Patterns and Operators

The Foundations of Data Representation

J. C. Simon

Patterns and Operators

The Foundations of Data Representation

by

J. C. Simon

Université Pierre et Marie Curie

Translated by

J. Howlett

NORTH OXFORD ACADEMIC
A division of Kogan Page

English edition first published 1986
by North Oxford Academic Publishers Ltd,
a subsidiary of Kogan Page Ltd, 120 Pentonville Road,
London N1 9JN

British Library Cataloguing in Publication Data
Simon, J. C.
Patterns and Operators
1. Pattern perception 2. Algorithms
I. Title II. La reconnaissance des formes par
algorithmes. *English*
006.4 Q327

ISBN 0-946536-16-3

Typeset by Keyspools Ltd, Golborne, Lancs.
Printed and bound in Great Britain by
Anchor Brendon, Ltd, Tiptree, Essex.

Contents

Foreword 1

1. Some basic concepts of algorithmic processing 5

1.1. Definitions 5
- 1.1.1. Primitive recursive and partial recursive functions . . 6
- 1.1.2. Random access machines 7
- 1.1.3. Turing machines 7
- 1.1.4. Markov's algorithms 7
- 1.1.5. Theorem: identity of classes of functions 7
- 1.1.6. Comments. 8

1.2. Computational complexity: hierarchical order 8
- 1.2.1. Coding the data 9
- 1.2.2. Complexity and usefulness of an algorithm 9
- 1.2.3. Undecidable problems 11
- 1.2.4. Problems of intrinsically exponential or higher order complexity. 11
- 1.2.5. NP problems. 12
- 1.2.6. Polynomial problems 12

1.3. NP–complete problems 12
- 1.3.1. Example of an NP problem 12
- 1.3.2. Some definitions 14
- 1.3.3. Problems in graph theory 14
- 1.3.4. Problems concerning sets and partitions 15
- 1.3.5. Problems concerning compression and representation 16
- 1.3.6. Problems concerning syntactical relations 16
- 1.3.7. Comments. 17

1.4. Polynomial problems 17
- 1.4.1. Problems in graph theory 18
- 1.4.2. Sorting and memory searching. 18
- 1.4.3. Signal and image processing. 18

1.5. Overcoming exponential intractability 19
- 1.5.1. Relaxing the conditions of a problem 20
- 1.5.2. Making use of the semantics of the problem. 21

2. Representation and interpretation 23

2.1. Signs: signifiers and signifieds. 23

2.2. Representation and interpretation of information 24

2.2.1. Definitions 24
2.2.2. The aims of pattern recognition 26
2.2.3. Some further definitions 27
2.2.4. The methods of pattern recognition 28

2.3. Properties of the representation space 30
2.3.1. Relations between the representations 30
2.3.2. Object–concept distance; characteristic functions . . 31
2.3.3. Finite set of representations 33
2.3.4. Ordinal variables 34
2.3.5. Reduction of the dimension 35
2.3.6. Euclidean vector space 35
2.3.7. Invariant representations 39
2.3.8. Influence of the interpretation on the properties of the representation 40

2.4. Properties of interpretations and of recognition operators . 42
2.4.1. Operators as signifiers 42
2.4.2. Linear operators 43
2.4.3. Logical, syntactic and linguistic operators 45
2.4.4. Combinations of operators 48
2.4.5. Semantics and pragmatics 54

3. Representing and processing the signal 61

3.1. Representation by continuous functions 61
3.1.1. Linear filters 62
3.1.2. Fourier transform 63
3.1.3. Signals with bounded domain spectrum 65
3.1.4. Fractals 69

3.2. Periodic signals and Fourier series 71

3.3. Fourier transforms in information science 74
3.3.1. Discrete Fourier transform 75
3.3.2. Fast Fourier transform 76

3.4. Sequences of samples: z-transforms 78
3.4.1. Functions of a complex variable: recapitulation . . . 78
3.4.2. z-transform for linear filters 79
3.4.3. Linear prediction 80

3.5. Compression of information 81
3.5.1. Entropy and volume of information 81
3.5.2. Use of unitary transforms 86
3.5.3. Correlation matrix for an image 91
3.5.4. Compression of multi-spectral images using the Karhunen–Loève algorithm 93

3.5.5. Use of unitary transforms other than the Karhunen–Loève transform 94

3.6. Precision and quantization 95

3.6.1. Vignes' permutation–perturbation method 95

3.6.2. Practical application: software for Vignes' method . . 96

4. Finite representation set 97

4.1. Partitions of a finite set 97

4.1.1. Enumeration of partitions 98

4.2. Hierarchies 101

4.3. Ultrametrics and clustering 101

4.3.1. Ultrametric distance 102

4.3.2. Ultrametric space 103

4.3.3. Obtaining an ultrametric from a metric 104

4.3.4. Distances and linkage effect 106

4.3.5. Comments on the clustering methods 107

4.4. Finite sets and the graph of distances 108

4.4.1. Minimum length tree 109

4.4.2. Algorithms for graphs of distances 111

4.5. Data structures 112

4.5.1. Relational data structures 112

4.5.2. Comparison of descriptions 113

4.5.3. Classification of descriptions 117

4.5.4. Inference and expert systems 120

4.6. Pattern matching 125

4.6.1. String matching 126

4.6.2. Searching for repeated patterns 127

4.7. Distance between elements 129

4.7.1. Quantitative variables 130

4.7.2. Qualitative variables 132

4.7.3. Distance between character strings 134

4.7.4. Dynamic programming 135

4.8. Measures of similarity between variables 140

4.9. Distance measures for objects and concepts 141

4.9.1. Comparison of object–concept distances 143

4.10. Adaptive partitions 145

4.10.1. Basic principles, and some history 146

4.10.2. The dynamic cluster algorithm 147

4.10.3. Some properties of $\mathbb{R}^n$ 154

4.10.4. Initial tests for inhomogeneity 155

5. Ordinal representation space 157

5.1. Peano–Hilbert scan in an n-ordinal space 158
5.1.1. Definition of the Peano–Hilbert scan 159
5.1.2. Properties of the Peano–Hilbert scan 161
5.1.3. Application of the Peano–Hilbert scan 162

5.2. Intrinsic dimension of a data set 166

5.3. Neighbourhood graph in an ordinal space 169
5.3.1. Properties of the neighbourhood graph 169

5.4. n-ordinal data structures. 174
5.4.1. Oriented binary search tree 175

6. Euclidean representation space 177

6.1. Linear separation 177
6.1.1. The perceptron algorithm. 178
6.1.2. The Ho Kashyap algorithm 179
6.1.3. Partitioning by hyperplanes into k classes 180
6.1.4. The birth and death of a myth 181

6.2. Statistical pattern recognition. 181
6.2.1. Bayesian decision theory 182
6.2.2. Parametric Bayesian learning 185
6.2.3. Non-parametric learning 187
6.2.4. The q nearest neighbours 189

Bibliography 193

Index 209

Foreword

In the form in which it is presented in this book, pattern recognition was born at the end of the 1960s at the same time as the powerful computers now described as third generation machines. The arrival of these machines made it possible to experiment with the information (the data) provided by instruments in many fields of observation: in visual images, in spoken words, in the fields of physics, medicine, economics, linguistics, etc. A new field of observation and study was transferred from philosophy to experimental science. This is a general and crucial phenomenon in the history of Man's efforts to understand Nature: the telescope gave birth to astronomy, and metallurgical, chemical, electrical and vacuum techniques gave birth to physics. There is some paradox in the fact that the computer, designed for commercial accounting and scientific computation, gave birth to pattern recognition and artificial intelligence.

The scope of pattern recognition is very wide indeed. About a thousand scientific publications appear each year and the burgeoning subject seems to be devoid of all order. There is much repetition, often concealed by different modes of expression, showing the variety of cultural origins of the specialists. A newcomer to the field is thus disoriented by the number of papers, approaches and special terms. Textbooks are necessary to present the field from a unified point of view. This book is an effort along that tutorial line. Our aim is to present pattern recognition topics in a logical order, around some ideas which, to us, seem the best threads to follow in order to grasp the unity of the field of algorithmic pattern recognition:

- computational complexity;
- the properties of representation spaces;
- the properties of interpretation spaces.

For the pure mathematician, pattern recognition is a trivial problem which can be expressed formally as follows. Let $\mathbb{X}$ be a representation space, preferably a 'nice' topological space, and let Ω, the interpretation space, be a finite set of names. A recognition or identification is a mapping $\mathscr{E}:\mathbb{X} \to \Omega$ to which certain properties are ascribed and from which elegant theorems can be deduced.

This, however, is not where the problem lies: in practice, the question is one of *constructing* $\mathscr{E}$, i.e. of providing operators or programs which, given any $X \in \mathbb{X}$, enable us to decide automatically onto which $\omega \in \Omega$ the element X is mapped.

An extended definition of $\mathscr{E}$, to be held in full in memory, is out of the question, even for small-scale problems, because here we come up against the problem of *computational complexity*. In the search for usable operators, pattern recognition is continually confronted with problems of information complexity; so many pattern recognition problems are exponential that we are constantly obliged to adopt less than optimal, polynomial solutions. Pattern recognition is first and foremost a battle against complexity. For this reason, the first chapter of this book recapitulates the necessary results of complexity theory: they are the 'safety barriers' essential to anyone practising the subject.

The other guiding thread seems to us to be the *semantics* of the general pattern recognition problem, which varies according to the question under consideration. Is there in fact a general, universally applicable method for constructing a pattern recognition operator? It was thought in the 1950s, when self-organizing or automatic learning systems based on perceptrons were flourishing, that there might be such a general method. However, just as there is no computer program which can decide whether or not any given program will halt within a finite number of steps, there is no program which can construct an operator able to solve any given pattern recognition problem.

We must therefore treat each problem in a specific manner and look for any items of information that will enable us to construct the required operators. Our view is that information is to be found in the properties of

- the representation space,
- the interpretation space or spaces.

The introductory survey is followed by a chapter on representation and interpretation, dealing with general methods of presentation, after which we turn to signal processing—a subject which needs to be understood by anyone working in pattern recognition. The final chapters deal with the various possible representation spaces: finite, *n*-ordinal and Euclidean. In this book, we have therefore limited ourselves to the treatment of representation spaces, which may be regarded as the most classical part of the subject; the use of the properties of the interpretation space will be dealt with in a second volume.

In order to counteract the effects of complexity, a pattern recognition operation is broken down into a number of successive steps. In this volume, we shall consider the techniques used at the lowest levels in these processes, those closest to the physical sensors. Those used at the higher levels are similar to the methods of artificial intelligence, although in artificial intelligence there is a tendency to deal with 'toy' problems whereas the pattern recognition expert is confronted with 'real' problems—the recognition of speech, images and written characters—and has to attack them as a physicist or engineer rather than as a theoretician. Our 'pure' colleagues may find our field of pattern recognition research somewhat confused and perhaps not very glamorous. We recommend that they tackle some of these real problems: they will find it a rather frustrating experience because, with our so-called powerful machines, we are far from doing as well as living creatures with their senses of sight and hearing. However, they will also find it a rich source of theoretical problems, as physics was for the mathematicians of the 19th century, for men such as Poincaré.

Thus, in this first volume we aim to give a structured account of a broad field which is in a state of rapid expansion; we also attempt to classify the various categorization systems which have been proposed and to bring together algorithmic techniques which differ, not in the syntax of the algorithms themselves, but in the discussion their authors provide to accompany them. In short, it should be regarded as a textbook for postgraduate and research level studies.

It is the reward of a university professor and leader of a research team to be surrounded by talented young people and to witness the development of their skills. It is for them that I have taken the time to write this book, despite the many tasks calling for my attention; here, they will find echoes of their own excellent work, and it is to them that I dedicate the book.

I wish to offer my sincere thanks to Dr J. Howlett and D. Beeson for a careful translation and review of this work.

*J. C. **Simon***
Bonas, 1984

Chapter 1

Some basic concepts of algorithmic processing

The use of computers, and therefore of programs, for the recognition of patterns requires a good knowledge of the fundamental concepts of information science. It seems useful, therefore, before embarking on the subject proper, to give a short survey of the ideas that have emerged from basic research. Whilst many of these concepts, e.g. those of procedure, algorithm and computation, go back to antiquity, recent research in information science has made it possible both to extend them and to make them more precise.

1.1. Definitions

An information processing machine (digital or numerical) or computer is a physical system comprising

(1) a set of registers or memory cells in which coded representations of information can be stored,
(2) a set of processors or automata which can perform operations on this information and
(3) input and output devices for communication with the outside world.

A register or memory cell consists of a series of binary flipflops; each able to exist in either of two states which are conventionally represented by 0 and 1. A set of n such elements forms a register of length n, or an n-register, for which there are 2^n possible states. As long as it has been properly programmed, a processor can perform any operation which changes the state of a register.

Loosely, we can define

- a **procedure** as a list of tasks to be performed in the order given, e.g. the checklist to be gone through before an aeroplane is allowed to take off;
- an **algorithm** as a list of operational rules for performing a computation; it is thus a computational procedure, e.g. the algorithm for multiplication.

In information science, no distinction is made between algorithms and procedures, and in fact all operations are performed on the machine registers. As we shall see, a state *represented* by a series of 0s and 1s can be *interpreted* in many different ways. If the interpretation (the *type* of the representation) has the properties of a number (integer, real or other), then we are dealing with numerical computation, but it is more often the case that the interpretation is in terms of the elements, i.e. symbols, of a set. The proportion of computer applications dealing with numerical computation is now in continual decline, in spite of the fact that these machines were originally designed as 'number crunchers'. The processing of strings of characters, or 'symbols' in the sense that we shall give the term in the next section, has been their main activity for quite a while. The discussion which follows will make it easier to understand why the computer is a 'universal' machine for symbol manipulation.

Several definitions of algorithms or finite procedures are based on methods which differ greatly but which all make use of closure operations, formal systems for symbol manipulation, and computing devices; such methods have been developed by Church, Gödel, Herbrand, Kleene, Markov, Post, Turing and others. All the definitions proposed turn out to be equivalent, in the sense that they all relate to the same class of functions: partial recursive functions. The Church–Turing theorem states that these functions form the class of *computable functions*; a rigorous and detailed account is given by Matchey and Young (1978) for example.

1.1.1. Primitive recursive and partial recursive functions

The following is an algebraic definition of a computable function, by closure operation on a set of words[†].

Let $A = (a_1, a_2, \ldots, a_k)$ be a k-alphabet (i.e. an alphabet of k letters) and A^* the infinite set of words x, including the empty word ε, that can be formed with the letters of A. With domain $(A^*)^n$ and values in A^*, we have the following elementary functions:

zero	$Z(x) = \varepsilon$
jth successor	$S_j(x) = xa_j$
projection	$P^n{}_j$

A primitive recursive function is constructed from these elementary functions by using the composition and primitive recursion operations.

However, it can be shown, by the use of Cantor's diagonal argument, that certain computable functions are not primitive recursive; in practice, this means that not all algorithms will lead to a result for all values of their arguments—they can loop. Partial recursive functions are so called because they are not defined for all arguments in their domain and are therefore, in a sense, indeterminate. Such functions are defined by recursive minimization instead of by the primitive recursive operation, but otherwise they are the same as primitive recursive functions. Every primitive recursive function is thus a partial recursive function defined for all arguments in its domain.

[†] The terms element, symbol, character and word will be used more or less interchangeably for members of a set which we call an alphabet in the case of characters or letters and a vocabulary in the case of words.

1.1.2. Random access machines

A random access machine is a computing device with a potentially infinite memory, meaning that the number of memory cells is as large as required. Any word in A^* can be written into the memory and the machine can execute seven instructions as follows, where Y and Z are addresses in the memory:

$app_j\ Y$	append a_j to the right of the word in cell Y
$del\ Y$	delete the first letter of the word in Y
$nul\ Y$	replace the word in Y by the null (empty) word ε
$Y \leftarrow Z$	copy into Y the word in Z, leaving Z unchanged
$skip\ X$	skip the instruction labelled X
$Y\ skip_j\ X$	conditional skip, executed if the first letter of Y is a_j
stop	do nothing

A program for such a machine is a finite sequence of instructions; it stops if it reaches a 'stop' instruction.

1.1.3. Turing machines

A Turing machine is an automaton with a potentially infinite linear tape which can be moved any number of steps in either direction and has a read and/or write head for tape reading or writing. The characters written on the tape are the letters of a finite alphabet A, together with the 'space' character. The machine has p internal states allowing it to control read and write operations, e.g. 'if when in state i character a_j is read on the tape, replace this by a_n, move the tape one space in one direction (say, to the right) and enter state m'.

1.1.4. Markov's algorithms

Markov's algorithms are procedures for the formal manipulation of symbols, expressing the rules for writing formal grammars. A Markov algorithm is a finite ordered sequence of production rules, i.e. of rules for deriving a word q from a word p, both belonging to A^*. Certain production rules are *terminal*, meaning that nothing further can be derived from them. A partial recursive function is computable in the Markov sense if it can be computed by a Markov algorithm.

1.1.5. Theorem: identity of classes of functions

The theorem states that *the following classes of functions are identical:*

- *partial recursive functions*
- *functions computable by a random access machine*
- *functions computable by a Turing machine*
- *functions computable by a Markov algorithm*

These are, furthermore, actual equivalences in the sense that, if a function is defined *by a program for one of these formalisms, the program can be translated into any of the others and the same partial recursive function will be obtained.*

1.1.6. Comments

(a) The concepts of recursive functions, Turing machines, Markov processes and automata were developed *before* the arrival of the electronic computer but *after* the mechanical calculators designed by Pascal, Babbage or Lady Lovelace (the beautiful Ada).

(b) All functions used in practice in pattern recognition are primitive recursive. When restricted to this class, the above theorem is fairly simple to prove.

(c) If A^* is the infinite set of words formed from the finite alphabet A, every function $(A^*)^n \to A^*$ is a recursive function and can be implemented by a random access machine or a Turing machine. Conversely, every program for such a machine defines a recursive function. *No machine more powerful than a random access or Turing machine can exist.* Clearly, less powerful machines can be conceived, e.g. a stack machine or a finite state automaton. A modern computer is equivalent to a random access machine or a Turing machine, neither more nor less; it is clearly much easier to use, as a consequence of the development of high level programming languages.

(d) Computers were developed essentially for numerical computation and commercial data processing, but their scope is much more general: to implement *any* recursive function. It is interesting to consider at what stage the scientific community understood this important fact. According to Hamming (1976), the perception of computers as symbol manipulators rather than number crunchers came fairly late, and, as ontogeny flows from phylogeny, the same steps towards this understanding will be taken by every individual learning information science.

It is not clear whether Turing himself, after proving that a Turing machine can do anything that a computer can do, really understood that a computer is a manipulator of symbols (i.e. representations), and, in any case, the famous paper by Burks, Goldstein and von Neumann (1947) shows that they saw the computer only as a number cruncher. According to Hamming, it was not until 1952–1954 that a majority began to see that computers were more than this. It has taken a long time for the message to spread and one can say with some confidence that it is not yet understood by the general public, who still see the computer as a kind of super pocket calculator.

1.2. Computational complexity: hierarchical order

A finite number of elementary operations is necessary for the execution of an algorithm or finite procedure. Let f denote an algorithm which, given data x, produces result y. Put very generally, the number of operations which have to be performed in this process depends on the size n of data x (a concept which we shall study in Section 1.2.1). If n is large, the dominant term $O(n)$ is an estimate of the complexity of the algorithm. For example, if the number of elementary operations is a polynomial of degree r in n, then the order is $O(n^r)$.

1.2.1. Coding the data

We must now give a precise meaning to the term 'size n of data x'.

In pattern recognition, data are generally given simply as a string of binary digits (bits) 0 and 1, or of numerical values to the same base, but this is not necessarily the case for every problem that can be solved by means of an algorithm; e.g. for problems concerning graphs there are many techniques for coding the data provided by the graph, such as an array, a list of nodes together with their neighbours etc. However, it is always possible to transform one representation into any other by means of a *polynomial* algorithm.

We shall assume that the coding scheme is 'reasonable' and takes the form of a list of elements formed from a k-alphabet. These elements, or symbols, cannot necessarily be combined with each other as can numerical values, for example. Alphabet A can be coded in bits, each symbol requiring $\lfloor \log_2 k \rfloor$ for its representation, and therefore any item of input data can always be reduced to a string of bits. Furthermore, this is the only form that is accepted by a computer at the level of its processing registers or memory cells; the decoding or interpretation of these bit strings is not always possible from the strings alone because other important information often has to be provided.

We shall also assume that the passage from one reasonable representation to another can be made by means of a polynomial transformation, in particular by a simple multiplication by a constant.

Note It is usual to omit any constant multiplier from an expression $O[f(n)]$, e.g. to write $O(n^2)$ rather than $O(3n^2)$.

1.2.2. Complexity and usefulness of an algorithm

Clearly, from the practical point of view the best algorithm for solving any given problem is the least complex, but this leads to the question of whether it is possible to prove that there is or is not a better algorithm than the one we happen to know. For example, can it be proved that there is *no* polynomial algorithm for the solution of some given problem or that the best possible algorithm is exponential?

We should note that the answer to this question can depend on the data: there are cases in which an algorithm can change its complexity class (e.g. from exponential to polynomial) for a small change in the nature of the data. We shall give an example of this later, with the evaluation of a logical expression. When considering complexity as a function of the data, we must distinguish between moderate complexity, the most favourable complexity and the least favourable complexity.

We consider now the variation in complexity with the nature of $O[f(n)]$, as illustrated by Fig. 1. This figure shows clearly why any problem of complexity other than polynomial is intractable. A modern computer can execute about 10^8 elementary operations per second, or about 10^{13} per day; even if we assume that this performance will be increased by a factor of 10^3 in the foreseeable future, we can see that a figure of 10^{15} operations per day will always be difficult to attain and that 10^{20} will never be reached. An exponential algorithm requires this number of operations for only a moderately large value of data size n. Since, as we shall see, a problem in speech recognition can easily involve data sizes of 10^4 bits (and visual image recognition 10^6 bits), exponential algorithms are completely out of the question here.

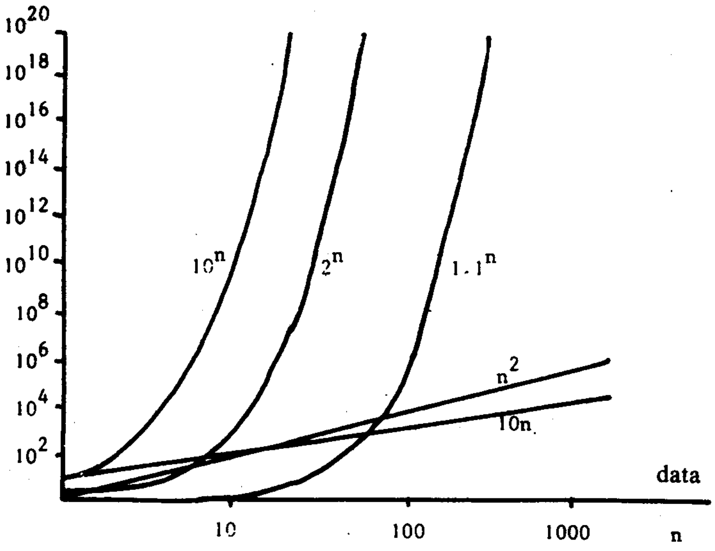

Fig. 1

More precisely, the computing power of large-scale modern machines is often quoted in megaflops (millions of floating-point arithmetical operations per second (Mflops) usually on 32-bit registers). For example, the CDC Cyber 205 is usually quoted as operating at 200–800 Mflops and the Cray-1 at 40–160 Mflops; estimates for the new Cray XMP are given as running up to 1000 Mflops. Doubtless, even more powerful machines will be produced in the future. The power of a central processor is sometimes quoted in millions of instructions per second (mips).

Table 1 gives the increase in the size of problem that can be handled for given increases in power, for a variety of complexity laws; here, N is the size that can be handled by the original machine. This shows clearly that, with an exponential algorithm, even a very great increase in power has no effect if the size N of the data exceeds a few tens. Thus, *it is better to look for a polynomial algorithm* (provided