exception to this rule is when the tapehead is already at the uppermost cell and $s = -1$, in which case the tapehead stays in the same place. The machine is initially provided with a tape with the uppermost cell containing the symbol $*$, a finite string of 0s and 1s, and the rest of the tape consists of blanks. The R/W head is over the uppermost symbol, and the state of the machine is set to a (special) starting state, say, q_s. The machine looks through the program lines until it finds the quintuple $(q, \sigma, ., ., .)$ for which q corresponds to the current state of the machine and σ correspond to the current symbol under the head. It then executes the line as described above. If it enters a (special) halting state, say, q_h, or if no quintuple matches the current condition of the machine, then it stops. For a more detailed description of the TM model, see Aho et al. [1974], Garey and Johnson [1979], or Nielsen and Chuang [2000].

There are many variants on the foregoing TM formulation: a tape that is *infinite in both directions*; a TM with several tapes; a tape head that must move up or down at each executed step, i.e., $s \in \{-1, +1\}$; a smaller or larger vocabulary of symbols; primitive-program lines defined by quadruples rather than quintuples; and so on. These variants do not alter the inherent power of the resulting machines, which compute the class of partial recursive functions in all cases, but with increased or diminished efficiency.

Finally, we come to the key notion of a Universal Turing Machine (UTM), an idealized model of computation proposed by Alan Turing. It conceptualizes the modern general-purpose electronic digital computer, but predated the invention of the latter by Presper Eckert and John Mauchly by a decade. The idea is much the same as we have seen for earlier models, namely, encode the quintuples of a TM defined above so that they can be represented digitally on the tape, or machine memory. Then define a *fixed* TM that can simulate any other when furnished with the TM's encoding and data. This is the desired universal Turing machine. For a detailed and very readable formulation of such a UTM, see Penrose [1989]. In his formulation, Penrose uses a strictly digital vocabulary, i.e. $\{0, 1\}$, so that the encoded TM corresponds to a binary pattern—a representation of a number—as does its data. Indeed, because the associated UTM can itself be encoded, it too has

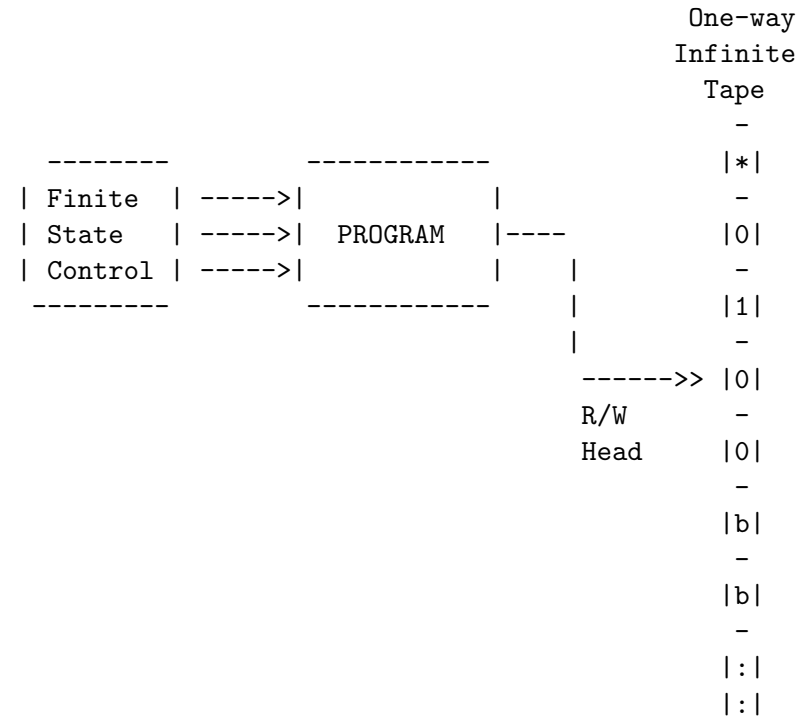


Figure 1.3: Turing Machine

a number, say u, and as Penrose [1989; p. 57] observes:

> This number no doubt seems alarmingly large! Indeed it *is* alarmingly large, but I have not been able to see how it could have been made significantly smaller. The coding procedures and specifications for Turing machines are quite reasonable and simple, yet one is inevitably led to a number of this kind of size for the coding of an actual universal Turing machine.

Later, he notes the following regarding the value of u (Penrose [1989, p. 73]) (italics ours):

> Some lowering of the value of u might have been possible with a different specification for a Turing machine. For example, we could dispense with STOP and, instead, adopt the rule that the machine stops whenever the internal state 0 is re-entered after it has been in some other internal state. This would not gain a great deal (if anything at all). A bigger gain would have resulted had we allowed tapes with marks other than 0 or 1. Very concise-looking universal Turing machines have indeed been described in the literature, but *the conciseness is deceptive, for they depend on exceedingly complicated codings for the descriptions of the Turing machines* generally.

In Chapter 3, we will return to this issue concerning concise universal Turing (and other) machines.

Among the different approaches, Turing's proved to be preeminent as a *foundation for computer science*. Turing machines compute the class of functions termed *partial recursive* (see, for example, Davis [1958, Chapter 3]), as do almost all the other models of this chapter, The exception is MMIX, which is the least powerful of our models, in principle, because its memory is finite, but the most usable in a more practical sense. MMIX can be idealized to allow memory cells and registers to hold binary integers of arbitrary size and make available an unbounded number of memory cells. This idealized version would again be equivalent in computational capability to the random-access and Turing models.

It is worth mentioning explicitly that a unit, or uniform, cost RAM model, or so-called UMRAM, although equivalent to a TM model in principle, is *not* polynomially equivalent to a Turing machine. Equivalence in terms of both computational power and speed (polynomial equivalence) requires the use of the *logarithmic* cost version of a RAM. Here our focus has

been the basic computation capability of the models and we do not address issues of the foregoing nature concerning computational complexity and algorithm efficiency.

1.5 Notes

Section 1.4: For a detailed discussion of computational complexity, see, for example, Aho, Hopcroft, and Ullman [1974], Garey and Johnson [1979], or Mertens and Moore [2010].

Chapter 2

Magnitude-Based Computing

This chapter is from the e-book by J.L. Nazareth (2010), *Algorithmic Science and Engineering within Scientific Computing: Foundations and Organization*, 128 pgs., www.math.wsu.edu/faculty/nazareth/asebook.pdf .

Within the symbol-based computational models of the previous chapter, numbers are restricted to integers or rationals and represented by finite, binary strings; see, in particular, the rational arithmetic computational model (RaM) mentioned at the end of section 1.3.1. Such models cannot represent and operate on real numbers, the numerical mainstay of the mathematical and engineering sciences.

We now turn to machine models that are capable of performing computations over the real field,[1] at least *in principle*. Their representation of real numbers is in the spirit of the following remarks of Stainslas Dehaene [1997, p. 237] on the number sense of humans and other animal species as viewed from the perspective of cognitive science (italics and the footnote are ours):

> The peculiar way in which we compare numbers thus reveals the original principles used by the brain to represent parameters in the environment such as a number. Unlike the computer it does not rely on a digital code, but on a continuous quantitative internal representation. *The brain is not a logical machine, but an analog device.* Randy Gallistel[2] has expressed this conclusion with remarkable simplicity: "In effect, the nervous system in-

[1] Hence also the complex numbers.

[2] Professor of Psychology and Cognitive Science, Rutgers University, New Jersey.

> verts the representational convention whereby numbers are used to represent linear magnitudes. Instead of using number to represent magnitude, the rat [like the Homo sapiens!] uses *magnitude to represent number*."

Our notion of *magnitude-based computing*, as contrasted with symbol-based computing of the previous section, is related in *concept* to the foregoing, although, obviously, the design and construction of our computational models is along completely different lines from the mammalian brain.

We introduce magnitude-based computing by constructing a class of idealized, *special-purpose* devices called *string-node, or SN, machines* that represent real numbers by *analog* quantities. They are motivated by an observation of Strang [1986], and they exhibit an interesting relationship to Dijkstra's algorithm of Chapter 1, section 1.1.

Next, we construct an idealized, *general-purpose* model that is based on an extension of the random-access computational model of Chapter 1, section 1.3.1. The resulting *continuous-discrete random-access machines (CD-RAMs) and stored programs (CD-RASPs)* can model computations performed over the field of real numbers, and, more generally, the complex numbers viewed as pairs of reals. They formalize the approach to computation spearheaded by Traub [1999]—see also Traub and Werschulz [1998]—as a foundation for scientific computing.

In the concluding section of this chapter, we describe the BCSS model of Blum, Cucker, Shub, and Smale [1998] and place it within the framework of models discussed previously.

2.1 String-Node Machines

Our motivation for the idealized machines introduced here comes from Gilbert Strang [1986, p. 617], who concludes a discussion of algorithms for finding shortest paths on an *undirected network with non-negative arc lengths* with the following brief, but pithy, observation:

> We cannot resist one more algorithm. The shortest paths can be found with pieces of string. Cut them to the lengths given by the network and attach them to the nodes. Then pick up node i in one hand and node j in the other, and pull until some path between them becomes taut. This is the optimal path.

In this quotation, the word "algorithm" is used in the sense of a systematic procedure for constructing a particular physical device that can then be manipulated manually, in a prescribed manner, in order to find a shortest path in a given network. But now let us examine this constructed device itself and explore the algorithms—in the sense of Chapter 1—that an *idealized version* of the device is capable of emulating when manipulated in a different way.

Let us make the following assumptions:

- a string can be cut to an exact length defined by a positive real number;
- strings are joined in nodes that are infinitesmal, or dimensionless, points;
- strings are unidimensional, unbreakable, and non-entangling, so that the constructed device retains its integrity when manipulated physically.

Suppose the idealized device, which is henceforth called a *string-node, or SN, machine*, is constructed for a network (N, A) of section 1.1 corresponding, say, to the highway network described there. Place this SN machine on a flat surface $\mathcal{S}$ and slowly raise the given source node s above the surface. At some stage a node v that is nearest to s will be lifted off the surface and the string joining s to it will become taut. (Without loss of generality, we can assume v is unique.) Let the nodes s and v define X, the set of nodes that are currently "labeled." As s continues to be lifted upwards, the strings attached to nodes in X will now be lifted off the surface and eventually one of them will become taut, identifying the second closest node to s. Add this node to X and continue the procedure. Eventually the terminal node t will begin to be lifted off the surface and the shortest path from s to t will have been found. Similarly, the machine can be used to solve the single-source shortest-path problem by continuing to lift the source s until all nodes have left the surface $\mathcal{S}$. The set of taut strings will form a *tree* that identifies the shortest paths from s to all nodes of the given network.

It is evident that the SN machine, used in the foregoing manner, is carrying out a version of Dijkstra's shortest-path algorithm of Chapter 1, section 1.1, one that employs a considerable amount of "parallel processing." Observe that comparisons between strings of different lengths take place simultaneously as they lift off the surface $\mathcal{S}$ and that other strings and nodes

remain inactive on the surface. The "complexity" of the SN machine is governed simply by the length of the shortest path from s to the terminal node t (or the set of terminal nodes). The entire computation could be done in parallel, if desired, by holding the source node s fixed and letting the surface $\mathcal{S}$ fall away. In this case, the other nodes of the SN machine will fall downwards, in parallel, and assumes a final configuration defined by the shortest path tree.

Research Project: An interesting open question is whether the idealized string from which an SN machine is constructed can be generalized, so that the resulting machine is able to compute shortest paths for other network problems. It is not difficult to see how this can be done for a *directed* network, again with nonnegative arc lengths. Formulate this extension.

Explore the more challenging extension to a directed network with *unrestricted* arc lengths. Formulate a generalized SN machine that is capable of emulating Ford's extension of Dijkstra's algorithm (see the conclusion of section 1.1).

SN machines and their extensions are a useful *metaphor* for the "computation" that one frequently encounters in nature. For example, the folding of a protein molecule, from its denatured state into its final native configuration, can be viewed as a highly parallel "physical computation" of this type. Identifying an underlying algorithm[3] and expressing it in the conventional terms of Chapter 1, so that it can be executed on an electronic digital computer, is currently viewed as one of the important challenges of computational molecular biology; see Lesk [2002, p. 237] for further discussion.

2.2 Continuous-Discrete Random Access Models

Traub [1999] makes the following important observation on the foundations of computing (see also Traub and Werschulz [1998, Chapter 8]):

> A central dogma of computer science is that the Turing-machine model is the appropriate abstraction of the digital computer. $\cdots$ I argue here that physicists should consider the real-number model of computation as more appropriate and useful for scientific computation. $\cdots$ The crux of this model is that one can *store and perform arithmetic operations and comparisons on real numbers*

[3] If such an algorithm does indeed exist!

exactly and at unit cost.

We will now consider the design of an *idealized, general-purpose* machine that formalizes this real-number model, and additionally permits logarithmic-cost assumptions. It uses magnitude-based techniques derived from geometry to perform elementary, real-number arithmetic and comparison operations—originally proposed in Nazareth [2003, Chapter 15]—and incorporates them into the random-access computational models of section 1.3.

2.2.1 CD-RAM Registers

Figure 1.1 in Chapter 1, section 1.3, depicts a random access machine (RAM) and its idealized registers, or memory cells. We now extend a RAM register so it can assume either of two states, which we call 'passive' and 'active,' respectively.

In the passive state, the register is identical to that of a RAM, i.e., it can store a positive or negative integer as a *binary* sequence, and the number zero.

In the active state, it can store a positive or negative real number in the form $\pm a2^{\pm e}$, where $a \in [1, 2)$ and $\pm e$ is an integer. The signed quantities $\pm a$ and $\pm e$ are called the *mantissa* and the *exponent*, respectively. The positive, real number a is represented as a *magnitude*, or length, in the classical Greek sense—see, for example, the Eudoxian theory of proportion within Euclid's elements in the anthology of Hawking [2005, p. 25-62]—and a *unit* length is assumed to be defined within the machine. A positive or negative sign is associated with a. The signed integer e is represented in the *unary* number system (sequence of 1's).[4] This implies that the number of bits needed to store any given integer, say, i, in the passive state, is essentially the *same* as the number of bits needed to store its exponent, when i in converted to an active, real representation.[5] Note also that "zero" is representable in the passive state, but not the active.

A register in an active state uses a representation of the mantissa analogous to that used in standard finite-precision, floating-point arithmetic. But, in the latter case, the mantissa is stored, instead, in *symbol-based* form as a

[4]This string of 1's can be viewed as the "digital length" of the exponent, the counterpart of the "analog length" of the mantissa.

[5]If desired, binary representation could be used instead for the exponent and the number of bits needed would then be $\log_2 i$.

normalized, finite binary sequence, and the exponent as a binary sequence. Again, zero cannot be represented as a normalized floating-point number.

2.2.2 Geometric Operations involving Magnitudes

Consider two magnitudes, say $a \in [1, 2)$ and $b \in [1, 2)$ as defined above. Arithmetic and comparison operations between them will be performed in the magnitude-based, or geometric, sense as follows.

Geometric Addition and Subtraction

The two magnitudes are aligned and added or subtracted in the classical geometric manner. If the magnitudes are equal and the operation is subtraction, the result 'zero' is represented in the passive form; see also section 2.2.4 below.

Geometric Multiplication and Division

The operation of multiplication is performed geometrically as depicted in Figure 2.1. The vertical and horizontal lines in the figure are parallels, and similarity of triangles implies that

$$\frac{a}{1} = \frac{AG}{BG} = \frac{OG}{DG} = \frac{FG}{EG} = \frac{c}{b}.$$

Hence, $c = ab$.

The operation of division is defined in an analogous way. For example, division by 2, or *geometric halving*, is performed as shown in Figure 2.2.

Geometric Comparison

Comparison of the two magnitudes a and b to determine which is the greater, or to determine equality, is again accomplished in the classical geometric sense.

2.2.3 Active-State Operations

Give the geometric operations on magnitudes as defined above, we can now define real-number operations for our CD-RAM. Consider two real numbers with mantissas $\pm a$ and $\pm b$, respectively, and corresponding exponents $\pm e$ and $\pm f$. For convenience of discussion and without loss of generality, let us choose positive signs for all four quantities and also assume that $e > f$.

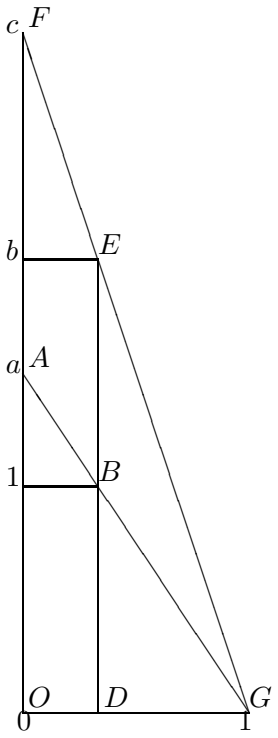


Figure 2.1: Geometric Multiplication

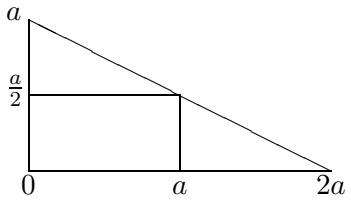


Figure 2.2: Geometric Halving

Active-State Addition and Subtraction

The operation of addition is performed as follows:

1. Form $\bar{b} \leftarrow \frac{b}{2^{(e-f)}}$ by *repeated* geometric halving.
2. Form c by geometric addition of a and $\bar{b}$, and set $g \leftarrow e$.
3. Using geometric comparison, if $c \geq 2$ then decrease $c \leftarrow \frac{c}{2}$ by geometric halving and increase $g \leftarrow g + 1$ by unary addition.
4. The final result is $c2^g$.

The operation of subtraction and other variants, when the assumptions on the real numbers above are removed, are defined in an analogous manner.

Exercise: Develop the procedure for the subtraction operation, considering specifically the case when $e = f$ and the mantissas a and b are almost equal (this will require geometric doubling that is analogous to Figure 2.2).

Active-State Multiplication and Division

The operation of multiplication between the foregoing two real numbers ($a2^e$ and $b2^f$ under the stated assumptions) is defined as follows:

1. Form c by geometric multiplication of a and b.
2. Form g as the unary sum of e and f.
3. Using geometric comparison, if $c \geq 2$ then decrease $c \leftarrow \frac{c}{2}$ by geometric halving and increase $g \leftarrow g + 1$ by unary addition.
4. The final result is $c2^g$.

Extensions when mantissas and exponents have arbitrary signs and the definition of active-state division are analogous.

Active-State Comparison

If the two real numbers have opposite signs then the result is immediate. Likewise, if the mantissas have the same signs and the exponents differ. The remaining case is achieved by geometric comparison of magnitudes (corresponding to the mantissas of the two numbers).

Conversions between Active and Passive States

These operations convert an *integer* between its active and passive representations.

a) Active → Passive: Consider a number that is assumed to be an *integer* and currently represented in the active form $a2^e$. Without loss of generality, we take the number to be positive. Obviously $e \geq 0$. The following procedure for conversion to the passive representation of the specified integer includes a safeguard for incorrect input:

1. Take the leading digit of the passive representation to be 1.

2. Reduce $a \leftarrow a - 1$ by geometric subtraction. If $a = 0$ then set each of the last e digits of the passive representation to 0 and stop—the conversion has been successful.

2.1 Increase $a \leftarrow 2a$ by geometric addition. Using geometric comparison, if $a \geq 1$ then goto step 3.

2.2 Set the next passive digit to 0 and reduce $e \leftarrow e - 1$ by unary subtraction. If $e = 0$ then terminate with an error message, because the number is not an integer and the conversion has failed. Otherwise, return to step 2.1.

3. Set the next passive digit to 1 and reduce $e \leftarrow e - 1$ by unary subtraction. If $e = -1$ then terminate the conversion with an error message, because the number is not an integer. Otherwise, return to step 2.

Note that instead of returning an error message, the foregoing procedure could instead terminate with the floor (or ceiling) function value returned in the passive representation.

b) Passive → Active: This operation is straightforward using geometric halving and geometric addition, in combination. Details are omitted. Note that unlike the previous conversion, this procedure is always successful, i.e., it cannot terminate with an error return.

2.2.4 Passive-State and Mixed Operations

Operations between integers represented in the passive state proceed as in Chapter 1, section 1.3. A mixed operation, for example, multiplication of a number represented actively and a number represented passively, entails a conversion of the latter, necessarily an integer, to its active representation, followed by multiplication of the two active-state numbers (as described in section 2.2.3). If one of these operands is zero, which can only be represented passively, the multiplication operation would be handled as a special

case. Other mixed operations, involving addition, subtraction, and division, between an active and a passive operand, are performed analogously.

2.2.5 CD-RAM and CD-RASP Models of Computation

Given the foregoing repertoire of basic operations, one can then formally develop a machine model for real-number computation, termed a *continuous-discrete random access machine*, or CD-RAM. It could be used, for example, in conjunction with Dijkstra's algorithm, to find shortest paths on networks whose arc lengths are no longer restricted to integers or rational numbers.

The CD-RAM would be analogous to the machine depicted in Figure 1.1 of Chapter 1, but its memory registers and accumulator would now be of the type described in section 2.2.1. Likewise, its input and output tapes also have cells that can store either active or passive representations of numbers. The instruction set of a standard RAM and the interpretation of each instruction's operand, as summarized in Figure 1.1 and given, in detail, in Aho et al. [1974], can be extended appropriately.

As described in section 1.3.1, the operand in a RAM instruction has three options, which were denoted by $=i$, i, and $*i$. In a CD-RAM, an operand will again have these three options, but now the quantity i can be either an *i*nteger (represented *p*assively or *a*ctively) or a *r*eal number (always represented actively). For emphasis, we will henceforth replace i by *ipar*—derived from the foregoing four italicized letters—and denote these three options as $=ipar$, $ipar$, and $*ipar$, respectively.

For the first case, $=ipar$, the operand is used directly within an instruction and specifies a passive or active quantity. For the other two options, the operand identifies an address. If *ipar* is represented actively then it is converted to its passive representation (see section 2.2.3). The machine stops if the conversion procedure indicates that *ipar* is not an integer (error return). Similarly, for the $*ipar$ option, the contents of the address *ipar*, namely, $c(ipar)$, can be passively or actively represented. In the latter case, $c(ipar)$ is converted to the passive representation to identify the quantity $c(c(ipar))$, which is then employed by the associated machine instruction. Again the machine stops if $c(ipar)$ is not an integer.

As in a RAM, the instructions of a CD-RAM fall into five main groups as follows:

1. *LOAD, STORE*: If the operand is specified directly by a LOAD instruction in the form $=ipar$ then it is loaded into the accumulator as

the corresponding passive or active quantity. As in a RAM, the instruction 'STORE =*ipar*' is meaningless. The other two options of the operand define a machine address as discussed above. The LOAD instruction transfers the (passively or actively represented) contents of this address to the accumulator. Similarly, the STORE instruction transfers the contents of the accumulator to the address.

2. *ADD, SUB, MULT, DIV, CONV*: As in a RAM, these instructions perform the corresponding arithmetic operation between the contents of the accumulator and the contents of a machine address (or a directly specified quantity). If both quantities involved in an ADD, SUB, MULT, or DIV arithmetic operation are passively represented then the CD-RAM behaves identically to a RAM and the result of the operation in the accumulator is passively represented—recall for the DIV operation the ceiling function value is returned. Otherwise, arithmetic operations are carried out as described in section 2.2.3 (operations between two active quantities) and section 2.2.4 (mixed operations).

   In addition, an explicit conversion instruction is provided in a CD-RAM. Using procedures given near the end of section 2.2.3, it converts the quantity in the accumulator to its complementary representation, i.e., active to passive, or passive to active. In the former case, if the (active) quantity specified is *not* an integer then CONV can be implemented to return the passively represented ceiling (or floor) integer in the accumulator.

3. *READ, WRITE*: These are implemented in a CD-RAM in an analogous manner to the LOAD and STORE instructions above, and details are omitted.

4. *JMP, JGTZ, JZERO, JACT*: As in Figure 1.1, each of these instructions has an associated label that identifies the instruction in the program to which a jump is executed, depending on the quantity in the accumulator. The extension of each RAM jump instruction to the corresponding CD-RAM jump instruction is straightforward. An additional instruction, JACT, executes a jump to the instruction identified by its label, if the accumulator holds a number that is represented actively. It can be used, in conjunction with CONV, to convert a number representation from active to passive, or viceversa.

5. *STOP*: This is identical to the RAM instruction.

We now briefly consider properties of the foregoing CD-RAM model, and we conclude this discussion by contrasting the CD-RAM model with other models proposed for scientific computing.

Computing Power

There are no rounding errors when executing an arithmetic operation in a CD-RAM. It can therefore produce the exact solution of a problem that is solvable in a finite number of arithmetic steps, for example, a linear system of equations $Ax = b$, where A is a nonsingular, square matrix and b is a specified right-hand side.

On the other hand, the computation of a function as simple as $\mathrm{e}^x, x \in R$, is not exact over all values of its domain. For example, e^n, where n is an integer, can be computed exactly under the assumption that an active representation of e is specified as a *machine constant*, via the read tape or the $=ipar$ option of an operand. But $\mathrm{e}^{1/2}$ must then be found from an infinite series or by solving the equation $z^2 - \mathrm{e} = 0$ for z, and it can be computed only to within a truncation error.

More broadly, a CD-RAM program can contain a finite set of real machine constants that are specified in the above manner, for example, the numbers e, π, or $\sqrt{2}$. It can then compute other real numbers in the subfield generated by these machine constants, without truncation error. Numbers that have not been specified or derived in this manner may not be computable in a finite number of steps. For example, suppose that $\sqrt{2}$ is not provided as a machine constant. It must then be computed, typically by applying Newton's method to solve the equation $z^2 - 2 = 0$, and will only be available to an accuracy determined by a termination criterion.

The ability to specify real number as machine constants within a CD-RAM instruction is similar to the use of real numbers as machine constants in the rational maps within the nodes of a BCSS machine, as will be described in the next section. Since real numbers can be used within programs, it also follows that the set of CD-RAM programs is not countable.

Digital Bits versus Analog Bits

Consider a (positive) integer, say i, which can be represented passively as a binary string (of digital bits consisting of 1s and 0s) or actively in the form $a2^e$, where the exponent e is represented as a unary string (of digital bits consisting of 1s) and the mantissa a as a magnitude, as described in section

2.2.1. The length of the binary string, or number of digital bits, in the passive representation of the integer is essentially the same as the length of the unary string employed in its active representation, i.e., each requires the same number of digital bits. The magnitude corresponding to the mantissa of the active representation of the integer can be *interpreted as an analog bit* (henceforth abbreviated to A-bit) of the CD-RAM. See also the footnotes for section 2.2.1. In the case of a real number, the representation must be active, and again it takes the form of a finite string of unary bits defining the associated exponent and a *single* A-bit defining the associated mantissa.

Each basic operation of addition, subtraction, multiplication, and division between two given magnitudes defining associated mantissas—executed geometrically as described in section 2.2.2—can be interpreted as an operation between a corresponding pair of A-bits within a CD-RAM, *from which a new A-bit defining the result of the operation* is created by the machine.[6] We shall assume that each such basic operation between two A-bits has an associated cost t, which is *comparable* to, i.e., of the same order as, the cost of an operation between two digital bits, say τ.

Under the foregoing assumptions, a CD-RAM can be designed to utilize a logarithmic cost model of complexity for passively- or actively-represented numbers. This is explored in more detail in the next subsection.

Note also that the number of bits (digital and analog) for actively representing a real number is essentially the same as the number of bits for representing its inverse. Consequently, numbers that are very large in magnitude have comparable representational and arithmetic-operation costs to numbers that are very small in magnitude.

Complexity

Within a standard RAM, Aho, et al. [1974, pg. 13] define a "*logarithmic cost criterion* based on the crude assumption that the cost of performing an instruction is proportional to the length of the operands of the instruction." Consider, for example, the RAM instruction 'MULT i'. Let $l(i)$ be defined as the smallest integer that exceeds or equals $\ln(i)$, where 'ln' denotes the logarithmic function. Then the cost of the multiplication operation 'MULT i', between the contents $c(0)$ of the accumulator, namely, register 0, and the

[6]The digital bits, '0' and '1', used in the machine can be represented beforehand. In contrast, only a finite set of A-bits are furnished in the form of program or data and other analog bits are *constructed by the CD-RAM* during the course of its computation.

contents $c(i)$ of register i, is crudely estimated by

$$[l(i) + l(c(0)) + l(c(i))]\tau, \tag{2.1}$$

where the first term is the cost associated with decoding the address i and τ is defined in the previous subsection.

Consider now the CD-RAM multiplication operation 'MULT *ipar*', and assume *ipar* is passively represented. For emphasis, we will replace it by i and write the operation again as 'MULT i'. The cost of this operation in a CD-RAM is identical to that of a RAM when the numbers in the accumulator and register i of the CD-RAM are represented passively, and this cost is thus given by (2.1). Suppose the numbers, or operands, in the accumulator and register i that are to be multiplied together are represented actively (they may or may not be integers and the product operation is performed as described in section 2.2.3). Assume that their exponents use a unary representation and that the cost of unary operations is proportional to the number of bits in the operands. The cost of decoding the address i and of adding together the exponents in the multiplication operation is again given by (2.1). There is the additional cost of multiplying the mantissas of the two operands as described in section 2.2.2. We assume that the cost of this operation is independent of the magnitudes of the two mantissas and is given by a fixed cost, say, t (see the previous subsection). It may be necessary to renormalize the product which would increase the cost by a small multiple κ of t, say $1 \leq \kappa \leq 3$. Thus the logarithmic cost of the CD-RAM multiplication operation between operands that are actively represented is crudely estimated by the quantity

$$[l(i) + l(c(0)) + l(c(i))]\tau + \kappa t. \tag{2.2}$$

Consider now the case 'MULT *ipar*' where the address of the register is an actively-represented, positive number *ipar*. It must then be converted to its passive representation and verified to be a positive integer (valid address), as described earlier in this section. Let us make the crude assumption that the cost of each of the basic *geometric* operations between two mantissas—addition, subtraction, multiplication (including doubling), division (including halving)—is again given by t. Consequently, the cost of one cycle of the procedure given in section 2.2.3 for converting an active to a passive number is approximately $2t$, corresponding to the two geometric operations. We can then approximate the cost of decoding the address *ipar* by $2l(ipar)t$, where

$l(ipar)$ is the number of unary bits in the exponent of $ipar$. Thus the cost of the operation 'MULT $ipar$' can be crudely estimated by

$$[2l(ipar)]t + [l(c(0)) + l(c(ipar)]\tau + \kappa t.$$

The costs of performing other arithmetic operations between actively represented numbers in the accumulator and register $ipar$, based on the procedures given in section 2.2.3, can be estimated in an analogous way. Details are omitted.

Exercise: Develop estimates of the cost of arithmetic operations when the operands are actively represented and their exponents are in binary form (instead of unary).

Exercise: Develop more precise estimates of costs for converting a number from active to passive form, and viceversa, within the procedures of section 2.2.3.

Exercise: In conjunction with the foregoing estimates of cost, consider the implications of number representation—passive versus active—on the efficiency of arithmetic operations.

As mentioned above, the complexity estimates in this and the previous subsection are very crude. A more careful and precise investigation of the complexity of CD-RAM operations and the associated model would be a worthwhile undertaking.

Variants and Extensions

A CD-RAM can be extended to operate on *complex numbers*, just as a RAM, which operates on integers, can be extended to perform rational arithmetic. The former is achieved via complex-arithmetic operations on pairs of reals, the latter via rational-arithmetic operations on pairs of integers. Details are omitted.

The CD-RASP Model

Analogously to the RASP model of Chapter 1, section 1.3.2, a stored program version of a CD-RAM can also be formulated, yielding a CD-RASP model and a corresponding *universal machine*. As before, we can use two successive registers to store an instruction and it operand, where the latter can be a passive or active quantity when directly specified, or a passively

specified address. Again, indirect addressing is not needed.

Research Project: Formulate the CD-RASP model of computation in detail and explore the computational complexity of its arithmetic operations.

Contrasts with Other Models

As seen in Chapter 1, there are a variety of symbol-based models of computation. A fundamental result of theoretical computer science is that they are equivalent, i.e., they define the same notion of (partial-recursive) computability, the well-known Church-Turing thesis; for a very readable account, see Feynman [1996; p. 54].

In contrast, there is no single accepted model in the continuous, magnitude-based, setting. At one extreme is the "bit-model" in the Turing tradition. For a good introduction, see Braverman and Cook [2006] and its cited antecedents given there. At the other extreme is the "continuous-time" model of Moore [1996] and its cited antecedents, which are in the tradition of the general-purpose analog computer of Shannon [1941].

To date, the most widely known model for scientific computing is the aforementioned Blum-Cucker-Shub-Smale (BCSS) model, which will be described in more detail in the next section. The CD-RAM and CD-RASP models developed here are of interest in their own right and they also provide a bridge between the traditional symbol-based computational models of Chapter 1 and the BCSS model.

2.3 The BCSS Model of Computation

The BCSS model of Blum, Cucker, Shub, and Smale [1998] is the most comprehensive approach, to date, for performing idealized computation over the real and complex numbers (and other number fields).[7] Blum et al. [1998] employ a mathematically-oriented formulation and terminology, but their model can be viewed through the lens of the (more concrete) random-access (RAM) and Turing (TM) models of computation described previously. The main features of a BCSS machine can be stated in these two parallel terminologies as follows (the BCSS in italics and the RAM/TM in quotes):

[7]Indeed, the BCSS model is set up so that it applies to computation over any ring or field. In particular, when the integers modulus 2, or Z_2, are used for the underlying structure then the classical theory of computation considered in Chapter 1 is captured.

- a *state space* S_M, or 'memory';
- an *input space* I_M, or 'read-only input tape';
- an *output space* O_M, or 'write-only output tape';
- and a *directed graph* defined by a set of *nodes* and *directed edges*, or a 'program' with associated 'location counter/finite control.'

In a *finite-dimensional* BCSS machine, the memory consists of a *finite* number of registers, numbered 1, 2, 3, ..., m, which define their 'addresses.' Each register can store an arbitrary real number (including zero). The formulation in Blum et al. [1998] permits computation over other number fields as well, but here *we focus on the reals.* The memory can be identified with Euclidean space R^m and defines the state space S_M of the BCSS machine. (In contrast to the CD-RAM formulation, note that the manner in which a real number is represented within a BCSS register is not specified. It is viewed simply as an atomic quantity, or basic unit of computation.) Similarly the input and output 'tapes' consist of n and l contiguous cells, each of which can store a real number. The tapes can be identified with Euclidean spaces R^n and R^l that define the input space I_M and output space O_M, respectively, of the machine.

A program is defined by a finite set of 'primitive program lines' called *nodes*, each identified by a *unique* label. A primitive program line is given by a Turing-like 'quadruple' as follows:

$$(\beta, \text{instruction-type}, \text{instruction-field}, \beta'),$$

where β is the unique label given to the node, and β' is the label of a *single* successor node, *thereby defining the program's execution sequence.*

A *finite-dimensional* BCSS machine has four basic types of program lines, or nodes, and their 'instruction-types'—*computation, branch, input, output*—and associated 'instruction-fields' within the foregoing quadruples are as follows:

1. *COMPUTATION NODE*: In place of LOAD and STORE operations and the four elementary arithmetic operations within a RAM (see Figure 1.1), a BCSS machine has a single, generic, COMPUTATION operation defined by associating a rational map[8] with the node. This

[8] A rational map, say $g : R^m \to R^m$, has m components, $g_j, j = 1, \ldots, m$, with $g_j(x) = (p_j(x))/(q_j(x))$, and p_j and q_j are polynomials. Each such polynomial has m variables,

rational map, say g, which is specified in the 'instruction-field' of the program line, or node, maps the *entire memory, or state space, into itself*, i.e., $g : S_M \to S_M$. (Note that different nodes can have different rational maps, each of different degree.) A computational node transforms all m registers of the memory *simultaneously* and, for reasons that will become apparent below, it is useful to envision this operation being performed by a 'composite R/W head' that spans the m registers of the BCSS machine memory, akin to the R/W head of a Turing machine that transforms a single cell of its corresponding tape.

2. *BRANCH NODE*: Compute the result, say $r \in R$, of an associated rational functional, specified in the 'instruction-field', that maps the current memory state to the reals, i.e. $S_M \to R$. If $r \geq 0$ then go to a node identified by a label, say β'', also specified in the 'instruction-field.' Otherwise, the execution sequence is defined by the successor β'.

3. *INPUT NODE*: A *linear*, or matrix, map from I_M to S_M is used to initialize the memory. This map is specified in the instruction-field of the quadruple. An input node can only occur once at the start of the execution sequence, i.e., in any other program line, or node, β' cannot be set to the unique label attached to the input node.

4. *OUTPUT NODE*: A *linear*, or matrix, map from S_M to O_M. This map is again specified in the instruction-field. Then the BCSS machine halts, i.e., β' is null for each output node.

In the *normal form* of a BCSS machine, the unique labels identifying nodes, or program lines, are the numbers 1, 2, ..., N. The input node has $\beta = 1$. There is also only one output node and it is given the unique label N. The rational functional, $S_M \to R$, within a branch node is instead assumed to have been defined earlier, within a computation node, and the result, r, placed in the first register of memory. The branch node then makes its decision based on the number in this first register, i.e., it uses the first coordinate of the state space S_M.

A BCSS program in normal form can be viewed as a finite list of primitive program lines with an associated location counter. Their (integer) node

corresponding to the m registers, and is of finite degree $\leq d$. Thus m is the dimension of the rational map and d its finite degree. The polynomials, in turn, can be defined as the sum of monomials; for further detail, see Blum et al. [1998, p. 41].

labels define the execution, or control, sequence. (The node labels are the analogue of the finite states of a Turing machine and they should not be confused with the state space, or memory, of the BCSS machine, which corresponds to the memory tape in the TM context, or the registers of a RAM.) A 'location counter' identifies the current line, or node, within the program that is to be executed. Equivalently, as in Blum et al. [1998], a BCSS program can be described by a *finite, directed graph*, or flow chart, defined by the set of nodes, with the directed edges of the graph being deduced from the labels β and β' attached to each node (and, additionally, β'' for a branch node). The execution sequence is defined by following a *directed path* within the directed graph.

In the *general*, or uniform, version of a *BCSS machine*, the memory is potentially *infinite*, i.e., there is no upper limit on the number of registers (as is also the case with a standard RAM); likewise for the input and output tapes.[9] The state space S_M, the input space I_M, and the output space O_M, can each be identified with the space of infinite sequences of real numbers R^∞.

The general BCSS machine *program* still has finite dimension, say K_M, taken as the *maximum* of the dimension of all rational maps associated with its nodes. Likewise, the degree D_M is the maximum of the degrees of such rational maps. In normal form, all maps are assumed to have the *same* dimension K_M, and the 'composite R/W head' described earlier for an m-dimensional state space now spans these K_M registers. In order to allow the general BCSS machine to access registers with *arbitrarily high* addresses, a new node type is introduced, called a *SHIFT NODE*, which performs a function akin to the movement of the Turing R/W head. It can move the composite R/W head up or down by one register, except when the first cell of memory is already under its span and the head is commanded to shift up; in this case, the head stays put. Alternatively, instead of introducing a shift node explicitly, the primitive program lines defined above by Turing-like quadruples can be extended to 'quintuples' of the following form:

(β, instruction-type, instruction-field, β', s)

[9] The formulation in Blum et al. [1998] uses a two-way infinite memory, or state space, R_∞, akin to a two-way TM tape, and one-way infinite input and output spaces, identified with R^∞. However, we find it more convenient here to use memory, input, and output spaces or tapes that are all one-way infinite.

where $s \in \{-1, +1, 0\}$, and $s = -1$ means move the composite R/W head up by one cell, $s = +1$ means move it down by one cell, $s = 0$ means leave it in place.

Finally, a *universal* BCSS machine can be defined by designing a simulator for the rational maps associated with nodes of a general BCSS machine; further details can be found in Blum et al. [1998].

Observe that the BCSS machine model of computation can be viewed as a high-level mathematical abstraction and synthesis of RAM and Turing concepts discussed in Chapter 1. Additional variants related more closely to the approach of Emil Post can be envisioned, for example, a BCSS machine with a linear form of control akin to a Turing-Post program; see section 1.4.1. This would eliminate the need for the successor label β' in a primitive program line, or node. Also, if desired, the spaces I_M and O_M can be eliminated by permitting the input and output to be specified and returned via the memory tape.

2.4 Notes

Section 2.1: Recalling that records in the Inca civilization of South America were kept with quipu—a bundle of woven, coloured threads—and that the Incas were famous for the network of roads across their empire, one can almost imagine a string-node machine being constructed by a pre-Columbian engineer to find shortest paths between Inca cities.

Section 2.2: Likewise, the idealized, *active* CD-RAM operations could conceivably be realized, albeit approximately, by laser-based (optical) arithmetic units.

Section 2.3: The BCSS model provides the foundation for a deep study of computational complexity over the reals; see Blum et al. [1998; in particular, Parts I and III].

Chapter 3

Complex Behavior of Simple Programs

This chapter is from the e-book by J.L. Nazareth (2010), *Algorithmic Science and Engineering within Scientific Computing: Foundations and Organization*, 128 pgs., www.math.wsu.edu/faculty/nazareth/asebook.pdf .

Seemingly trivial programs, which can be implemented as simple instances of the machines described earlier, can exhibit behaviour that is remarkably complex and highly unpredictable. This phenomenon was discovered, and has been studied extensively, by Wolfram [2002]. In this chapter, we briefly survey some of his findings. Specifically, we will consider the behaviors of finite register machines (simplified RAMs), simple Turing machines, and unidimensional mobile and cellular automata, including particular instances that are capable of universal computation.

3.1 Finite Register Machines

A finite register machine is a stripped-down version of the random-access machine (RAM) model introduced in Chapter 1, section 1.3; see in particular, Figure 1.1. The RAM accumulator is discarded and its memory is further restricted to a finite number of registers, each of which can store a *non-negative* integer of arbitrary size. The instruction set of Figure 1.1 is considerably reduced. Wolfram [2002, pgs. 97-102] describes machines with *only* two registers and two instructions, where the latter are defined and executed as follows:

- INCR <*operand*>: Increase by 1 the contents of the register specified in the operand field. (The operand is a positive integer identifying the register, and, in a machine with two registers, it is restricted to the values 1 and 2.) If this is the last instruction of the program then loop back to the beginning of the program, otherwise continue to the next instruction.

- DECJ <*operand*> <*label*>: Decrease by 1 the contents of the register specified in the operand field and then proceed to the instruction whose label is specified. If, however, the contents of the register are already zero and cannot be decremented further—recall registers store non-negative integers—then do nothing and proceed to the next instruction in the program. If the current instruction is the last instruction then loop back to the start of the program.

As in a RAM, the program of a register machine is a list of numbered instructions, where the associated numbers serve as labels for jump instructions, and it is executed under the control of a location counter that points to the "current instruction." Programs are composed from the foregoing instruction set and they are restricted, in total length, to upto T instructions, where T is an arbitrary positive integer.

The input tape of the RAM is also discarded, and the registers of the machine are assumed to be initialized to zero. An "output snapshot" of the contents of one or both registers is taken after each step, or executed instruction, of the program. Alternatively, the snapshot can be taken when a "write-condition" is satisfied. For example, the contents of the second register can be preserved in binary form, each time the contents of the first register are reduced to zero. Let us assume that the snapshots are written on successive lines of some output medium that can be viewed visually, by depicting '0' as the color 'white' and '1' as 'black.'

Consider the resulting two-dimensional output pattern. For simple programs composed from the above 2-instruction set and restricted to length T, where T is small, one might expect the output pattern also to be simple, i.e., to exhibit some overall regularity—convergent, repetitive or nested—and therefore behave predictably. For $1 \leq T \leq 7$, this is indeed the case. For example, for $T = 5$, Wolfram [2002, p. 99] observes that there are approximately a quarter of a million possible programs. For almost all such programs, the contents of the two registers exhibit simple repetitive behaviour. In two exceptions, nested behaviour is found. However, the situation changes

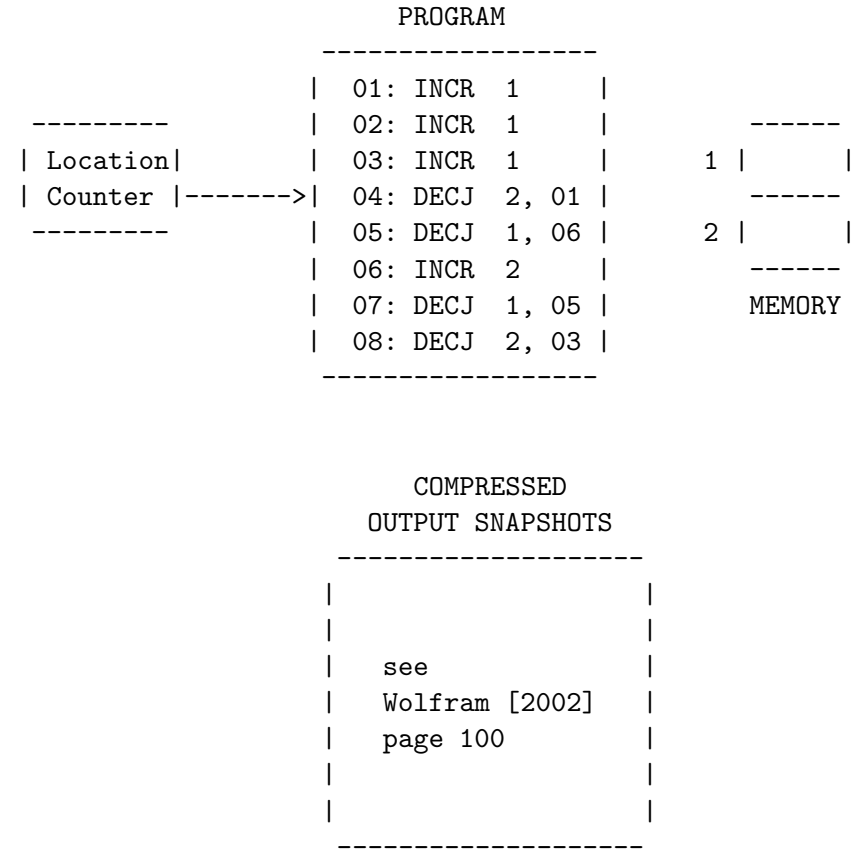


Figure 3.1: 2-Register, 2-Instruction Machine with $T = 8$

dramatically with further increase of permitted program length T. Considering specifically the case $T = 8$, Wolfram [2002, p. 100] finds that most programs of this short length continue to produce simple behaviour. But he now discovers the existence of exceptional, counter-intuitive instances of programs, albeit a small fraction of the total (approximately five in a million) that exhibit extraordinarily complicated and unpredictable behaviour. An example of such a program is given in Figure 3.1, along with reference within Wolfram [2002] to where its associated output shapshot pattern can be found. The latter is defined by the successive binary numbers in the second register when the number in the first register is zero, with '0' depicted as white and '1' as black. Randomness predominates within this pattern, and there is no observable regularity.

Since simple programs such as the one in the figure can be implemented very easily in any existing high or low-level computer languages, for example, MIXAL, similar conclusions can be drawn about them, namely, *seemingly trivial programs in any computer language can produce surprisingly complex and unpredictable behavior.*

3.2 Simple Turing Machines

The study of Turing machine behavior in Wolfram [2002] is premised on the following observation—see page 889 of his monograph (italics and footnotes ours):

> Since Turing's time, Turing machines have been extensively used as abstract models in theoretical computer science. But in almost no cases has the explicit behaviour of simple Turing machines been considered. In the early 1960s, however, Marvin Minsky[1] and others did work on finding the simplest Turing machines that could exhibit certain properties. Most of their effort was devoted to finding ingenious constructions for creating appropriate machines .... But around 1961 they did systematically study all 4096 2-state 2-color[2] machines and simulated the behavior of some simple Turing machines on a computer. They found repetitive and nested behaviour, *but did not investigate enough examples* to discover the more complex behaviour ...

[1] A renowned pioneer in the field of computer science.

[2] 'Color' is used as a synonym for 'symbol.'

Let us consider the Turing machine (TM) variant in section 1.4.2 for which the memory tape is two-way infinite, and the R/W head is *required to move* to an immediately adjacent cell at each iteration. Primitive program lines of such a TM are defined by quintuples of the form

$$(q, \sigma, \sigma', q', s),$$

where the machine has an alphabet of n_σ symbols σ (or σ'), and a set of n_q states q (or q'). The R/W head movement is defined by two choices, instead of the previous three, i.e., $s \in \{-1, +1\}$. There are upto $n_q n_\sigma$ quintuples in a valid Turing machine program, and it is easy to verify that the number of possible Turing machine programs is bounded by $(2n_q n_\sigma)^{(n_q n_\sigma)}$. Note that not all such programs are distinct or meaningful.

Henceforth, let us further restrict the machine to the binary alphabet, which can again be represented visually by the colors 'white' and 'black.' Following the execution of each step of the TM, preserve the memory tape and consider the resulting two-dimensional pattern of cells that is created. For convenience, we will assume the output 'snapshot' is specified *horizontally* and corresponds to the tape being rotated counter-clockwise (see Figure 3.2). Only cells that have been scanned so far by the R/W head are written, i.e., the portion of the memory tape between the topmost and bottommost movement, so far, of the R/W head. Alternatively, a *compressed form* of the output can be preserved, where a snapshot is taken whenever the number of cells scanned so far has *increased*, i.e., the R/W head has moved further up or down than it has ever been before. Again, one might intuitively expect the pattern of snapshots for simple Turing machines to exhibit simple behaviour, i.e., overall regularity. As noted in the foregoing quotation, all 2-state 2-symbol Turing machines do indeed exhibit simple repetitive or nested behavior. Wolfram [2002, page 79] observes that this is again true for all 3-state 2-symbol Turing machines. However, the situation changes dramatically when one or more additional states are introduced, corresponding to $n_q \geq 4$. Suppose the machine has just four states, denoted by 'N', 'S', 'E', and 'W', the points of the compass. The upper bound on the number of such 4-state 2-symbol Turing machines is 16^8 and each has a program length of 8, i.e., a set of 8 quintuples specified in any order. Assume also that the TM is always initiated in state 'N' and that all cells of the memory tape are initialized to '0' (white).

A variety of 4-state 2-symbol TM programs are described in Wolfram [2002], along with their interesting dynamical output. Repetitive and nested

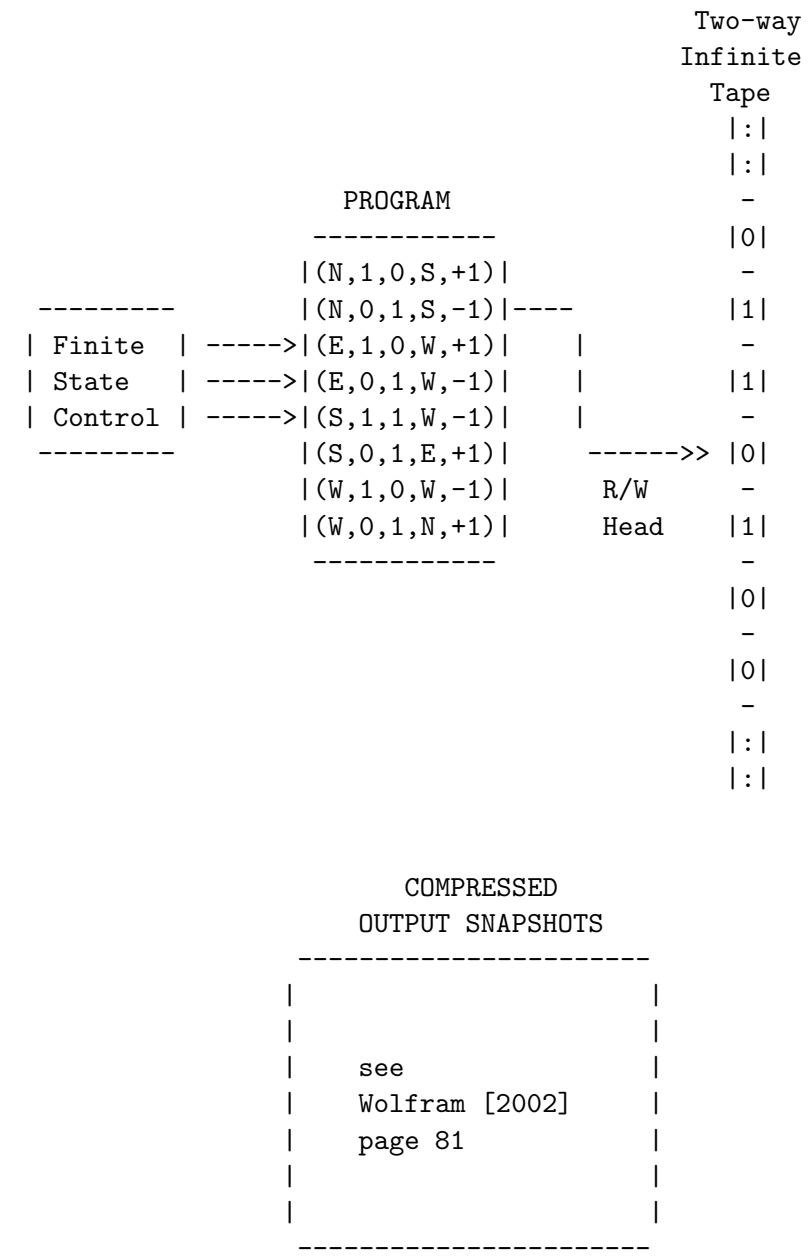


Figure 3.2: 4-state 2-symbol Turing Machine

patterns are the norm, but now Turing machines can be found whose behavior is highly unpredictable. An example of such a machine is shown in Figure 3.2 along with a reference to its output pattern (in the compressed form described above), which exhibits astonishing complexity. For further examples of the behaviour of simple Turing machines with four states (or more) and two symbols (or more), see Wolfram [2002, pages 78-81].

3.3 Unidimensional Mobile and Cellular Automata

A mobile automaton is a simplified Turing machine that dispenses with the states of the machine. Instead the machine is governed purely by the symbol in the cell under the R/W head—called the *active cell*—and the content of its two immediate neighbouring cells. It will be more convenient now to have the two-way infinite tape itself lie *horizontally*, so the behaviour of the mobile automaton is governed by the symbol in the active cell and the symbols in the two cells on its left and right. These three symbols determine the symbol that replaces the one under the head, which then moves to the adjacent cell on the left or right. Variants permit the content of the two neighbouring cells also to be altered and/or allow more than one cell to become active. Wolfram [2002; p. 76] makes the following observation (italics ours):

> By looking at many examples, a certain theme emerges: complex behavior almost never occurs except *when large numbers of cells are active at the same time*. Indeed there is, it seems, a significant correlation between overall activity and the likelihood of complex behavior.

This line of investigation therefore leads naturally to the study of unidimensional cellular automata, which are unidimensional mobile automata where *all* cells of the memory tape are active simultaneously. A R/W head, identifying the active cell in a mobile automaton, is therefore no longer needed. Assume again a 2-symbol, or binary, alphabet. A rule, or cellular automaton program, is defined by prescribing the symbol written to a cell from its own symbol, or color, and that of its two immediate neighbours. For example, "rule 30" is shown in the upper part of Figure 3.3. The eight, i.e. 2^3, possible combinations of three symbols are listed in the top line. For each combination, a symbol is written on the tape, and the particular choice of 8 written symbols defines the rule governing the cellular automaton. There are therefore 2^8 different rules, i.e., 256 possible undimensional, 2-symbol

cellular automata. Assume that an automaton is started from a tape with a single cell containing 1 and the rest 0.

Preserve the state of the memory tape at each step—the output snapshot that, as before, can be depicted displayed visually in a very convenient way by depicting '0' as the color white and '1' as black—and display the two-dimensional pattern that is formed from the sequence of output snapshots. One finds that different 2-symbol cellular automata produce a variety of interesting patterns, some regular, other nested, and yet others of astonishing complexity. These are presented, in detail, in Wolfram [2002], and their visual beauty is arresting. In particular, the output for the unidimensional cellular automaton governed by rule 30 is referenced in the lower part of Figure 3.3. Wolfram [2002, p. 28] makes the following observation regarding this output pattern:

> [Using] ... even the most sophisticated mathematical and statistical methods of analysis ... one can look at the sequence of cells directly below the initial black cell. And in the first million steps in this sequence, for example, it never repeats, and indeed none of the tests I have ever done on it show any meaningful deviation at all from perfect randomess.

Wolfram [2002, p. 27] describes this as "probably the single most surprising scientific discovery I have ever made."

A comprehensive study of the behaviour of cellular automata and a classification scheme into four basic classes of behaviour is given in Wolfram [2002, Chapter 3]. In particular, see pages 55-56 of this reference for a thumbnail sketch of the output pattern for each of the 256 possible 2-symbol, unidimensional, cellular automata.

Wolfram [2002, Chapter 11] on the notion of universal computation is a tour de force. In particular, it is shown that one of the foregoing cellular automata—governed by 'rule 110' depicted in Figure 3.4 and differing from rule 30 of Figure 3.3 in just three instances—is universal, i.e., capable of emulating any other possible cellular automaton when given the appropriate initial conditions; see also Cook [2004]. Furthermore, cellular automata can be designed to emulate a wide variety of other computational models. This yields universal Turing machines that are considerably simpler than the ones known previously. (The universality of rule 110 is used to derive a 2-state 5-symbol universal TM and it is conjectured there exists a 2-state 3-symbol TM that is universal.) Note that these results are not at variance with the

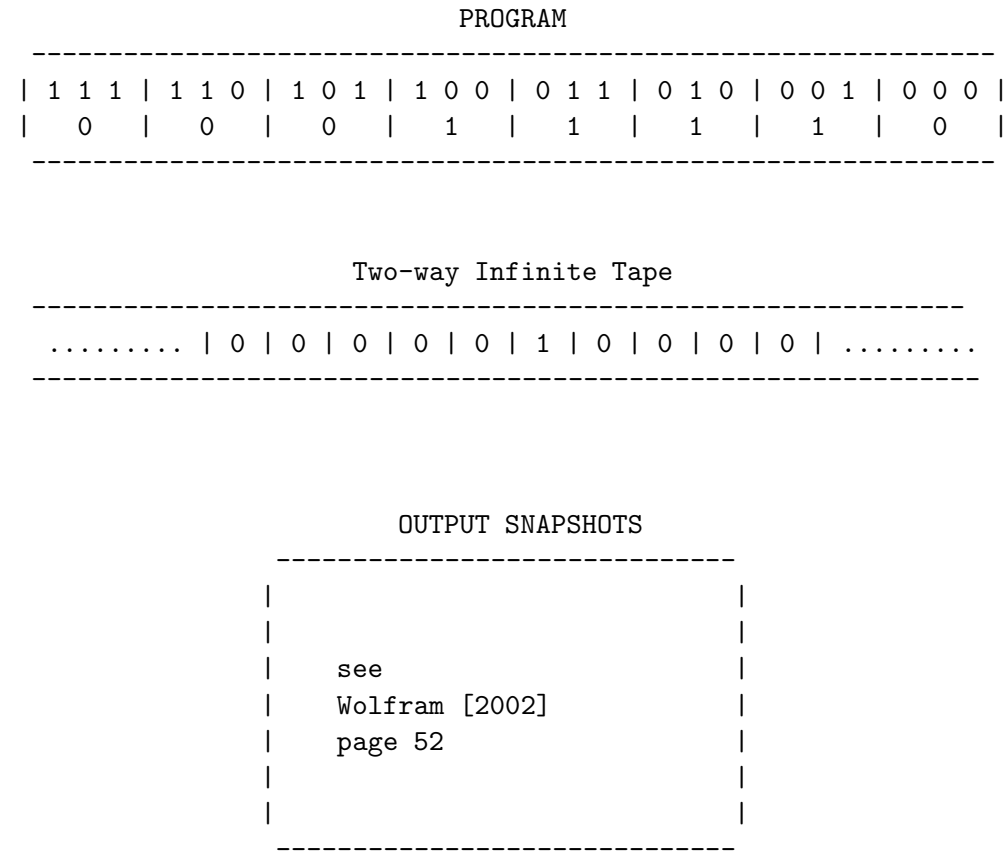


Figure 3.3: Cellular Automaton under “Rule 30”

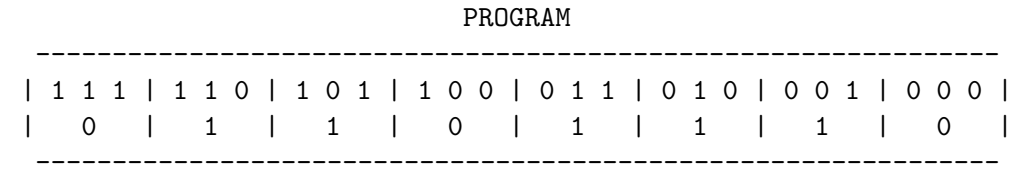


Figure 3.4: Rule 110

observations of Penrose on universal Turing machines, which are quoted at the end of Chapter 1.

3.4 Notes

Sections 3.1-3.3: This survey is based on Wolfram [2002, Chapters 2, 3, and 11]. Note that our interest is in the behavior of the models of computation *per se*. We do not address the wider objective of Wolfram's treatise, namely, the use of cellular and other automata to model the natural world and provide the foundation for a new kind of science. This is outside the scope of our monograph. Note also that *cellular automata* are the primary objects of study in Wolfram's treatise. Other machine models, in particular, register (RAM) and Turing machines, are studied in much less detail. In our brief survey of results quoted from Wolfram [2002], our emphasis has been just the reverse, i.e., our primary concern is regiser and Turing machines, in order to illustrate the behaviour of the models introduced in our earlier chapters.

By coincidence, each 'program' quoted in Figures 3.1-3.4 has precisely eight 'instructions.' These lists of machine instructions do not contain a 'STOP' instruction and they are therefore properly classified in Wolfram [2002] as 'programs.' (An algorithm is a program that halts after a finite, albeit unbounded, number of steps, on *every* possible input.)

Finally, note that only symbol-based computation is studied in Wolfram [2002]. However, one can conjecture that simple but artfully chosen magnitude-based, CD-RAM machines of Chapter 2 would exhibit correspondingly complex behaviour. This is an interesting avenue for further investigation.

Chapter 4

Essentialism versus Population-Thinking

This chapter is from the e-book by J.L. Nazareth (2010), *Algorithmic Science and Engineering within Scientific Computing: Foundations and Organization*, 128 pgs., www.math.wsu.edu/faculty/nazareth/asebook.pdf .

The renowned computer scientist, Richard M. Karp, characterizes the complexity of computational problems and algorithms as follows—see his foreword to Blum et al. [1998, p. v] (italics ours):

> Computational complexity provides a framework for understanding the cost of solving computational problems, as measured by the requirement of resources such as time and space. The objects of study are algorithms defined within a formal model of computation. Upper bounds on the computational complexity of a problem are usually derived *by constructing* and analyzing *specific algorithms.* Meaningful lower bounds on computational complexity are harder to come by, and are *not* available for most problems of interest.

An algorithm constructed, as mentioned above, specifically to solve a given problem often has a (relatively small) number of *key parameters* that determine its effectiveness. For instance, a quasi-Newton, or variable-meric, algorithm for finding a local minimum of a smooth nonlinear function selects a value for the so-called Broyden parameter, in order to implicitly or explicitly define its Hessian updating formula, and values for the so-called

Wolfe parameters that define the accuracy of its line search. It is natural to seek a "best," or ideal, choice for these key parameters, an approach to algorithm complexity that can be characterized as being "essentialist " in nature and motivated by the physical sciences.

There is an alternative, less familiar approach to algorithm formulation and efficiency that has its roots in the biological sciences and is captured by the following observation of the Nobel-Laureate Gerald Edelman, quoted from his landmark monograph (Edelman [1992, p. 73]):

> It is not commonly understood that there are characteristically biological modes of thought that are not present or even required in other sciences. One of the most fundamental of these is *population thinking*, developed largely by Darwin. Population thinking considers *variation* not to be an error but, as the great evolutionist Ernst Mayr put it, to be real. Individual variance in a population is the source of diversity on which natural selection acts to produce different kinds of organisms. This constrasts starkly with Platonic *essentialism* $\cdots$

The foregoing *population-based*, or Darwinian, perspective can be brought to bear on the construction of algorithms. The "best" values for an algorithm's key parameters are no longer sought. Instead, the variation within each such parameter is treated as intrinsic, or real, and utilized to construct a "Darwinian multialgorithm." This fundamental *algorithmic paradigm* was first proposed and investigated in Nazareth [2001a], [2003].

In this chapter, we will consider the multialgorithms approach within the context of a particular problem and algorithm: finding a local minimizing point of a function that is unconstrained, nonlinear, and not necessarily smooth, using a classic, direct-search algorithm due to Nelder and Mead [1965]. This algorithm has intuitive appeal and remarkable simplicity, and, although it is heuristic in nature, it continues to be one of the most widely used techniques of practical nonlinear optimization. The algorithm is very easy to describe and implement, and it thus provides an excellent arena for introducing the reader to, and contrasting, the two algorithmic paradigms mentioned above, i.e., the essentialist approach that seeks the "best," or ideal, choice for key parameters within the Nelder-Mead algorithm vis-à-vis the population-based approach that leads to a promising Nelder-Mead multialgorithm. Darwinian multialgorithms are ideally suited to implementation on a parallel computer as discussed at the end of the chapter.

4.1 The Nelder-Mead Algorithm

Direct-search algorithms for unconstrained minimization are based on an organized sampling of points, using either a systematic pattern or a statistical (random) procedure. Only the function is evaluated, i.e., algorithms in this family do not employ derivatives. Thus they can be effective even when the objective function $f(\mathbf{x}), \mathbf{x} \in R^n$, is noisy or exhibits other forms of non-smoothness. We will employ a direct search technique due to Nelder and Mead [1965].

4.1.1 Two Dimensions

Let us first consider the case $n = 2$. The unconstrained minimization problem can then be described, metaphorically, as the task faced by a person in a small boat on an opaque lake who seeks the point on the surface where the water is deepest—for convenience, assume the lake's bottom has a single local minimizing point, which is therefore also the global minimizer. At any chosen location on the surface, the vertical distance to the bottom can be measured, but this incurs a significant cost. It is this cost of obtaining *information* about the invisible topography of the bottom of the lake—the 'information-based complexity' of the associated unconstrained minimization problem—that makes the task algorithmically challenging. Otherwise, one could simply place a fine grid of points on the surface, measure the distance from the surface to the bottom at each grid point, and, by comparing them, find the deepest point of the lake.

The Nelder-Mead approach in this setting is childishly simple to describe. It proceeds as follows: measure the depth of the lake at three points that form the vertices of a (non-degenerate) triangle on the lake's surface and choose the point corresponding to the deepest of the three as the current iterate. Identify the vertex that is closest to the surface—the 'worst' vertex—and compute the *mid-point*, or centroid, of the other two vertices. Along the line (on the surface) joining the worst point and the centroid, one or more more additional samples are taken at fresh points that fall into some specified pattern, in order to identify an improving point. The triangle is then updated by replacing the worst vertex by this new point and the procedure is repeated.

Reverting to the equivalent problem of minimizing an arbitrary function in R^2, an iteration of the Nelder-Mead algorithm is summarized in Figure 4.1. Let the vertices of the triangle be $\mathbf{x}_1$, $\mathbf{x}_2$, and $\mathbf{x}_3$, with associated

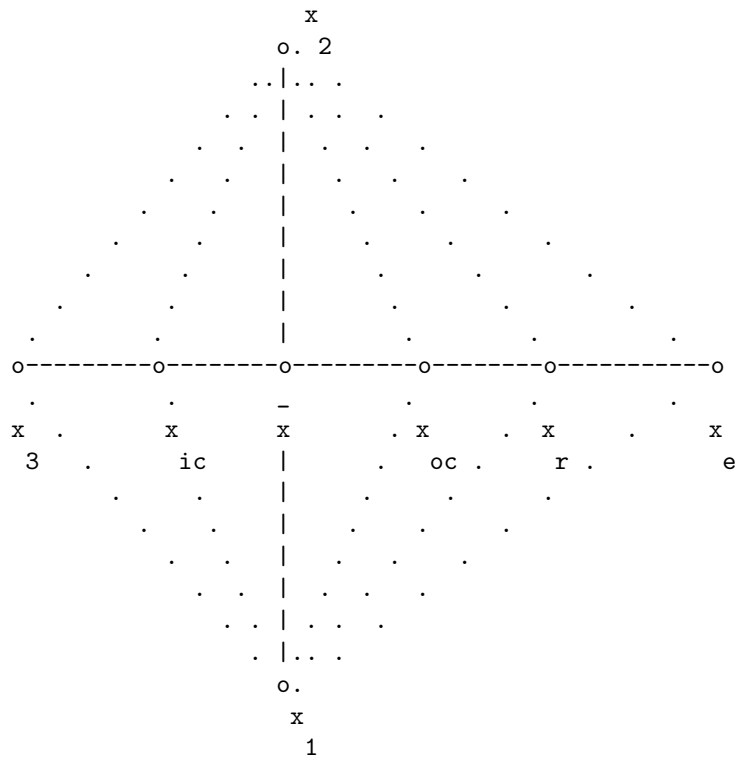


Figure 4.1: Nelder-Mead Algorithm

function values, f_1, f_2, and f_3 and assume the numbering of the vertices of the triangle was chosen so that $f_3 \geq f_2 \geq f_1$. The point $\mathbf{x}_\mathrm{r}$ is the reflection (typically isometric) of the point $\mathbf{x}_3$ in the centroid $\bar{\mathbf{x}}$ of the other two vertices of the triangle, namely, $\mathbf{x}_1$ and $\mathbf{x}_2$. The points $\mathbf{x}_\mathrm{e}$, $\mathbf{x}_\mathrm{oc}$ and $\mathbf{x}_\mathrm{ic}$ are extrapolation (expansion), outer contraction and inner contraction points, placed according to some preestablished pattern, as will be discussed in more detail below.

4.1.2 In General

For functions defined on R^n, the Nelder-Mead algorithm just outlined generalizes in a natural way as follows: at each iteration $k = 0, 1, ...$, it maintains an n-dimensional *simplex*, with vertex set $S^{(k)}$ defined by $n+1$ points $\mathbf{x}_1, ..., \mathbf{x}_{n+1}$ in R^n that are in general position.[1] The algorithm moves the worst vertex (largest function value) in the direction of the centroid of the remaining vertices according to predetermined step sizes in order to achieve descent, or, if it fails, it then shrinks the entire simplex towards its best vertex.

In the formal statement of this algorithm that now follows, $S \backslash \{\mathbf{x}_{n+1}\}$ denotes the remaining set of n vertices of the simplex when the point $\mathbf{x}_{n+1}$ is omitted.

Algorithm NM: Choose any set $S^{(0)}$ of $n+1$ vectors in R^n with vertices in general position. For $k = 0, 1, ...$, generate $S^{(k+1)}$ and $\mathbf{x}^{(k)}$ from $S^{(k)}$ as follows:

Step 0. (Initiate) Express $S = S^{(k)} = \{\mathbf{x}_i\}_{i=1}^{n+1}$ in ascending order of function value, i.e., $f_1 = f(\mathbf{x}_1) \leq \ldots \leq f_{n+1} = f(\mathbf{x}_{n+1})$. Define

$$\bar{\mathbf{x}} = (1/n) \sum_{i=1}^{n} \mathbf{x}_i. \tag{4.1}$$

Let $\mathbf{x}^{(k)} = \mathbf{x}_1$.

Step 1. (Reflection Step) Compute

$$\mathbf{x}_{\mathrm{r}} = \mathbf{x}_{n+1} + \tau(\bar{\mathbf{x}} - \mathbf{x}_{n+1}). \tag{4.2}$$

where τ is a *reflection* stepsize (typically, $\tau = 2$).

Then compute a new point according to the following three cases:

(1) ($\mathbf{x}_{\mathrm{r}}$ **has min cost**): If $f(\mathbf{x}_{\mathrm{r}}) < f_1$ go to Step 2.

(2) ($\mathbf{x}_{\mathrm{r}}$ **has intermediate cost**): If $f_n > f(\mathbf{x}_{\mathrm{r}}) \geq f_1$ go to Step 3.

(3) ($\mathbf{x}_{\mathrm{r}}$ **has max cost**): If $f(\mathbf{x}_{\mathrm{r}}) \geq f_n$ goto Step 4.

[1] This means the n vectors joining any chosen vertex and all other vertices are linearly independent.

Step 2. (Attempt Expansion) Compute

$$\mathbf{x}_{\rm e} = \mathbf{x}_{n+1} + \lambda(\bar{\mathbf{x}} - \mathbf{x}_{n+1}), \tag{4.3}$$

where λ is an *expansion* stepsize (typically, $\lambda = 3$).

If $f(\mathbf{x}_{\rm e}) \leq f(\mathbf{x}_{\rm r})$ then let $S^{(k+1)} = (S\backslash\{\mathbf{x}_{n+1}\}) \cup \{\mathbf{x}_{\rm e}\}$; else let $S^{(k+1)} = (S\backslash\{\mathbf{x}_{n+1}\}) \cup \{\mathbf{x}_{\rm r}\}$. **Exit**.

Step 3. (Use Reflection) Let $S^{(k+1)} = (S\backslash\{\mathbf{x}_{n+1}\}) \cup \{\mathbf{x}_{\rm r}\}$. **Exit**.

Step 4. (Perform Contraction) If $f(\mathbf{x}_{\rm r}) < f_{n+1}$ then compute

$$\mathbf{x}_{\rm oc} = \mathbf{x}_{n+1} + \mu(\bar{\mathbf{x}} - \mathbf{x}_{n+1}), \tag{4.4}$$

where μ is an *outer-contraction* stepsize (typically, $\mu = 3/2$), and if $f(\mathbf{x}_{\rm oc}) < f_{n+1}$ then let $S^{(k+1)} = (S\backslash\{\mathbf{x}_{n+1}\}) \cup \{\mathbf{x}_{\rm oc}\}$ and **Exit**;

Otherwise compute

$$\mathbf{x}_{\rm ic} = \mathbf{x}_{n+1} + \nu(\bar{\mathbf{x}} - \mathbf{x}_{n+1}), \tag{4.5}$$

where ν is an *inner-contraction* stepsize (typically, $\nu = 2/3$), and if $f(\mathbf{x}_{\rm ic}) < f_{n+1}$ then let $S^{(k+1)} = (S\backslash\{\mathbf{x}_{n+1}\}) \cup \{\mathbf{x}_{\rm ic}\}$ and **Exit**;

Step 5. (Shrink Simplex towards Best Vertex) Let $S^{(k+1)} = \mathbf{x}_1 + \theta(S - \mathbf{x}_1)$, i.e., the vertex set defined by $\mathbf{x}_1$ and the n points $\mathbf{x}_1 + \theta(\mathbf{x}_i - \mathbf{x}_1)$, $i = 2, \ldots, n+1$, where θ is a *shrink* stepsize (typically, $\theta = 1/2$), and **Exit**.

Exit, in the above algorithm, implies a return to **Step 0** to begin a fresh cycle. A suitable termination criterion is also required in an implementation.

The Nelder-Mead algorithm is heuristic in nature. It contains several exogenously-specified, arbitrary parameters $(\tau, \lambda, \mu, \nu, \theta)$ that define its main steps, and it possesses very little in the way of convergence theory. From a theoretical standpoint, the NM algorithm is primitive. On the other hand, as we have noted above, it is extremely useful in a practical sense, especially when n is modest in size and high accuracy in locating the local minimum is not needed. The algorithm requires only function values and it is not defeated by noise in the function value or an imprecise function evaluation procedure. These features and the algorithm's ease of implementation account for its great popularity with optimization practitioners within science and engineering.

4.2 The Essentialist Approach

The central issue within the NM algorithm is the choice of the five key quantities, $(\tau, \lambda, \mu, \nu, \theta)$. For instance, a standard choice is given in the statement of the algorithm above, namely,

$$\tau = 2; \quad \lambda = 3; \quad \mu = 3/2; \quad \nu = 2/3; \quad \theta = 1/2. \tag{4.6}$$

What is the "best" choice of these parameters? Useful insight is obtained by examining a "continuity" between the above NM algorithm and a well-known technique for *univariate* direct search.

4.2.1 Golden-Section Direct Search

In the univariate case, a popular algorithm is golden-section (GS) direct search. This algorithm contains no heuristic parameters, it is easily shown to be convergent at a linear rate when f is strictly unimodal, and it has a close relationship with Fibonacci search, which is known to be optimal in a minimax sense; see Luenberger [1984].

Golden-section direct search is summarized in Figure 4.2. Given boundary points A and E such that the minimum of the objective function is contained in the interval defined by them, two points B and C are placed so that $AC/AE = BE/AE = \alpha \equiv 1/\rho$, where $\rho = (\sqrt{5}+1)/2 \approx 1.618$ is the *golden ratio*. These quantities satisfy $\rho^2 = \rho + 1$, $\alpha^2 = 1 - \alpha$ and $\alpha^3 = 2\alpha - 1$, where $\alpha \approx 0.618$.

If the function value at C is no greater than that at B then one can reduce the interval containing the minimum to that defined by B and E. The use of the golden ratio ensures that $CE/BE = 1/\rho = \alpha$. Thus, only one additional point, say R, needs to be placed in the new interval with $BR/BE = 1/\rho$. An analogous statement holds for A and C when the value at C is no smaller than that at B, leading to the placement of an additional point S. Then the procedure is repeated. For a detailed description, see Luenberger [1984].

Figure 4.2 depicts the above set of points and relationships between them. Note, in particular, that $AB = BR$, i.e., the point R is an *isometric* reflection of the point A in B along the line AB. Analogously, $CE = SC$. Note also that $AB/BE = \alpha$.

When applied to a strictly unimodal function f, the golden-section procedure converges to the unique minimum of f at a linear rate.

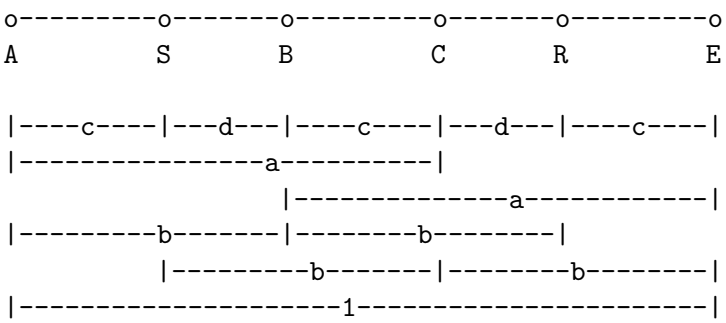


Figure 4.2: Golden Section Search: $a = \alpha$, $b = \alpha^2$, $c = \alpha^3$, $d = \alpha^4$

4.2.2 The "Best" Choice of Nelder-Mead Parameters

Superimpose Figure 4.2 on Figure 4.1 and make the following identification between points in the two figures:

$$\mathbf{x}_3, \mathbf{x}_{\rm ic}, \bar{\mathbf{x}}, \mathbf{x}_{\rm oc}, \mathbf{x}_{\rm r}, \mathbf{x}_{\rm e} \quad \text{with} \quad A, S, B, C, R, E.$$

This suggests the following golden-section (GS) choice of parameters within the Nelder-Mead algorithm:

$$\tau = 2; \;\; \lambda = \rho^2; \;\; \mu = \rho; \;\; \nu = \rho^{-1}; \;\; \theta = \rho^{-2}. \tag{4.7}$$

It yields the Nelder-Mead Golden-Section (NM-GS) algorithm proposed and studied by Nazareth and Tseng [2002]; see also Nazareth [2003; Chapter 6]. The following facts are left as exercises for the interested reader. They demonstrate the value of the GS choice, as will be discussed subsequently.

Exercise: Specialize Algorithm NM of section 4.1.2 to the *unidimensional* case where some of the steps of the NM algorithm become redundant. In particular, use the golden-section (GS) choice of parameters (4.7) to state a unidimensional NM-GS algorithm.

Exercise: Establish a *correspondence* that arises between a set of lattice points generated (potentially) by the golden-section search and a set of points utilized

(potentially) within the unidimensional NM-GS algorithm of the previous exercise.

The GS choice of parameters and the above correspondence of points diminishes the heuristic character of the Nelder-Mead variant (NM-GS) and makes the algorithm more amenable to theoretical analysis. In particular, convergence of the unidimensional NM-GS algorithm can be guaranteed on strictly unimodal univariate functions, in marked contrast to the approach taken in Lagarias et al. [1998] where considerable effort is expended to show convergence of the (original) Nelder-Mead algorithm in the univariate case and only on strictly convex functions. For details, see Nazareth and Tseng [2002].

Some numerical experience with the NM-GS algorithm can also be found in Nazareth and Tseng [2002]. The performance of this "best" version of the NM algorithm turns out to not differ significantly from the parameter choice (4.6). This is not surprising when one observes that (4.7) is approximated by

$$\tau = 2.00; \;\; \lambda = 2.56; \;\; \mu = 1.62; \;\; \nu = 1/1.62; \;\; \theta = 1/2.56, \qquad (4.8)$$

and the standard choice (4.6) is

$$\tau = 2.00; \;\; \lambda = 3.00; \;\; \mu = 1.50; \;\; \nu = 1/1.50; \;\; \theta = 1/2.00. \qquad (4.9)$$

Within the heuristic NM algorithm, these two choices are only marginally different, especially from the perspective of a practitioner. However, from a *theoretical* standpoint, Algorithm NM-GS is far more satisfactory than Algorithm NM with the parameter choice (4.6).

4.3 The Darwinian Approach

An alternative to the foregoing 'essentialist' approach is based on the fundamental Darwinian idea of population-thinking described at the beginning of this chapter. It rejects the notion of "best," or ideal, NM algorithm. Instead, the *variation* that enters into the key quantities $(\tau, \lambda, \mu, \nu, \theta)$ is used to define a population of NM algorithms for finding a local minimum of the function f. (A particular choice of the parameters, for example, the golden-section choice (4.7), defines a member of the population, i.e., a corresponding NM algorithm.) It is this population *taken as a whole*—the variation between its members is treated as real, or fundamental—that addresses the minimization of f. The "fitness" of any member of the population is only *implicit*

within the competition, *in parallel*, between the members of a given population of algorithms, and no attempt is made to explicitly define and optimize algorithm fitness.

4.3.1 A Nelder-Mead Multialgorithm

Consider a finite, fixed *population* of NM algorithms with n_P members. (The way to choose this population is considered in more detail later.) Each quintuple $(\tau, \lambda, \mu, \nu, \theta)$ will be called the G-type of the corresponding NM algorithm. The embodiment of the algorithm itself, whose parameters are defined by a G-type, will be called its corresponding P-type. The motivation for these terms comes from "genotype" and "phenotype," respectively.

Each algorithm is run in parallel from a given starting point, say, $\mathbf{x}$, and initial simplex, say, S, with known function values at its vertices, and it is allowed up to M_F calls to the function evaluation routine that returns information in the standard way—henceforth we say f-value for each such evaluation. (The multialgorithm is terminated if any member finds the solution.) We call M_F f-values the *lifespan* of one generation of the foregoing population. Since each member of the population is run in parallel, a *processor group* of n_P processors is required. The best iterate over the population, i.e., the one with the least function value, can then be identified and its associated simplex and function values at its vertices preserved. (The determination of this simplex is considered in more detail in the itemized discussion below.)

In addition to a single generation, at the same time consider another optimization path with two generations, each with a lifespan of $\frac{1}{2}M_F$ f-values. In this case, all members of the population can again be run in parallel for up to $\frac{1}{2}M_F$ f-values, the best iterate, say $\mathbf{x}_+$, and its associated simplex S_+ are taken as the initiating information, and the procedure is repeated for the second generation. This requires a second processor group of n_P processors. In a similar vein, consider additional cases of 3, ..., n_G generations. Thus, a total of $n_G * n_P$ processors, in n_G processor groups, is needed.

The foregoing is an example of a multialgorithm and it is summarized by the pseudo-code of Figure 4.3. Each *minor* cycle (after the first in its sequence) is initiated with the best iterate over the population and and its associated simplex. At the end of a *major* cycle, the best iterate *over all processor groups* is identified—denote it again by $\mathbf{x}$—and a simplex S is chosen to associate with it (see itemized details below given under the title "Choice

of Simplex"). The entire procedure is then repeated.

Exercise: Consider the extreme case $n = 2$ (see section 4.1.1), $n_G = 1$, and $n_P = 2$. Let the first G-type, or member of the population, be defined by (4.6), and let the second G-type be defined by (4.6) with $\tau = 2$ replaced by $\tau = 2.5$. Choose an equilateral starting simplex (triangle) with known function values at the three vertices.

Take $M_F = 5$ and assume that the two NM algorithms in the population always exit at Step 3 (successful reflection) during the execution of the foregoing Nelder-Mead multialgorithm. Depict the sequence of simplices developed by the two members of the population over a single major cycle.

We now consider some additional details.

Choice of Population

The population of n_P members, or G-types, whose size is governed by the number of available processors, is defined by making n_P choices for the quintuples $(\tau, \lambda, \mu, \nu, \theta)$.

One can proceed, for instance, as follows: define the first member of the population using the golden-section choice (4.7). Then define suitable intervals around each of the golden-section parameter settings and choose the components of the remaining $(n_P - 1)$ G-types randomly within these five intervals.

Another approach is to define the first G-type by (4.7) and the remainder by making $(n_P - 1)$ different "scalings" of the golden-section parameter settings, i.e., choose a factor s, e.g. $s = 1.5$, and define a corresponding member of the population, for example, as follows:

$$\tau = 2s; \;\; \lambda = \rho^2 s; \;\; \mu = \rho s; \;\; \nu = \rho^{-1} s^{-1}; \;\; \theta = \rho^{-2} s^{-1}. \tag{4.10}$$

Repeat the foregoing $(n_P - 1)$ times with different choices of s.

One can also combine the foregoing two techniques, and there are a variety of other approaches to population selection. When different G-types have the same value for the reflection parameter τ, note that the corresponding NM algorithms may follow identical paths for many iterations. Thus the particular characteristics of the NM algorithm of section 4.1.2 must be taken into account when defining the population of the multialgorithm.

A *key option* in the multialgorithm of Figure 4.3 is to begin with some initial population (whose size n_P is governed by the number of available

```
0: Given the optimization problem
0: Select an initial population
0: Specify the starting point x and initial simplex S
0: sequential_for until the solution is found
   major cycle:
   1: parallel_for each processor group (PROG)
     2: initiate at x (set x → x_+ and S → S_+)
     2: sequential_for each successive generation of PROG
        minor cycle:
        3: parallel_for each member of the population
           4: run the NM routine for the lifespan of PROG
              starting from x_+ and simplex S_+;
              if it converges then stop
        3: end_parallel_for population
        3: set x_+ to the best iterate over population
           and set S_+ to the associated simplex
     2: end_sequential_for minor cycle
   1: end_parallel_for PROG
   1: find the best iterate over all processor groups
      and set to x with an associated simplex S.
   1: option: evolve a new population
0: end_sequential_for major cycle
```

Figure 4.3: Pseudo-Code of an NM Multialgorithm.

processors) and to introduce evolutionary operations of *recombination* and *mutation*. Suppose the initial population is chosen as discussed above. At the end of a major cycle, as summarized by Figure 4.3, the *winner* from each of the processor groups is identified and its G-type retained. A winner can be defined, for example, to be the G-type that wins most frequently, i.e., attains the least function value, within the set of minor cycles that constitute the major cycle. Additional *descendents* can be generated by taking random pairs (parents) among the winners, and randomly mixing G-types within each pair. An additional set of *mutants* can be obtained, for example, by making (suitably bounded) random perturbations within G-types that are themselves randomly selected from the set of winners and descendents. The winners, descendents, and mutants can constitute a new population, again chosen to be of size n_P, for use in the next major cycle of the multialgorithm.

Choice of Lifespan and Processor Groups

The lifespan M_F is governed by the dimension of the problem, say n, and the number of processor groups n_G. For example, it can be defined as follows.

Let c_m be the least common multiple of the numbers $1, 2, \ldots, n_G$, and define L to be the smallest positive integer such that

$$\Gamma(\frac{n}{L})\frac{c_m}{n_G} \geq t\ n, \tag{4.11}$$

where $\Gamma(x)$ denotes the ceiling function, i.e., the smallest integer that exceeds or equals x, and $t \in R$ is an exogenously specified "adjustment" factor, say, $1 \leq t \leq 3$.

Then define the lifespan by

$$M_F = \Gamma(\frac{n}{L})c_m. \tag{4.12}$$

An illustration is given in section 4.3.2 below.

Choice of Simplex

In order to specify the simplex S used to initiate the multialgorithm of Figure 4.3, an evaluation of the function at each of the $n+1$ vertices of S is required. These evaluations can be performed *in parallel* using the available set of $(n_P n_G)$ processors.

When a member of the population is executed on a processor during a minor cycle, it may attain its allotted upper limit on the number of times

that it can call the function evaluation routine before it reaches a normal **Exit**. For instance, it could reach its limit before completing an expansion, contraction, or shrink operation within Algorithm NM of section 4.1.2. A simple remedy is to allow the algorithm to continue until it reaches its next **Exit**. It can then be terminated with a simplex that has complete function information at its vertices for use in a fresh initiation of a Nelder-Mead algorithm. In the case of a successful expansion or contraction, this remedy requires, at most, two additional calls to the evaluation routine. If the algorithm terminates prematurely within a shrink operation, the remedy may require upto n additional evaluations.

Various other more complex strategies can be devised that use available processors in parallel to prepare a suitable simplex S_+, at the end of a minor cycle, and S, at the end of a major cycle. Details will not be pursued here.

4.3.2 Implementation of the NM Multialgorithm

Suppose that 32 processors can be allocated to run members of the population within the NM multialgorithm of Figure 4.3, with one (or more) additional processors being available, as needed, for coordination purposes. The size of the population and the number of processor groups can then be chosen, for example, as follows: $n_P = 8$ and $n_G = 4$.

Suppose the function f has dimensionality $n = 10$ and let us use (4.11)-(4.12) to define the lifespan. The quantity c_m, the least common multiple of 1, 2, 3, and 4, is 12. Choose the adjustment factor $t = 1$. Then it is easily verified that $L = 2$ and $M_F = 60$. Within the multialgorithm and its four processor groups, this results in the following four cases: one generation with a lifespan of 60, two generations each with a lifespan of 30, three generations each with a lifespan of 20, and four generations each with a lifespan of 15, respectively.

Under the assumption that a function evaluation is the dominant cost and that, in relation to it, the overhead is insignificant, all processors within a processor group will terminate simultaneously at the conclusion of a minor cycle, and the entire set of processors will terminate simultaneously at the end of a major cycle. Thus the coordination of the parallel processors is easy to accomplish and greatly facilitates the implementation of the multialgorithm. Note that the simple technique described in the previous subsection to attain a normal **Exit** within Algorithm NM could introduce some discrepancy. This can be addressed by more complex procedures for repairing the simplex at the conclusion of a major or minor cycle.

A Beowulf cluster or other multiple-instruction multiple-data (MIMD) machine architecture and the Message-Passing Interface (MPI) programming environment (see Gropp et al. [1994]) provide a very convenient platform for implementing the NM multialgorithm. This platform can be *simulated* on a single processor machine using a standard programming language, for example, Fortran or C^{++}, and the performance of the multialgorithm recorded in *parallel-machine terms* as follows: suppose the multialgorithm terminates (at level 4 in Figure 4.3) within processor group k after the completion of N_M major cycles, N_m minor cycles within processor group k, and the use of N_l additional function evaluations within the last (incomplete) minor cycle. Then the measure of performance can be taken to be

$$N_M M_F + N_m \frac{M_F}{k} + N_l.$$

See Nazareth [2001a], [2003; Chapter 14] for the formulation and study of a multialgorithm similar to Figure 4.3 for minimizing smooth nonlinear functions, but based on a population of nonlinear conjugate gradient (CG) algorithms, instead of the NM algorithms discussed here. The implementation of the CG multialgorithm on an MIMD parallel computer was simulated on a single processor PC, and a set of standard test problems was used to evaluate performance in the parallel machine measure described above. The subsequent ease of transporting the simulated CG multialgorithm to a realistic MIMD/MPI programming environment was also discussed in the foregoing references.

Research Project: Using the CG multialgorithm study described in Nazareth [2001a], [2003; Chapter 14] as a blueprint, simulate an MIMD implementation of a fixed-population Nelder-Mead multialgorithm (section 4.3.1.-2.) on a single-processor PC, or similar machine, and evaluate its performance relative to the "ideal" NM-GS algorithm of section 4.2.2. Also explore the use of evolving-population techniques outlined in section 4.3.1 (at the end of subsection "Choice of Population").

Research Project: Building on experience gained with the above simulated implementation, develop an MPI-based implementation of a Nelder-Mead multialgorithm for nonlinear minimization on a Beowulf cluster or other available MIMD machine. Explore its subsequent transformation into user-oriented, derivative-free, quality software for minimizing noisy, nonlinear functions on a parallel computer, a counterpart of the popular Nelder-Mead software currently available for single processor machines.

4.4 Notes

Section 4.2: Other choices of parameters within Algorithm NM can be found in Nazareth and Tseng [2002].

Section 4.3: Multialgorithms take their metaphor from population-thinking and variation *in their broadest sense*; see, in particular, Mayr [1982], [2001] and Edelman [1992]. For a much more comprehensive and general discussion of the multialgorithms paradigm, including a contrast with standard evolutionary algorithms, see Nazareth [2001a], [2003].

Note also the school of biological thought, spearheaded by the eminent biologist Brian Goodwin, where genes within a genotype are conceived as setting "parameter values" that produce "component parts of the system, within a range of values," of a morphogenetic dynamical process that yields a phenotype; for a discussion within the popular literature, see Lewin [1992, p. 36], from which the foregoing quotations are taken. One can draw a parallel with our notion of G-type. But note that the G-type has no role in the construct of its associated P-type (algorithm). The latter has a fixed, exogenously-specified form, or morphology. The G-type governs the performance, or behavior, of this algorithm on a minimization problem (the 'environment'). In contrast, the much more complex genotype within evolutionary biology governs the aforementioned morphological dynamical process that produces an associated phenotype, and subsequently also governs the latter's additional development and performance within a given environment.

PART II

Context and Organization

Chapter 5

A Visual Icon for Sci-En-Math

This chapter is from the e-book by J.L. Nazareth (2010), *Algorithmic Science and Engineering within Scientific Computing: Foundations and Organization*, 128 pgs., www.math.wsu.edu/faculty/nazareth/asebook.pdf .

We first present a "big" picture, or visual icon, of the disciplines embraced by the natural sciences, engineering, and mathematics, the interfaces between them, and their modi operandi. This will permit us to clarify the nomenclature associated with these areas and the interrelationships between them, and thus enable us to set the stage for the discussion of algorithmic science & engineering (AS&E) in subsequent chapters.

5.1 Arenas and their Interfaces

Our visual icon is shown in Figure 5.1 where each of the three labelled bands will be called an *arena* of activity. Mathematics, the core arena, is classically partitioned into *algebra* (including foundations and the real-number system), *geometry*, and *analysis*—with many additional subdivisions and areas of intersection; see, for instance, the definitive, three-volume survey of Behnke et al. [1974]. Within this integrated discipline, "pure" mathematics, the heart of mathematics, is symbolized by the inner perimeter of the mathematics arena; "applied" mathematics by the outer perimeter, bordering science and engineering.

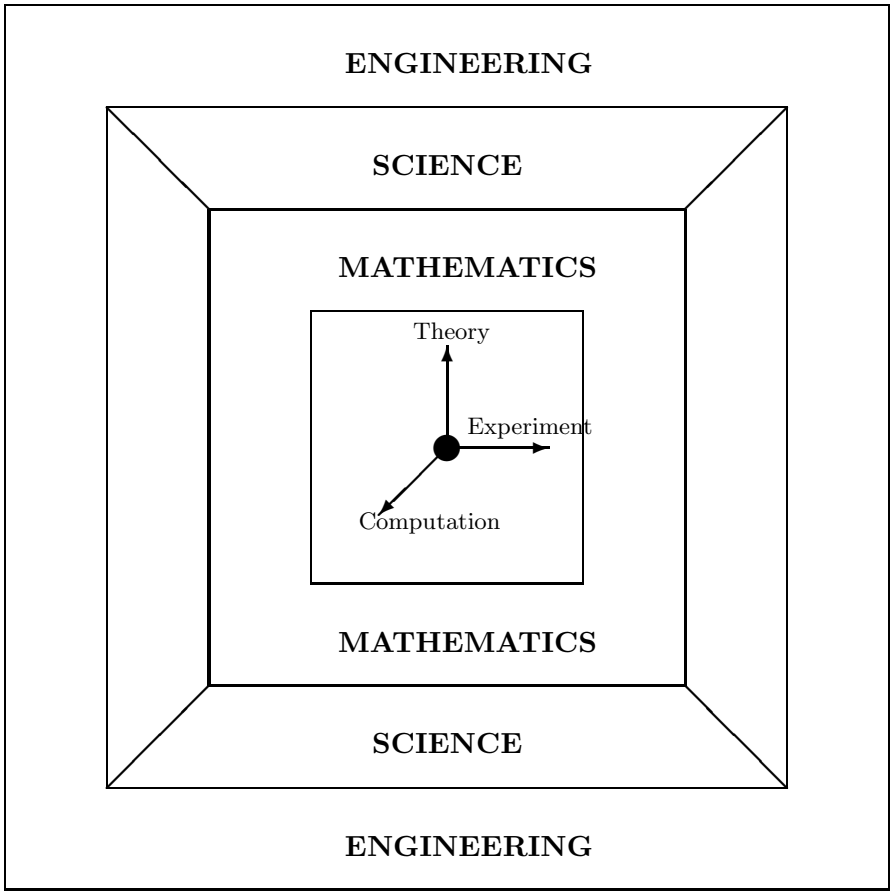


Figure 5.1: A Visual Icon

Surrounding mathematics is the arena of science,[1] often partitioned into the physical, chemical, and biological sciences. Further partitioning would identify the many specific sciences within each of these three broad subdivisions. The subfields of the sciences with the word "mathematical" prefixed, interfacing with mathematics, can be viewed as occupying the inner perimeter of the sciences arena, and "applied science" the outer perimeter, at the interface with engineering. In our schema, "mathematical sciences" and "applied mathematics" are overlapping designations, the former reflecting the perspective of science and the latter the perspective of mathematics. The region in which applied mathematicians and physicists find common ground is called mathematical physics; applied mathematicians and biologists intersect in the region called mathematical biology.

Engineering, the outermost arena of Figure 5.1, again has many subdivisions: mechanical, chemical, electrical, and so on. In this case, the inner perimeter symbolizes the interface with science, specifically the subject known as "engineering science," with the outer perimeter representing "applied" engineering in its usual sense: the development of engineered objects premised on science and mathematics, e.g., bridges, aeroplanes, and computers. The diagonal lines at the corner of the science arena, by explicitly linking the two arenas of Figure 5.1, serve as a reminder that engineering is also connected to mathematics. From the perspective of the engineer, the linking diagonals represent "mathematical engineering"; from the perspective of the mathematician, they represent the applicable region known as "engineering mathematics." We think they also add an appealing visual dimension to the figure.

Some academic fields naturally traverse arena boundaries. Computer science, for instance, has its origins in theoretical models of computation and complexity developed by mathematicians, which lie at its foundation. But the subject evolved subsequently, in tandem with the electronic computer, into a separate discipline that unites science and engineering. The field is now commonly designated computer science and engineering (CS&E) or electrical engineering and computer science (EE&CS), a union that arises because digital computers, both their hardware and software components, are themselves engineered objects. Likewise, the discipline of statistics traverses the boundary between the mathematics and science arenas, uniting mathematical probability (measure) theory and the science of statistical data.

[1] Here we consider only the so-called "hard sciences," in particular, the natural sciences.

5.2 Modi Operandi

The three axes at the center of Figure 5.1 depict the three main *modes*, or forms of investigation, that are employed within every branch of mathematics, science and engineeering—namely, the theoretical, experimental, and computational modes. Notice that "theoretical" does not necessarily imply the use of mathematics. For example, much of evolutionary biological theory is non-mathematical. "Computational" is relevant across the board, even to computer science, i.e., the term "computational computer science" is not redundant. For example, the properties of hashed symbol tables within computer science could be studied either theoretically or computationally (by simulation on a computer).

The central dot of the figure, from which the three axes emanate, can represent any subfield in any of the arenas. Evolutionary biology, for instance, can be investigated in theory, by experimentation in the laboratory, or via simulations on a computer. Within mathematics, cf. the recent, landmark monograph of Borwein and Bailey [2004] on the experimental/computational mode. Because the computer can be used as a "laboratory," the distinction between the experimental and computational modes can become blurred. The term "empirical" could then designate the union of these two modes, the counterpart to "theoretical," and be symbolized by the plane defined by the two horizontal axes of Figure 5.1.

5.3 Summary

The inner perimeter of each arena—mathematics, science, or engineering—represents the "pure" side of its associated subject, and the outer perimeter represents the "applied." Mathematics "applied" borders science (and engineering) "pure"; science "applied" borders engineering "pure." And every subfield within the arenas, symbolized by the central dot, has three main modi operandi that correspond to the three axes: theory, experiment, and computation.

We call the schema in Figure 5.1 an "icon," because it is akin more to a "compound symbol" for the complex interrelationships between mathematics, science, and engineering than a strict pictorial representation of them. It will serve to frame the discussion in the next chapter.

5.4 Notes

Section 5.1: Observe the back-and-forth shift in perspective in Figure 5.1: mathematics at the apex of a pyramid, flowing downward to science and engineering, vis-à-vis mathematics in a well, beneath science and engineering, and deriving motivation and direction from the other two arenas.

Sections 5.1-5.2: This discussion is derived from Nazareth [2006a].

Chapter 6

The Emerging AS&E Discipline

This chapter is from the e-book by J.L. Nazareth (2010), *Algorithmic Science and Engineering within Scientific Computing: Foundations and Organization*, 128 pgs., www.math.wsu.edu/faculty/nazareth/asebook.pdf .

Computational science and engineering (CSE) is the modern modus operandi whereby challenging problems of science and engineering are investigated via computation. As described in the previous chapter, it complements the traditional theoretical and experimental modes of investigation.

CSE is undergirded by the fields of *computer science* and *scientific computing*, which furnish its requisite tools. Computer science relies primarily, although not exclusively, on "discrete" algorithms; scientific computing, likewise, on "continuous" algorithms; and the foundations for the two fields are provided by the symbol-based and magnitude-based models of computation described in Part I.

In this chapter, we discuss the foregoing "pillar" of support for computational science and engineering. This enables us to present the rationale, or *raison d'etre*, for an emerging, core discipline within scientific computing, to which the name algorithmic science & engineering (AS&E) will be attached.

In the concluding section, we give a blueprint for creating informally structured, small-scale, in-house AS&E research centers (ASERCs) for the scientific study of real-number algorithms, as a practical step towards the nurture of the emerging AS&E discipline.

6.1 Discrete Algorithms of Computer Science

In an invaluable collection of articles discussing the philosophy of computer science, Knuth [1996, p. 5] makes the following observation:

> My favorite way to describe computer science is to say that it is the study of algorithms. ..... Perhaps the most significant discovery generated by the advent of computers will turn out to be that algorithms, as *objects of study*,[1] are extraordinarily rich in interesting properties; and, furthermore, that an algorithmic point of view is a useful way to organize information in general. G.E. Forsythe has observed that "the question 'What can be automated?' is one of the most inspiring philosophical and practical questions of contemporary civilization".

However, one need only look at one of the standard computer science texts, for example, the multi-volume classic starting with Knuth [1968], or a thematic overview of the subject, for example, Hopcroft and Kennedy [1989], to see that the emphasis in computer science is primarily on algorithms that arise within the construction and maintenance of the tools of computing, i.e., *algorithms of discrete mathematics*, and not on algorithms that arise when computers are used for real-number computation. Indeed, in the above quoted work, Knuth notes the following:

> The most surprising thing to me, in my own experiences with applications of mathematics to computer science, has been the fact that so much of the mathematics has been of a *particular discrete type*[2] .... .

We see that despite the early, de jure goals of computer science to be more all-embracing, a *de facto alliance developed between computer science and discrete, or combinatorial, mathematics*, underpinned by the Turing-machine model (Garey and Johnson [1979]). Simultaneously, close ties have been forged between academic departments of electrical engineering, computer science, and computer engineering under the banners EE&CS or CS&E. Thus, in an apt and prescient definition of Newell, Perlis and Simon [1967], modern computer science is best characterized, nowadays, simply as *"the*

[1] Italics ours.

[2] Italics ours. The mathematical needs that typify the algorithms of computer science are gathered together in Graham, Knuth and Patashnik [1989].

study of computers" and the phenomena surrounding them in all their variety and complexity.

The study of discrete, or combinatorial, algorithms is one of these phenomena, which lies at the heart of the field of computer science and is sufficiently rich to unify its various branches. It is, in turn, undergirded by the symbol-based, random-access and Turing computational models described in Chapter 1.

6.2 Continuous Algorithms of Scientific Computing

Consider, on the other hand, the following oft-quoted, illuminating remarks of John von Neumann, made in 1948 in his Hixon Symposium lecture (see his collected works edited by Taub [1961]) at the dawn of the computer era:

> There exists today a very elaborate system of formal logic, and specifically, of logic applied to mathematics. This is a discipline with many good sides but also serious weaknesses .... Everybody who has worked in formal logic will confirm that it is one of the technically most refractory parts of mathematics. The reason for this is that it deals with rigid, all-or-none concepts, and has very little contact with the continuous concept of the real or the complex number, that is, with mathematical analysis. Yet analysis is the technically most successful and best-elaborated part of mathematics. Thus formal logic, by the nature of its approach, is cut off from the best cultivated portions of mathematics, and forced onto the most difficult mathematical terrain into combinatorics.
>
> The theory of automata, of the digital, all-or-none type as discussed up to now, is certainly *a chapter in formal logic.*[3] It would, therefore, seem that it will have to share this unattractive property of formal logic. It will have to be, from the mathematical point of view, combinatorial rather than analytical.

Problems that require real-number, or continuous, algorithms and associated mathematical software for their solution are common in all areas of science and engineering. They are addressed, in particular, within the

[3]Italics ours.

fields of numerical analysis and optimization, which are, in turn, often subsumed under the rubric of *scientific computing*. As the name "numerical analysis" suggests, this subject came to flower[4] within mathematics as the region of analysis where number representation in numerical processes and the cumulative effects of rounding error play a central role. This aspect is exemplified by the classic monographs of Wilkinson [1963], [1965]. However, the subject today enjoys a much broader interpretation—the study of numerical algorithms and underlying mathematical models as described by Trefethen [1992]—and it has diffused widely into most branches of the natural sciences and engineering. The field of optimization has enjoyed a similar success—its classic is the monograph of Dantzig [1963]. These monographs of G.B. Dantzig and J.H. Wilkinson, which coincidentally appeared within the span of 1963-65, are the crown jewels of *finite-dimensional* optimization and numerical analysis. The two interrelated fields of *numerical analysis* and *optimization*, over spaces of finite or infinite dimension, have both advanced rapidly, in tandem with computer hardware developments, during the past five decades. Key facets of the two fields and their encompassing arena of scientific computing are as follows:

- *mathematical modeling*, i.e., casting real applications into one of a wide array of basic mathematical models. In the case of numerical analysis, these models generally take the form of linear or nonlinear equation-solving (including algebraic, ordinary, and partial differential equations), defined over finite- or infinite-dimensional spaces. In the case of optimization they generally involve the maximization or minimization of objective functions over feasible regions defined by equality and/or inequality constraints, again in finite or infinite dimensions;

- *algorithmics*, i.e., the invention and implementation of efficient algorithms for solving numerical models, including the development of high-quality computer software; and

- the *applied analysis of models and algorithms*, in particular the analysis of solution sets and other properties of, and duality relationships between, various classes of models; and the analysis of algorithms from the standpoints of convergence, complexity, and numerical stability.

[4]The field originated much earlier with the mechanical aids to computation and the making of tables: interpolation, divided differences, etc. For this reason numerical analysis specialists have sometimes refered to themselves in the past as "number engineers." But, nowadays, the name "numerical algorist" is perhaps more accurate—see Trefethen [1992].

While the mathematical modeling aspects of numerical analysis and optimization are often best carried out in close partnership with the areas of science or engineering where applications originate (see, for example, Tung [2007]), the twin fields within scientific computing have retained a strong *algorithmic core* and an associated mathematics of computation. Undergirding this retained area are the real-number computational models of Chapter 2, which seek to liberate scientific computing from the intellectual grip of the symbol-based, universal Turing machine model, but simultaneously to recognize the pivotal role of the electronic digital computer. For further background, see the panel discussion entitled "Does numerical analysis need a model of computation?" in Renegar, Shub, and Smale [1996].

6.3 The Supporting Pillar for CSE

Figure 6.1 summarizes the foregoing two subsections. The new mode of investigation termed computational science and engineering (CSE),[5] whereby challenging, often very large-scale problems of science and engineering are tackled through computation, is undergirded by computer science and scientific computing. As noted earlier, the former is nowadays closely allied with electrical engineering (EE&CS) or computer engineering (CS&E), and the latter implicitly embraces mathematical and engineering computing.

Computer science relies, first and foremost, on discrete, or combinatorial, algorithms; see, in particular, the encyclopedia of algorithms edited by Kao [2008]. Scientific computing relies primarily on continuous, or real-number, algorithms that arise, in particular, within numerical analysis and optimization; see, for instance, the comprehensive collection given in Press, Teukolsky, Vetterling, and Flannery [1992]. Discrete algorithms, in turn, are premised on the classical Turing model, continuous algorithms on more recent real-number models of computation; and the entire edifice rests on the foundations of computational mathematics (FoCM)—see Shub [2001]—and general BCSS-type models of computation.

Note that important problems of scientific computing involve an interplay between the two halves of the figure, as highlighted by the $\Leftrightarrow$ symbol. The same holds true for computer science. For example, *combinatorial scientific computing (CSC)* has emerged recently as a hybridizing, or bridging, discipline of this type. Likewise, recent "bit-models" of computation—see, in

[5] As observed in the previous chapter, this modus operandi complements the long-established theoretical and experimental modes.

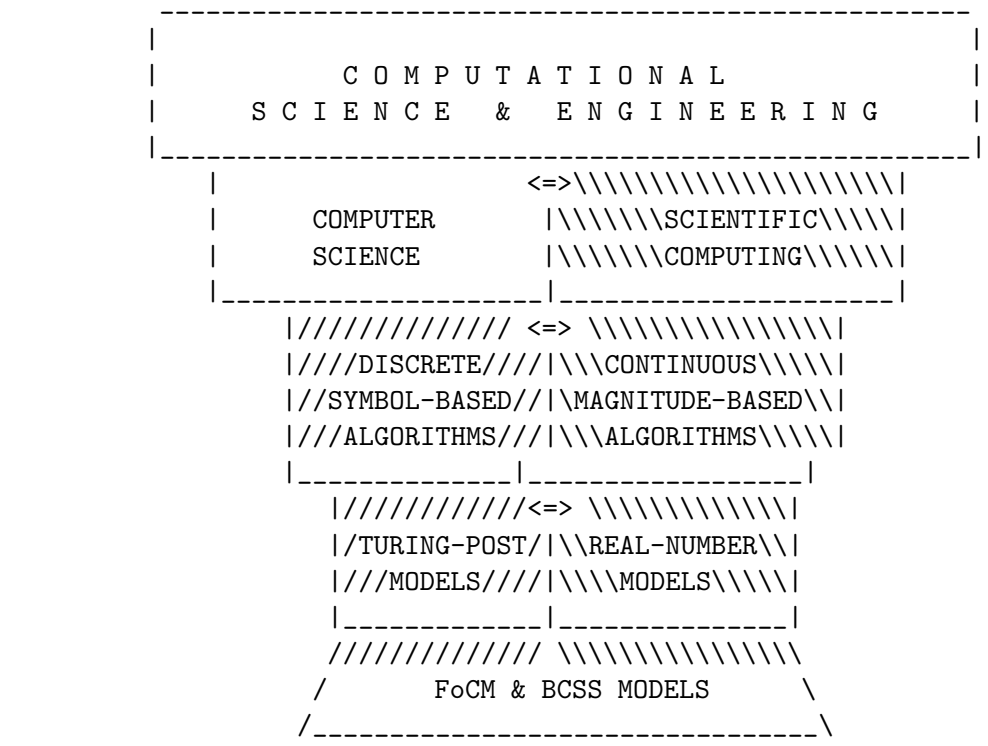


Figure 6.1: Supporting "Pillar" of CSE

particular, Braverman and Cook [2006] and references cited therein—are in the Turing tradition, but oriented towards the needs of scientific computing over the reals. Figure 6.1 thus captures only the extremes of the two fields, but it accurately portrays the *vertical hierarchy of support* for computational science and engineering.

6.4 Algorithmic Science & Engineering (AS&E)

Algorithmic science is an emerging discipline associated with the shaded portion of Figure 6.1 where the emphasis is on computation of a *numeric nature.*

'Algorithmic' identifies the *objects* that lie at the heart of the discipline, and 'science' is attached in its fullest sense. Thus, as with *any* scientific field, the discipline has three main modes of expression:

- *theoretical* algorithmic science establishes the convergence, complexity and stability of algorithms and provides the foundations for underlying models of numerical computation;
- *experimental* algorithmic science studies algorithms on the bench, for example, through the use of physical models, and investigates the numerous manifestations of algorithms within nature, for example, algorithmic processes involving annealing, spin glasses, or evolution based on genetic recombination and mutation; and
- *computational* algorithmic science uses the electronic digital computer and the large body of associated software tools, for example, Mathematica, Matlab, etc., as a laboratory for the study of algorithms.

'Empirical' is used sometimes to combine the 'experimental' and 'computational' modes. Furthermore, an *integral* part of the discipline involves the practical implementation of algorithms, and the development of reliable and robust mathematical software, i.e., the 'technology' of the subject. For this reason, the word 'engineering' is also attached and the discipline delineated as follows:

> Algorithmic science & engineering (AS&E) is the theoretical and empirical study and the practical implementation and application of symbol-based and magnitude-based algorithms of a numeric nature, continuous or discrete, that arise within scientific computing and the modern modus operandi known as computational science and engineering.

A full-fledged AS&E discipline would help to integrate algorithmic research for solving numerical models, which is currently conducted within numerous fields, often in a disparate manner, with little intersection. This would have significant benefits for research. In addition, significant educational advantages would be gained from the recognition of the AS&E discipline and *its incorporation into the university curriculum in a suitable way.* Requirements for graduate qualifying examinations could be set appropriately, to achieve the right balance between mathematical and computer science emphases. Coherent training could be provided for much-needed

real-number computing professionals within academia and industry, producing graduating students that would be well-positioned in the job market. Mathematical software engineering would be a respectable activity within AS&E and would be officially recognized and rewarded, as is the writing of (non-mathematical) software within computer science.[6] An AS&E discipline could comfortably embrace a full spectrum of orientations, ranging from theorem provers to mathematical software developers, all within the unifying rubric of computing, algorithms, and software. It is worth again recalling Knuth's original vision of computing as the "study of algorithms" as quoted in section 6.1. This vision could be realized by a full-fledged AS&E at the heart of scientific computing, a counterpart to the algorithmic core of the well-established field of computer science & engineering, thus creating a situation whereby the study of discrete (combinatorial-based) and continuous (number-based) algorithms would receive more equal emphasis.

The AS&E discipline within scientific computing, as delineated above, interfaces with a wider domain to which it can contribute in a fundamental way. For instance:

- *grand challenge problems of computing*, e.g., algorithmic investigations of protein folding (see the discussion at the end of section 2.1 in Chapter 2) or the intriguing near-optimality of the genetic code (De Duve [2002, p. 73]);

- *new computing paradigms*, e.g., the dramatic paradigm-shift proposed by Wolfram [2002];

- *cross-fertilizations*, e.g., the symmetric algebraic eigenproblem (Parlett [1980]) viewed from the standpoint of optimization and nonlinear equation-solving, or the synergy between linear and nonlinear conjugate gradient-related research (Adams and Nazareth [1996]);

- *foundations of computational mathematics*, e.g., BCSS-type models of Chapter 2, section 4.3, analog models over the reals in continuous time (Moore [1996]), and recent "bit-models" for scientific computing (Braverman and Cook [2006]).

[6] Writing a large piece of mathematical software is as challenging a task as proving a convergence theorem—harder, perhaps, because the computer is an unrelenting taskmaster.

6.5 Blueprint for an ASERC

An immediate and practical way to nurture the emerging discipline is to create informally-structured AS&E research centers, organized in-house on a small scale, in order to encourage *interaction among algorithm-oriented researchers from diverse areas* and facilitate the exploration of algorithmic issues *at a root level*. An AS&E research center, or ASERC, could find a home within any one of a number of settings—within a university department of computer science, mathematics, applied mathematics, mathematical sciences, or operations research; within a national research laboratory; or (as a subcenter) within an existing mathematical or computational sciences research center—and it can pursue the following types of activities:

Blueprint for an ASERC within Scientific Computing

- Provide an *intellectual staging-ground* that brings together algorithm-oriented researchers from optimization and numerical analysis, as well as from other scientific and engineering disciplines where numerical algorithms play a prominent role. An ASERC can also encourage participation of graduate research students and research-oriented undergraduates.
- Organize a *regular seminar*—options are monthly, biweekly, or weekly meetings, as determined by need—where local and outside speakers could create a useful forum for the periodic discussion of research into algorithms of a numeric nature, continuous or discrete, and their applications within computational science & engineering.
- Conduct (occasional, not-for-credit) *courses*, ranging from introductory minicourses aimed at a broad audience, for example, Nazareth [2004a], to advanced seminar-type courses suitable for graduate research students.
- Provide a *repository of information about root-level algorithmic experimentation* and an environment within which specific case studies could be pursued. Illustrations of such activities are given in the next chapter.
- Organize *meetings* akin to the pioneering Pacific-West Algorithmic Science Meeting [2000]; see also the proceedings of its antecedent research

conference, Adams and Nazareth [1996].

- Create a local ASERC *website* and distribute a *newsletter* to participants via the internet.
- In cooperation with other ASERCs, spearhead the creation of an AS&E *research journal* to foster the discipline.
- Explore opportunities for raising the *modest level of funding* needed to support the foregoing activities from a governmental agency, a private foundation, or a computer-related company.

The foregoing activities would require only a relatively minor commitment of resources, both physical space and other infrastructure and administrative support, and the organizational effort itself could be kept to a minimum. ASERC computational needs could be satisfied by small-scale, in-house systems already in place (networked PCs, Beowulf clusters), because computer programming activities would be typically along lines described by Trefethen [2005]. In contrast, *highly-organized, peta-scale computation centers*—multi-disciplinary research teams, parallel machines with tens or even hundreds of thousands of processors, and large capital investments for software and hardware—lie at the opposite end of the computing spectrum and serve the high-end needs of computational science and engineering (CSE); see the upper portion of Figure 6.1. They would derive considerable sustenance from the activities outlined above that more informally-organized ASERCs can better pursue. Examples of such activities are described in the next chapter.

6.6 Notes

Sections 6.1-6.2: This discussion is derived from Nazareth [2003; chapter 15].

Section 6.3: Note that CSE is the commonly-used acronym for the *modus operandi* of computational science and engineering, and CS&E for the *discipline* of computer science & engineering. Sometimes the two acronyms are exchanged.

It is also worth noting a dual usage for the name *scientific computing*. The term "computing" broadly connotes algorithms and software, but the qualifying adjective "scientific" is ambiguous, used in two senses: "of, relating to, or exhibiting the principles of science" and "as it applies to the sciences." The first interpretation,

in conjunction with "computing" as "algorithmics," yields the term "scientific algorithmics," and it can then be identified more closely with "algorithmic science." The second, oft-used interpretation corresponds to "computational science and engineering' as delineated in Chapters 5 and 6. Thus the term 'scientific computing' can run the gamut from AS&E, on the one hand, to CSE, on the other.

Section 6.4: This discussion is derived from Nazareth [2006a]. The earlier characterization of algorithmic science & engineering in Nazareth [2003, p. 222], [2006a] used the term "real-number algorithms" in place of "algorithms of a numeric nature," and noted that "algorithms of discrete mathematics are not excluded ..., but they take second place to the algorithms of continuous mathematics." The one we have given here more accurately embraces complex- and discrete-number algorithms as well. It also implicitly highlights the fact that much of the algorithmic territory within computer science is non-numeric and is *not* covered by the AS&E umbrella, while simultaneously recognizing a significant intersection between the two disciplines. Since the study and implementation of algorithms that are primarily of a non-numeric nature is so clearly identified with the field of computer science & engineering (CS&E), a separate name that highlights its algorithmic core, for example, algorithmic CS&E, non-numeric AS&E, or AS&E within computer science, seems superfluous. In other words, the name "computer science & engineering" already well represents *both* the core algorithmic area and the wider field. It therefore seems reasonable to use the name "algorithmic science & engineering" for the discipline at the heart of the more nebulous subject of scientific computing. Alternatively, those who prefer could explicitly term the core discipline "AS&E within scientific computing," as in the title of this monograph.

A convenient home for the emerging algorithmic discipline can be envisioned under a rubric of *mathematical science & engineering*, encompassing the three key components itemized in section 6.2, namely, *mathematical modeling*, *AS&E within scientific computing*, and the *applied analysis of models and algorithms*. This would be a natural counterpart to the traditional, tripartite subdivision within *mathematics*, namely, *geometry*, *algebra* (including foundations and real-number systems), and *analysis*; see the three corresponding volumes of Behnke et al. [1974] and also the discussion in Gowers et al. [2008, pgs. 1-3]. 'Mathematical science & engineering' is often fully or partially realized under an alternative departmental banner, for example, 'applied mathematics,' 'numerical analysis,' or 'combinatorics & optimization.'. At the other end of the spectrum, one can also envision a home for a more broadly defined AS&E at the intersection of the twin fields comprising an EE&CS department, or within a computer science or a CS&E department, thereby embracing traditional symbol-based (combinatorial, discrete) computation and *reembracing* magnitude-based (continuous, numerical) computation.

Chapter 7

Illustrations of AS&E

This chapter is from the e-book by J.L. Nazareth (2010), *Algorithmic Science and Engineering within Scientific Computing: Foundations and Organization*, 128 pgs., www.math.wsu.edu/faculty/nazareth/asebook.pdf .

In this concluding chapter, we present two case studies that are illustrative of the AS&E discipline delineated in Chapter 6. The first, a case study in *algorithmic science*, describes a root-level exploration of quasi-Newton algorithms for unconstrained minimization in situations where computer storage is at a premium. The second, a case study in *algorithmic engineering*, utilizes a simple, but realistic, resource planning application (in the area of forestry) to describe "engineered" decision support software developed in Nazareth [2001b] and made available in the form of a PC-executable, demonstration program.

7.1 Case Study in Algorithmic Science

We return to the nonlinear unconstrained minimization problem introduced in Chapter 4 and assume, additionally, that the nonlinear objective function $f(\mathbf{x})$, $\mathbf{x} \in R^n$, is smooth. Let $\mathbf{g}(\mathbf{x})$ denote the gradient vector of f at $\mathbf{x}$.

The quasi-Newton algorithm for finding a local minimum of f was discovered by Davidon [1959] and refined by Fletcher and Powell [1963]. (It was originally called the variable metric algorithm, but nowadays the generic name "quasi-Newton" is used instead.) The algorithm has the same informational needs, namely, function and gradient values, as the classical steepest-descent algorithm of Cauchy. But, in terms of efficiency, it is much

closer in performance to the classical Newton's algorithm, while simultaneously dispensing with the latter's need for second derivative information, or Hessian matrices. The quasi-Newton algorithm has the same computer storage needs as Newton's algorithm, namely, $O(n^2)$, which is used to maintain approximations to the Hessian matrix or its inverse. This stands in contrasted to the $O(n)$ needs of Cauchy or conjugate gradient (CG) algorithms.

The Davidon-Fletcher-Powell (DFP) breakthrough of the early nineteen-sixties inspired a huge outpouring of new research results on quasi-Newton updates, over an extended period. During the course of this research, a wide variety of different QN updates were discovered and their theoretical properties investigated. In addition, a variety of techniques to enhance efficiency and numerical stability were studied. For an overview, see Dennis and Schnabel [1983]. The BFGS update (also called the complementary-DFP) was found to have the strongest set of properties in totality and to be among the most effective in practice. Today it is the recommended quasi-Newton algorithm for nonlinear minimization.

Let $\mathbf{s}_i = \mathbf{x}_{i+1} - \mathbf{x}_i$ be a non-zero step and $\mathbf{y}_i = \mathbf{g}_{i+1} - \mathbf{g}_i$ be its the associated change of gradient, where $\mathbf{g}_i = \mathbf{g}(\mathbf{x}_i)$ and $\mathbf{g}_{i+1} = \mathbf{g}(\mathbf{x}_{i+1})$. Assume the step $\mathbf{s}_i$ is such that $b_i = \mathbf{y}_i^T \mathbf{s}_i > 0$. Let $\mathbf{W}_i$ be a given symmetric, positive definite approximation to the inverse Hessian matrix. For reference below, a family of quasi-Newton updates of $\mathbf{W}_i$, over the foregoing step and gradient change, is defined as follows:

$$\mathbf{W}_{i+1} = \left[\left(\mathbf{I} - \frac{\mathbf{s}_i \mathbf{y}_i^T}{b_i} \right) \mathbf{W}_i \left(\mathbf{I} - \frac{\mathbf{s}_i \mathbf{y}_i^T}{b_i} \right)^T + \frac{\mathbf{s}_i \mathbf{s}_i^T}{b_i} \right] + \omega_i \mathbf{w}_i \mathbf{w}_i^T, \tag{7.1}$$

where $\omega_i \in R$ is a parameter,

$$\mathbf{w}_i = \frac{\mathbf{s}_i}{b_i} - \frac{\mathbf{W}_i \mathbf{y}_i}{c_i}$$

with $b_i > 0$ defined above, and $c_i = \mathbf{y}_i^T \mathbf{W}_i \mathbf{y}_i > 0$.

The parameter choice $\omega_i \equiv 0$ yields the BFGS update. The family (7.1) embraces many well-known quasi-Newton updates, each defined by an appropriate choice for ω_i. Several such updates are competitive with one another on standard test problems—see, for example, Zhu [1997, pg. 42, Table 2.2] for a numerical study using the Minpack test problems and their standard starting points. The BFGS is generally among the best performers in practice, and, as noted above, for both theoretical and computational reasons, it is today the recommended choice.

7.1.1 Memoryless Quasi-Newton Algorithms

Suppose computer storage is at a premium and only $O(n)$ storage is available. The *memoryless QN algorithm* removes the need for storing a matrix by instead updating a simple matrix, typically the identity matrix $\mathbf{I}$, over the most recent step. This updated QN approximation to the (inverse) Hessian is represented *implicitly* by storing the vectors that define the update. An associated search direction at the current iterate is obtained from this memoryless QN approximation, a line search is performed, and the process repeated. The conceptual algorithm proceeds as follows:

Initialization:
$\mathbf{x}_1$ = arbitrary
$\mathbf{g}_1$ = gradient of f at $\mathbf{x}_1$
$\mathbf{d}_1 = -\mathbf{g}_1$

Iteration i:
1. $\alpha_i^* = \text{argmin}_{\alpha \geq 0} f(\mathbf{x}_i + \alpha \mathbf{d}_i)$; $\mathbf{x}_{i+1} = \mathbf{x}_i + \alpha_i^* \mathbf{d}_i$
2. $\mathbf{g}_{i+1}$ = gradient of f at $\mathbf{x}_{i+1}$
3. $\mathbf{s}_i = \mathbf{x}_{i+1} - \mathbf{x}_i$; $\mathbf{y}_i = \mathbf{g}_{i+1} - \mathbf{g}_i$; $b_i = \mathbf{y}_i^T \mathbf{s}_i$; $c_i = \mathbf{y}_i^T \mathbf{y}_i$, and $\mathbf{w}_i = \frac{\mathbf{s}_i}{b_i} - \frac{\mathbf{y}_i}{c_i}$.
4. $\mathbf{d}_{i+1} = -\left[\left[\left(\mathbf{I} - \frac{\mathbf{s}_i \mathbf{y}_i^T}{b_i}\right)\left(\mathbf{I} - \frac{\mathbf{s}_i \mathbf{y}_i^T}{b_i}\right)^T + \frac{\mathbf{s}_i \mathbf{s}_i^T}{b_i}\right] + \omega_i \mathbf{w}_i \mathbf{w}_i^T\right] \mathbf{g}_{i+1}$

In step 1, assume the minimizing point chosen is the one closest to $\mathbf{x}_i$ on the half-line through $\mathbf{x}_i$ along direction $\mathbf{d}_i$. We call the foregoing algorithm "conceptual," because an exact minimizer cannot, in general, be found by a finite procedure. To make the algorithm "implementable," an improving point $\mathbf{x}_{i+1}$ is found at step 1 by using a *line search* procedure, normally based on quadratic or cubic polynomial fitting (suitably safeguarded) with an associated exit criterion to prescribe the accuracy of the line search.

Observe that the update in step 4 is the QN update (7.1) of the identity matrix $\mathbf{I}$. The matrix within the square barackets is *not* formed explicitly. Instead, the search direction $\mathbf{d}_{i+1}$ is found by a sequence of scalar product operations and vector additions. The term "memoryless" comes from the fact that update information developed at prior iterations of a quasi-Newton algorithm is discarded. Since the raison d'etre for quasi-Newton updating is the gathering of approximate Hessian, or curvature, information over a sequence of steps, the notion of discarding earlier information and developing the update afresh over only the most recent step seems counterintuitive at first sight. Is the idea viable?

7.1.2 Numerical Experiments

Root-level numerical experiments to investigate the foregoing question are described in Nazareth [2006b], [2004b], and they will be summarized in this section.

A representative set of memoryless QN algorithms based on six well-known QN updates were implemented *uniformly* using the same line search routine and exit criterion. (Relatively high accuracy in the line search was prescribed by setting an exit tolerance suitably.) These six updates were defined by appropriate choices of the parameter ω_i. Their details need not concern us here, and they will identified simply as M-QN1, ..., M-QN6. In order to provide a benchmark for the experiment, two standard conjugate gradient (CG) algorithms—the so-called Polyak-Polak-Ribiere (PPR) and an aperiodically-restarted variant called PPR$^+$—were also implemented, and in like manner. (The notes at the end of the chapter give some additional information on the implementation.)

All algorithms were run on a testbed of four well-known problems using standard starting points. Again their detailed definitions are not needed here and they will be identified as PROB-1, ..., PROB-4. Each algorithm was terminated using an identical criterion, and performance was measured, in the standard way, by the number of calls to the function/gradient evaluation routine, or f/g-calls, i.e., the number of requests for problem information. The test results reported in Nazareth [2006b], [2004b] are duplicated in Table 7.1. Each entry is the number of f/g calls. An entry '*' in the table indicates a failure to find a solution within an upper limit of 1000 f/g calls. The first six lines give the results for the memoryless QN algorithms and the last two lines give the benchmark results for the CG algorithms.

Observe that only one of the memoryless QN algorithms is viable, the others having failed 50 percent of the time or more. The exception, M-QN2, is the memoryless QN algorithm based on the BFGS update—henceforth identified as M-BFGS. Note also the general similarity between the results for M-BFGS and the PPR and PPR$^+$ algorithms.

The results in Table 7.1 support the intuition that discarding the memory in a QN algorithm is an idea without merit. Why is the M-BFGS algorithm an exception?

Algorithm	PROB-1	PROB-2	PROB-3	PROB-4
M-QN1	*	207	*	648
M-QN2	94	294	240	71
M-QN3	*	207	*	*
M-QN4	*	207	*	*
M-QN5	*	207	*	*
M-QN6	72	188	*	*
PPR	103	309	668	43
PPR$^+$	69	154	403	63

Table 7.1: Performance of Algorithms

7.1.3 Conceptual BFGS-CG Relationship

Let us return to the conceptual memoryless QN algorithm of section 8.1.1 and use the BFGS choice of update, namely, $\omega_i \equiv 0$, and two relationships that follow from the use of an *exact* line search, namely,

$$\mathbf{g}_{i+1}^T \mathbf{d}_i = 0; \quad \mathbf{g}_i^T \mathbf{d}_{i-1} = 0.$$

It is easily verified that the conceptual M-BFGS algorithm generates a sequence of iterates $\mathbf{x}_i$ that are *identical* to the iterates of the conceptual PPR algorithm, which is defined as follows:

Initialization:
$\mathbf{x}_1$ = arbitrary
$\mathbf{g}_1$ = gradient of f at $\mathbf{x}_1$
$\mathbf{d}_1 = -\mathbf{g}_1$

Iteration i:
1. $\alpha_k^* = \text{argmin}_{\alpha \geq 0} f(\mathbf{x}_k + \alpha \mathbf{d}_k)$; $\quad \mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k^* \mathbf{d}_k$
2. $\mathbf{g}_{i+1}$ = gradient of f at $\mathbf{x}_{i+1}$
3. $\beta_i = \frac{\mathbf{g}_{i+1}^T \mathbf{y}_i}{\mathbf{g}_i^T \mathbf{g}_i}$
4. $\mathbf{d}_{i+1} = -\mathbf{g}_{i+1} + \beta_i \mathbf{d}_i$.

For further discussion, see Nazareth [1976], [1979], Buckley [1978a]. Equivalence to a conjugate-gradient (CG) algorithm does *not* hold when other memoryless quasi-Newton updates are substituted for the M-BFGS. Thus

only M-BFGS inherits the desirable properties of the conceptual CG algorithm; see Kolda et al. [1998].

The conceptual 'BFGS-CG relationship' provides the necesary insight, lacking in the 'discarding of memory' perspective, into the effectiveness of the M-BFGS quasi-Newton algorithm. It suggests that memoryless QN algorithms based on updates *other than the BFGS* are unlikely to be effective in practice.

7.1.4 Implementable M-BFGS and Extensions

The observations of the previous section are given credence by the simple numerical experiments described earlier. These experiments necessarily employ *implementable* versions of M-QN and CG algorithms, where an inexact (but fairly accurate) line search is used. In this case, the memoryless BFGS algorithm does not reproduce exactly the iterates of the PPR algorithm. The implementable M-BFGS algorithm has the advantage that a direction of descent is guaranteed under much weaker termination criteria on the line search than the criteria prescribed for an (implementable) nonlinear CG algorithm.

Performance of M-BFGS can be further enhanced as follows:

- *Preconditioning:* The Oren-Luenberger strategy based on updating a suitably *scaled* identity matrix is recommended when implementing the memoryless BFGS algorithm; see Luenberger [1984].

- *Restarts:* The performance of the PPR conjugate gradient algorithm is improved, in both theory and practice, by using aperiodic restarting. The PPR^+ variant mentioned above replaces Step 3 of the algorithm of section 7.1.3 as follows:

$$\beta_i = \max\left(\frac{\mathbf{g}_{i+1}^T \mathbf{y}_i}{\mathbf{g}_i^T \mathbf{g}_i}, 0\right),$$

  and further refinements are given by Powell [1977]. They can be incorporated into the M-BFGS algorithm as well.

- *Efficiency:* Overhead, in terms of both computer storage and costs associated with linear algebra operations, was not of concern in the root-level experiments described in section 7.1.2. They simulated a situation where the cost of information, namely, function/gradient evaluation,

is dominant and therefore only the numbers of requests for information were counted. In the experiments, each memoryless update of the identity matrix over the current step was stored simply as a full matrix approximating the inverse Hessian, and the associated quasi-Newton search direction was obtained by a matrix-vector multiplication. In practice, these operations would be completely reformulated so the matrix in Step 4 of the algorithm of sections 7.1.1-2. is represented implicitly using vectors, and the search direction is obtained by a sequence of scalar product operations and vector additions.

- *Extensions:* A further *key enhancement* is to use BFGS updates over several recent steps as proposed by Nocedal [1980], Liu and Nocedal [1989]. This limited-memory BFGS algorithm (L-BFGS) has proved to be an efficient and versatile algorithm that can exploit additional computer storage when available. It generally outperforms the PPR^+ CG-algorithm and it supplanted earlier variants that employed quasi-Newton updates, which are developed over a limited number of previous steps and used to *precondition* the nonlinear CG algorithm; see Buckley [1978b] and Nazareth [1979].

  L-BFGS employs refinements to reduce storage and computational overhead, i.e., matrices approximating inverse Hessians are defined implicitly by storing vectors, and *recurrence relations* are used to obtain search directions efficiently. Oren-Luenberger preconditioning is also employed.

The foregoing L-BFGS features come in the way of performing numerical experiments analogous to those of section 7.1.2 when other QN updates are substituted. Root-level numerical studies of other limited-memory quasi-Newton (L-QN) algorithms are more conveniently performed using full matrices, which are computed at each iteration by updating an identity or scaled identity matrix over a set of preserved steps and associated gradient changes, in conjunction with matrix-vector multiplications. This would increase only the overhead costs, leaving f/g counts unaltered. For L-QN algorithms that develop updates over a few prior steps, typically between 2 and 5, it is reasonable to conjecture that numerical results analogous to those of Table 7.1 for M-QN algorithms would be obtained, suggesting, in turn, that limited-memory algorithms based on updates in the Broyden family, *other than the BFGS*, are also unlikely to be effective when almost all memory is discarded. However, confirmation of this conjecture requires further numerical study.

Results of root-level explorations such as the above are not generally available and reported in the literature, being sidelined by the needs of more immediate practicality. They could profitably be pursued within an ASERC organized along lines described in the previous chapter. For other examples of ASERC-oriented research, see the projects at the end of Chapter 4.

7.2 Case Study in Algorithmic Engineering

7.2.1 Motivation

Consider the following very simple resource-planning problem that stems from a real-life forestry project in southern Tanzania described by Dykstra [1984]. It involves finding the optimal cutting schedule for a woodland of *Pinus patula* grown on a 30-year rotation, in order to maximize total physical yield over the rotation period. The project planner wants to evaluate the possibility of thinning when the stands are 10 years old and again at 20 years. Thinning has two benefits: (1) small and diseased trees are harvested that would normally die before the stand reaches rotation age and whose volume would thus be lost without thinning and (2) there is more light and nutrients for the trees that remain after thinning so that diameter growth (and log value) are accelerated. The planner wants to simplify the instructions given to the work crews as much as possible. Thus, for each potential thinning age, he/she plans to consider only three possibilities: (1) no thinning at all, (2) a light thinning to remove about 20 percent of the standing volume, and (3) a heavy thinning to remove about 35 percent of the standing volume.

Initially, the stand is of age 10 and the average standing volume is 260 cubic meters per hectare. Figure 7.1 summarizes the set of possible planning alternatives.

Each node of the network corresponds to a state of the forest resource and is defined by the age of standing timber (in years) and the average volume of standing timber (in cubic meters per hectare). For example, the first node S_1 is defined by age 10, volume 260. The quantity associated with each arc, a transformation from one state to another, gives the volume of timber that is removed at the start of the ten-year interval, after which the stand grows to the volume for the node at the end of the arc. These quantities are given in a table below Figure 7.1.

At age 30 the stand is clear felled and replanted and grows[1] again to

[1]Note that growth is not assumed to be linear and this causes no difficulty in specifying

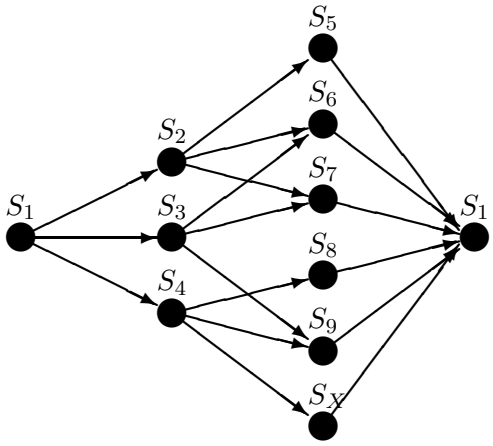


Figure 7.1: Network of alternatives for timber-harvesting example

State	Age	Volume
S_1	10	260
S_2	20	650
S_3	20	535
S_4	20	410
S_5	30	850
S_6	30	750
S_7	30	650
S_8	30	600
S_9	30	500
S_X	30	400

Transformation	Benefit
$S_1 \rightarrow S_2$	0.0
$S_1 \rightarrow S_3$	50.0
$S_1 \rightarrow S_4$	100.0
$S_2 \rightarrow S_5$	0.0
$S_2 \rightarrow S_6$	150.0
$S_2 \rightarrow S_7$	200.0
$S_3 \rightarrow S_6$	0.0
$S_3 \rightarrow S_7$	100.0
$S_3 \rightarrow S_9$	175.0
$S_4 \rightarrow S_8$	0.0
$S_4 \rightarrow S_9$	75.0
$S_4 \rightarrow S_X$	150.0
$S_5 \rightarrow S_1$	850.0
$S_6 \rightarrow S_1$	750.0
$S_7 \rightarrow S_1$	650.0
$S_8 \rightarrow S_1$	600.0
$S_9 \rightarrow S_1$	500.0
$S_X \rightarrow S_1$	400.0

state S_1. The objective is to choose the rotation that maximizes the amount of timber produced over the planning period, or *horizon*, of 30 years.

In the network of Figure 7.1, it is easy to enumerate the different planning alternatives or *paths* in the network from start to finish, and establish that there are nine paths in all. The optimal alternative is thus easily found by inspection. One can simply list each path and sum up its total production of timber over the three time periods, thus identifying the best one.[2]

This optimal solution is as follows:

- Age 10: Take no action.
- Age 20: Execute a light thinning cut, removing 150 cubic metres per hectare.
- Age 30: Clear fell, removing 750 cubic meters per hectare.

No other sequence of actions, or planning alternative, produces more timber over the 30 year period. It is also evident that the optimal solution is achieved by using a single planning alternative on the entire timber area. For example, if two different alternatives were used, one on part of the area and the other on the remainder, and if one of these alternatives had a higher total production of timber than the other, then obviously one would do better by replacing the less productive alternative by the more productive.

Let us extend the problem and network of planning alternatives by permitting clear felling and replanting, in addition to thinning, at the end of each of the 10 year planning intervals. For example, the initial stand can be clear felled and replanted, yielding 260 cubic metres per hectare and a new arc, or transformation, to the state S_1 a the end of the first interval. Similarly, the states S_2, S_3, and S_4 can be clear felled and replanted, yielding benefits of 650, 535, and 410, respectively, and transformations to state S_1 at the end of the second interval. (At the end of the third interval, note that other states in addition to S_1 are now attainable.) The extension permits rotations of lengths 1, 2 and 3, so it is natural to extend the total period of planning to 6 intervals (a multiple of rotations of lengths 1, 2 and 3), and to

---

the network. Linearity enters from the assumption that a decision action, for example, heavy thinning on, say, 14 hectares, will yield precisely twice the amount of timber obtained by heavy thinning on 7 hectares.

[2]Alternatively, one can use a simple dynamic programming procedure as in Dykstra [1984].

seek the optimal schedule. It can be demonstrated without difficulty that a 20-year rotation is optimal, which we leave to the following exercise.

Exercise: Extend the network and the tables of Figure 7.1 when clear felling and replanting are permitted at the start of each interval, as described above. Find the best planning alternative for *single* rotations of length 1, 2, and 3. Use the foregoing to show that the best planning alternative over a planning horizon of 6 intervals is a rotation of length 2.

Additionally, let us place environmental restrictions on clear felling. Let us allow some initial flexibility, but place restrictions so that at the end of interval 3, and thereafter at the end of each interval, at least 30 percent of the forest is in states S_5, S_6, or S_7, i.e., contain trees of age 30 years and standing timber of density at least 650. In the presence of these additional environmental constraints, the optimal solution is a *combination* of planning alternatives, each applied to a portion of the woodland, and it can no longer be found easily by inspection. The problem must be appropriately modeled and the optimal solution found by a suitable algorithm implemented on a computer, i.e., we need an "engineered" decision support system.

7.2.2 The $D_L P$ System, DLPFI Language, and DLPDEMO

The $D_L P$ optimization model and decision support system developed in Nazareth [2001b] has broad applicability to the planned development of natural and renewable resources in the presence of constraints on costs, benefits, environmental factors, and so on. It provides a domain specific language for defining models—so-called $D_L P$ Format Interface (DLPFI) language—and an optimization algorithm based on Dantzig-Wolfe decomposition and dynamic programming (hence the acronym $D_L P$). The algorithm and its underlying model were "engineered" into a prototype implementation for which a demonstration program, DLPDEMO, was provided in Nazareth [2001b] on attached CD-ROM. We will describe its application to the foregoing simple, yet realistic, forest planning problem.

Let us consider first the case with three intervals, maximization of total physical yield, and no additional constraints, for which the optimal solution was stated in the previous section. The specification of the model in the aforementioned DLPFI language is shown in Figure 7.2, and the input is largely self-explanatory. Each line in the figure that begins with a "?" is called a keyword record, and it corresponds to a main section of the prob-

lem data. The latter is then specified on zero, one, or more data records, continuing to the next keyword record. Thus the problem is given the name DYKSTRA. It has a planning period consisting of 3 planning intervals. Only one resource class is specified, which is given the name SAOHILL (derived from the region where the problem originated).[3] The states and actions for class SAOHILL are identified by a unique name of upto eight characters. The list of STATES is specified and the choice of state names is evident from the tables in Figure 7.1. Then the list of harvesting ACTIONS is specified: the actions names, NC, LT, HT, and CF, correspond to 'no cut,' 'light thin,' 'heavy thin,' and 'clear fell,' respectively. The next section specifies the network of planning alternatives and it reflects the information in the tables of Figure 7.1.[4] The LOCAL section specifies constraints on the resource class. It consists of the INITIAL record stating that the resource class consists of 1000 hectares. Additional constraints on desirable or undesirable states can also be specified in a CONSTRAINTS section (see below). Finally GLOBAL constraints on benefit (and cost) across all resource classes and the problem objective are identified. In the simple example, there are no benefit constraints and the objective is to maximize the timber production. Further detail on the DLPFI language is given in Nazareth [2001b].

The corresponding $D_L P$ output is shown in Figure 7.3 and defines the optimal solution (stated in the previous section). It too is largely self-explanatory. The ALTERNATIVES keyword gives the number of planning alternatives in the optimal solution. Next, the CLUSTER keyword gives the proportion of the resource class prescribed for the associated optimal planning alternative. The first number in the INITIAL keyword that follows restates the total size of the resource class, and the second number specifies the number of hectares prescribed for the alternative (the total size multiplied by the foregoing proportion). The subsequent records then give the planning alternative's sequence of actions, their (costs and) benefits per hectare, and the states attained, all obtained from the DLPFI specification. In our case, a single planning alternative is prescribed for the entire resource class and corresponds to the optimal solution mentioned earlier. The GLOBAL CONSTRAINTS section is not relevant for our simple application. The optimal OBJECTIVE value—a total yield of 900000 cubic metres—is

[3] In general, the $D_L P$ model permits many resource classes.

[4] The $D_L P$ system permits costs as well as benefits to be associated with each arc. Since costs do not enter into the simple forestry model, all associated numbers are 0. The second number of each data record specifies the benefit. The states before and after the action are also identified.

```
?PROBLEM    DYKSTRA
?PERIODS
   3
?CLASSES
  SAOHILL
?NAME  SAOHILL
?STATES SAOHILL
  A10V260 A20V650 A20V535 A20V410 A30V850 A30V750 A30V650
&  A30V600 A30V500 A30V400
?ACTIONS SAOHILL
  NC LT HT CF
?NETWORK SAOHILL
     A10V260 NC 0. 0. A20V650
     A10V260 LT 0. 50. A20V535
     A10V260 HT 0. 100. A20V410
     A20V650 NC 0. 0. A30V850
     A20V650  LT 0. 150. A30V750
     A20V650 HT 0. 200. A30V650
     A20V535 NC 0. 0. A30V750
     A20V535 LT 0. 100. A30V650
     A20V535 HT 0. 175 A30V500
     A20V410 NC 0. 0. A30V600
     A20V410 LT 0. 75. A30V500
     A20V410 HT 0. 150. A30V400
     A30V850 CF 0. 850. A10V260
     A30V750 CF 0. 750. A10V260
     A30V650 CF 0. 650. A10V260
     A30V600 CF 0. 600. A10V260
     A30V500 CF 0. 500. A10V260
     A30V400 CF 0. 400. A10V260
?LOCAL SAOHILL
?INITIAL SAOHILL
   A10V260 = 1000.
?CONSTRAINTS SAOHILL
?GLOBAL
  B[:] = ZBEN
?OBJECTIVE
  MAXIMIZE ZBEN
?ENDATA
```

Figure 7.2: DLPFI Input for Simple Example

```
?PROBLEM    DYKSTRA
?NAME  SAOHILL
?ALTERNATIVES       1
?CLUSTER   1.00000E+00
?INITIAL  A10V260      1.00000E+03     1.00000E+03
 NC           0.00000E+00     0.00000E+00    A20V650
 LT           0.00000E+00     1.50000E+02    A30V750
 CF           0.00000E+00     7.50000E+02    A10V260
?LOCAL CONSTRAINTS       0
?GLOBAL CONSTRAINTS       1
 -9.00000E+05    ZBEN         0.00000E+00
?OBJECTIVE       1
  9.00000E+05
?ENDATA
```

Figure 7.3: Output for Simple Example

returned as shown. Again, much more detail on the output format can be found in Nazareth [2001b].

Figure 7.4 gives the DLPFI input for the extension of the forest planning problem described at the end of the previous section; specifically, the problem with six planning intervals and environmental constraints from the end of interval 3 onwards. It is again largely self-explanatory. The STATES and ACTIONS are unchanged. The NETWORK section gives the additional transformations that permit clear felling in any interval, and the LOCAL section now specifies the additional environmental constraints at the end of intervals 3 through 6. Figure 7.5 gives the corresponding optimal solution in the output format described above. The optimal solution now contains 5 planning alternatives. These are prescribed for a partition of the 1000 hectares tract into 100, 100, 300, 200, and 300 hectares respectively. The optimal benefit is now 1689000 cubic metres for the forest tract as compared to 1800000 (from two optimal rotations since the planning period has been doubled) for the unrestricted problem. A penalty has been incurred in terms of the overall benefit (physical yield) in order to meet important environmental constraints.

Exercise: Using the optimal solution in Figure 7.4, derive pie charts that describe the state of the woodland at the end of each planning interval.

A variety of other interesting scenarios can be explored by making additional simple changes to the DLPFI input.

Further information on the availability of the D_LP software is given in the notes below, and much more elaborate D_LP applications can be found in Nazareth [2001b].

7.3 Notes

Section 7.1: Memoryless BFGS was originally proposed by Shanno [1978], using a now-outmoded derivation.

Full detail on the numerical experiments with M-BFGS can be found in Nazareth [2006b]. The six quasi-Newton updates were as follows: DFP, BFGS, SR1, Davidon's Optimally Conditioned (OC), Hoshino's, and Xie-Yuan's. For their definitions and a detailed discussion of their properties, see Zhu [1997].

The testbed consisting of four problems from the Minpack collection—see More et al. [1981]—are as follows: extended Rosenbrock, Brown-Dennis, Watson, and Biggs6. Their problem dimensions are $n = 10, 4, 9, 6$, respectively. Each was run from its standard starting point. The line search employed is described in Nazareth [2003, Chapter 5].

Section 7.2: The three-interval forestry example is also given in Nazareth [2001, Chapter 6], which is, in turn, quoted from Dykstra [1984]. For another simple example involving rangeland planning, see Nazareth [2004a]. Additional detail on running the DLPDEMO program can be found in Nazareth [2004a, pgs. 91-92]. For other more challenging D_LP applications, see Nazareth [2001b].

The executable DLPDEMO program is derived from Fortran source code for the D_LP decision support system (called DLPEDU). The latter can be provided under a licensing agreement by contacting the author at the following address: Computational Decision Support Systems (CDSS), P.O. Box 10509, Bainbridge Island, WA 98110, USA. E-mail: larry_nazareth@q.com or nazareth@amath.washington.edu.

```
?PROBLEM    DYKSTRA
?PERIODS
   6
?CLASSES
  SAOHILL
?NAME  SAOHILL
?STATES SAOHILL
  A10V260 A20V650 A20V535 A20V410 A30V850 A30V750 A30V650
&  A30V600 A30V500 A30V400
?ACTIONS SAOHILL
  NC LT HT CF
?NETWORK SAOHILL
     A10V260 NC 0. 0. A20V650
     A10V260 LT 0. 50. A20V535
     A10V260 HT 0. 100. A20V410
     A10V260 CF 0. 260. A10V260
     A20V650 NC 0. 0. A30V850
     A20V650 LT 0. 150. A30V750
     A20V650 HT 0. 200. A30V650
     A20V650 CF 0. 650. A10V260
     A20V535 NC 0. 0. A30V750
     A20V535 LT 0. 100. A30V650
     A20V535 HT 0. 175. A30V500
     A20V535 CF 0. 535. A10V260
     A20V410 NC 0. 0. A30V600
     A20V410 LT 0. 75. A30V500
     A20V410 HT 0. 150. A30V400
     A20V410 CF 0. 410. A10V260
     A30V850 CF 0. 850. A10V260
     A30V750 CF 0. 750. A10V260
     A30V650 CF 0. 650. A10V260
     A30V600 CF 0. 600. A10V260
     A30V500 CF 0. 500. A10V260
     A30V400 CF 0. 400. A10V260
?LOCAL SAOHILL
?INITIAL SAOHILL
   A10V260 = 1000.
?CONSTRAINTS SAOHILL
 3: A30V850 + A30V750 + A30V650 >= 300.
 4: A30V850 + A30V750 + A30V650 >= 300.
 5: A30V850 + A30V750 + A30V650 >= 300.
 6: A30V850 + A30V750 + A30V650 >= 300.
?GLOBAL
  B[:] = ZBEN
?OBJECTIVE
  MAXIMIZE ZBEN
?ENDATA
```

Figure 7.4: Input for Problem with Environmental Constraints

```
?PROBLEM    DYKSTRA
?NAME  SAOHILL
?ALTERNATIVES       5
?CLUSTER   1.00000E-01
?INITIAL  A10V260      1.00000E+03     1.00000E+02
 CF           0.00000E+00     2.60000E+02    A10V260
 NC           0.00000E+00     0.00000E+00    A20V650
 LT           0.00000E+00     1.50000E+02    A30V750
 CF           0.00000E+00     7.50000E+02    A10V260
 NC           0.00000E+00     0.00000E+00    A20V650
 CF           0.00000E+00     6.50000E+02    A10V260
?CLUSTER   1.00000E-01
?INITIAL  A10V260      1.00000E+03     1.00000E+02
 NC           0.00000E+00     0.00000E+00    A20V650
 CF           0.00000E+00     6.50000E+02    A10V260
 NC           0.00000E+00     0.00000E+00    A20V650
 CF           0.00000E+00     6.50000E+02    A10V260
 NC           0.00000E+00     0.00000E+00    A20V650
 HT           0.00000E+00     2.00000E+02    A30V650
?CLUSTER   3.00000E-01
?INITIAL  A10V260      1.00000E+03     3.00000E+02
 NC           0.00000E+00     0.00000E+00    A20V650
 CF           0.00000E+00     6.50000E+02    A10V260
 NC           0.00000E+00     0.00000E+00    A20V650
 LT           0.00000E+00     1.50000E+02    A30V750
 CF           0.00000E+00     7.50000E+02    A10V260
 CF           0.00000E+00     2.60000E+02    A10V260
?CLUSTER   2.00000E-01
?INITIAL  A10V260      1.00000E+03     2.00000E+02
 CF           0.00000E+00     2.60000E+02    A10V260
 NC           0.00000E+00     0.00000E+00    A20V650
 LT           0.00000E+00     1.50000E+02    A30V750
 CF           0.00000E+00     7.50000E+02    A10V260
 NC           0.00000E+00     0.00000E+00    A20V650
 HT           0.00000E+00     2.00000E+02    A30V650
?CLUSTER   3.00000E-01
?INITIAL  A10V260      1.00000E+03     3.00000E+02
 NC           0.00000E+00     0.00000E+00    A20V650
 CF           0.00000E+00     6.50000E+02    A10V260
 CF           0.00000E+00     2.60000E+02    A10V260
 NC           0.00000E+00     0.00000E+00    A20V650
 LT           0.00000E+00     1.50000E+02    A30V750
 CF           0.00000E+00     7.50000E+02    A10V260
?LOCAL CONSTRAINTS       4
  0.00000E+00    G     3.00000E+02
  0.00000E+00    G     3.00000E+02
  0.00000E+00    G     3.00000E+02
  0.00000E+00    G     3.00000E+02
?GLOBAL CONSTRAINTS       1
 -1.68900E+06    ZBEN          0.00000E+00
?OBJECTIVE       1
  1.68900E+06
?ENDATA
```

Figure 7.5: Output for Problem with Environmental Constraints

.

BIBLIOGRAPHY

Adams, L. and Nazareth, J.L. (eds.) (1996), *Linear and Nonlinear Conjugate Gradient-Related Methods*, SIAM, Philadelphia.

Aho, A.V., Hopcroft, J.E, and Ullman, J.D. (1974), *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts.

Behnke, H. et al. (eds.) (1974), *Fundamentals of Mathematics. Volume I: Foundations of Mathematics/The Real Number System and Algebra. Volume II: Geometry. Volume III: Analysis.* The MIT Press, Cambridge, Massachusetts (translated by S.H. Gould).

Berlinski, D. (2000), *The Advent of the Algorithm*, Harcourt, New York.

Bertsekas, D.P. (1999), *Nonlinear Programming*, Athena Scientific, Belmont, Massachusetts (second edition).

Blum, L., Cucker, F., Shub, M., and Smale, S. (1998), *Complexity and Real Computation*, Springer-Verlag, New York (with a foreword by R.M. Karp).

Borwein, J. and Bailey, D. (2004), *Mathematics by Experiment*, A K Peters, Nantick, Massachusetts.

Braverman, M. and Cook, S. (2006), "Computing over the reals: foundations for scientific computing," *Notices of the AMS*, 53, No. 3, 318-329.

Buckley, A. (1978a), "Extending the relationship between the conjugate gradient and BFGS algorithms," *Mathematical Programming*, 15, 343-348.

Buckley, A. (1978b), "A combined conjugate-gradient quasi-Newton minimization algorithm," *Mathematical Programming*, 15, 200-210.

Cook, M. (2004), "Universality of elementary cellular automata," *Complex Systems*, 15, p. 1.

Dantzig, G.B. (1963), *Linear Programming and Extensions*, Princeton University Press, Princeton, New Jersey.

Davidon, W.C. (1959), "Variable metric method for minimization," Argonne National Laboratory, Report ANL-5990 (Rev.), Argonne, Illinois. (reprinted, with a new preface, in *SIAM J. Optimization*, 1, 1-17, 1991).

Davis, M. (1958), *Computability & Unsolvability*, McGraw-Hill, New York.

Davis, M.D. and Weyuker, E.J. (1983), *Computability, Complexity, and Languages*, Academic Press, New York.

De Duve, C. (2002), *Life Evolving: Molecules, Mind and Meaning*, Oxford University Press, Oxford, England.

Dehaene, S. (1997), *The Number Sense*, Oxford University Press, Oxford, England.

Dennis, J.E. and Schnabel, R.B. (1983), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice-Hall, New Jersey.

Dreyfus, S.E. (1965), *Dynamic Programming and the Calculus of Variations*, Academic Press, New York.

Dykstra, D.P. (1984), *Mathematical Programming for Natural Resource Management*, McGraw-Hill, New York.

Edelman, G.M. (1992), *Bright Air, Brilliant Fire*, Basic Books, New York.

Evans, J.R. and Minieka, E. (1992), *Optimization Algorithms for Networks and Graphs*, Marcel Dekker, New York (Second Edition).

Feynman, R.P. (1996), *Feynman Lectures on Computation*, Addison-Wesley, Reading, Massachusetts (Hey, A.J.G. and Allen, R.W., eds.).

Fletcher, R. and Powell, M.J.D. (1963), "A rapidly convergent descent method for minimization," *Computer Journal*, 6, 163-168.

Garey, M.R. and Johnson, D.S. (1979), *Computers and Intractability*, W.H. Freeman and Company, San Francisco and New York.

Gowers, T., Barrow-Green, J., and Leader, I. (eds.) (2008), *The Princeton Companion to Mathematics*, Princeton University Press, Princeton, New Jersey.

Graham, R., Knuth, D.E., and Patashnik, O. (1989), *Concrete Mathematics: A Foundation for Computer Science*, Addison-Wesley, Reading, Massachusetts.

Gropp, W., Lusk, E., and Skjellum, A. (1994), *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, The MIT Press, Cambridge, Massachusetts.

Hawking, S. (ed.) (2005), *God Created the Integers*, Running Press, Philadelphia, Pennsylvania.

Hopcroft, J.E. and Kennedy, J.W. Chairs (1989), *Computer Science: Achievements and Opportunities*, Report of the NSF Advisory Committee for Computer Research, SIAM, Philadelphia.

Kao, M-Y. (ed.), (2008), *Encyclopedia of Algorithms*, Springer, New York.

Keyes, D. (1999), "Krylov, Lagrange, Newton, and Schwarz: Combinations and Permutations," Plenary Talk, SIAM Annual Meeting, Atlanta, Georgia.

Knuth, D.E. (1968), *The Art of Computer Programming: Fundamental Algorithms*, Addison-Wesley, Reading, Massachusetts.

Knuth, D.E. (1996), *Selected Papers on Computer Science*, CSLI Lecture Notes No. 59, Cambridge University Press, Cambridge, England.

Knuth, D.E. (2005), *MMIX: A RISC Computer for the New Millennium*, Volume 1, Fascicle 1, Addison-Wesley (Pearson Education), Upper Saddle River, New Jersey.

Kolda, T.G., O'Leary, D.P. and Nazareth, J.L. (1998), "BFGS with update skipping and varying memory," *SIAM J. Optimization*, 8, 1060-1083.

Kozen, D.C. (1992), *The Design and Analysis of Algorithms*, Springer-Verlag, New York.

Lagarias, J.C., Reeds, J.A., Wright, M.H., and Wright, P.E. (1998), "Convergence properties of the Nelder-Mead simplex algorithm in low dimensions," *SIAM J. Optimization*, 9, 112-147.

Lawler, E. (1976), *Combinatorial Optimization*, Dover Publications, Mineola, New York (reprint, 2001).

Lesk, A.M. (2002), *Introduction to Bioinformatics*, Oxford University Press, Oxford and New York.

Lewin, R. (1992), *Complexity*, Macmillan Publishing Company, New York.

Liu, D.C. and Nocedal, J. (1989), "On the limited memory BFGS method for large-scale optimization," *Mathematical Programming*, 45, 503-528.

Luenberger, D.G. (1984), *Linear and Nonlinear Programming*, Addison-Wesley (second edition).

Mayr, E. (1982), *The Growth of Biological Thought*, Harvard University Press, Cambridge, Massachusetts.

Mayr, E. (2001), *What Evolution Is*, Basic Books, New York.

Mertens, S. and Moore, C. (2010), *The Nature of Computation*, Oxford University Press (to appear).

Moore, C. (1996), "Recursion theory on the reals and continuous-time computation," *Theoretical Computer Science*, 162, 23-44.

Moré, J.J., Garbow, G.S., and Hillstrom, K.E. (1981), "Testing unconstrained optimization software," *ACM Transactions on Mathematical Software*, 7, 17-41.

Nazareth, J.L. (1976), "A relationship between the BFGS and conjugate gradient algorithms," Tech. Memo. ANL-AMD 282, Applied Mathematics Division, Argonne National Laboratory, Argonne, Illinois.

Nazareth, J.L. (1979), "A relationship between the BFGS and conjugate gradient algorithms and its implications for new algorithms," *SIAM J. Numerical Analysis*, 16, 794-800.

Nazareth, J.L. (1987), *Computer Solution of Linear Programs*, Oxford University Press, Oxford and New York.

Nazareth, J.L. (2001a), "Multialgorithms for parallel computing: a new paradigm for optimization," in *Stochastic Optimization: Algorithms and Applications*, S. Uryasev and P.M. Pardalos (eds.), Kluwer Academic Publishers, Dordrecht and Boston, 183-222.

Nazareth, J.L. (2001b), *D_LP and Extensions*, Springer, Heidelberg and Berlin.

Nazareth, J.L. (2003), *Differentiable Optimization and Equation Solving: A Treatise on Algorithmic Science and the Karmarkar Revolution*, Springer, New York.

Nazareth, J.L. (2004a), *An Optimization Primer*, Springer, New York.

Nazareth, J.L. (2004b), "On conjugate gradient algorithms as objects of scientific study," *Optimization Methods and Software*, 18, 555-565; 19, 437-438 (appendix).

Nazareth, J.L. (2006a), "On algorithmic science and engineering: an emerging core discipline of computing," *SIAM News*, 39, No. 2, 2-4.

Nazareth, J.L. (2006b), "Complementary perspectives on optimization algorithms," *Pacific Journal of Optimization*, 2, 71-79.

Nazareth, J.L. (2008), "Symbol-based via-à-vis magnitude-based computing: on the foundations and organization of algorithmic science & engineering," Presented at the Santa Fe Insitute, New Mexico, USA (Seminar: April 29, 2008).

Nazareth, J.L., and Tseng, P. (2002), "Gilding the lily: a variant of the Nelder-Mead algorithm based on golden-section search," *Computational Optimization and Applications*, 22, 123-144.

Nelder, J.A. and Mead, R. (1965), "A simplex method for function minimization," *Computer Journal*, 7, 308-313.

Newell, A., Perlis, A.J., and Simon, H.A. (1967), "Computer science," *Science*, 157, 1373-1374.

Nielsen, M.A. and Chuang, I.L. (2000), *Quantum Computation and Quantum Information*, Cambridge University Press, Cambridge, England.

Nocedal, J. (1980), "Updating quasi-Newton matrices with limited storage," *Mathematics of Computation*, 35, 773-782.

Pacific-West Algorithmic Science Meeting, April 8, 2000, Pullman, Washington (co-organized by J.L. Nazareth and K. Cooper).

Parlett, B.N. (1980), *The Symmetric Eigenvalue Problem*, Prentice-Hall, Englewood Cliffs, New Jersey.

Penrose, R. (1989), *The Emperor's New Mind*, Oxford University Press, Oxford and New York.

Powell, M.J.D. (1977), "Restart procedures for the conjugate gradient method," *Mathematical Programming*, 12, 241-254.

Press, W.H., Teukolsky, S.A., Vetterling, W.T., and Flannery, B.P. (1992), *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, Second Edition, Cambridge University Press, Cambridge, England.

Renegar, J., Shub, M., and Smale, S. (eds.), (1996), *Mathematics of Numerical Analysis*, American Mathematical Society, Providence, Rhode Island.

Shanno, D.F. (1978), "Conjugate gradient methods with inexact searches," *Mathematics of Operations Research*, 3, 244-256.

Shannon, C. (1941), "Mathematical theory of the differential analyzer," *Journal Math. Phys. MIT*, 20, 337-354.

Shub, M. (2001), Editorial, *Foundations of Computational Mathematics*, 1, No. 1, 1-2.

Stewart, I. (1987), *The Problems of Mathematics*, Oxford University Press, Oxford and New York.

Strang, G. (1986), *Introduction to Applied Mathematics*, Wellesley-Cambridge Press, Cambridge, Massachusetts.

Taub, A.H. (ed.) (1961), *Collected Works*, Macmillan, London, vol. V, 288-328, [von Neumann, J., The general and logical theory of automata].

Traub, J.F. (1999), "A continuous model of computation," *Physics Today*, May 1999, 39-43.

Traub, J.F. and Werschulz, A.G. (1998), *Complexity and Information*, Cambridge University Press, Cambridge, England.

Trefethen, L.N. (1992), "The definition of numerical analysis," *SIAM News*, 25, 6.

Trefethen, L.N. (2005), "Ten-digit algorithms," A.R. Mitchell Lecture, Biennial Conference on Numerical Analysis, Dundee, Scotland.

Tung, K.K. (2007), *Topics in Mathematical Modeling*, Princeton University Press, Princeton, New Jersey.

Wilkinson, J.H. (1963), *Rounding Errors in Algebraic Processes*, Prentice-Hall, Englewood Cliffs, New Jersey.

Wilkinson, J.H. (1965), *The Algebraic Eigenvalue Problem*, Clarendon Press, Oxford, England.

Wolfram, S. (2002), *A New Kind of Science*, Wolfram Media Inc., Champaign, Illinois.

Zhu, M. (1997), *Techniques for Nonlinear Optimization: Principles and Practice*, Ph.D. Dissertation, Department of Pure and Applied Mathematics, Washington State University, Pullman, Washington.