COMPACT NUMERICAL METHODS FOR COMPUTERS:

linear algebra and function minimisation

J. C. NASH

COMPACT NUMERICAL METHODS FOR COMPUTERS:

linear algebra and function minimisation

J. C. NASH

Adam Hilger Ltd
Bristol

British Library Cataloguing in Publication Data

Nash, J C
Compact numerical methods for computers
1. Numerical analysis—Data processing
I. Title
519.4 QA297

ISBN 0-85274-330-0

Published by Adam Hilger Ltd, Techno House, Redcliffe Way, Bristol BS1 6NX.
Adam Hilger is now owned by The Institute of Physics.

Filmset by The Universities Press (Belfast) Ltd, and printed in Great Britain by The Pitman Press, Lower Bristol Road, Bath BA2 3BL.

PREFACE

This book is designed to help people solve numerical problems. In particular, it is directed to those who wish to solve numerical problems on 'small' computers, that is, machines which have limited storage in their main memory for program and data. This may be a programmable calculator—even a pocket model—or it may be a subsystem of a monster computer. The algorithms that are presented in the following pages have been used on machines such as a Hewlett-Packard 9825 programmable calculator and an IBM 370/168 with Floating Point Systems Array Processor. That is to say, they are designed to be used anywhere that a problem exists for them to attempt to solve. In some instances, the algorithms will not be as efficient as others available for the job because they have been chosen and developed to be 'small'. However, I believe users will find them surprisingly economical to employ because their size and/or simplicity reduces errors and human costs compared to equivalent 'larger' programs.

Can this book be used as a text to teach numerical methods? I believe it can. The subject areas covered are, principally, numerical linear algebra, function minimisation and root-finding. Interpolation, quadrature and differential equations are largely ignored as they have not formed a significant part of my own work experience. The instructor in numerical methods will find perhaps too few examples and no exercises. However, I feel the examples which are presented provide fertile ground for the development of many exercises. As much as possible, I have tried to present examples from the real world. Thus the origins of the mathematical problems are visible in order that readers may appreciate that these are not merely interesting diversions for those with time and computers available.

Errors in a book of this sort, especially in the algorithms, can depreciate its value severely. I would very much appreciate hearing from anyone who discovers faults and will do my best to respond to such queries by maintaining an errata sheet. In addition to the inevitable typographical errors, my own included, I anticipate that some practitioners will take exception to some of the choices I have made with respect to algorithms, convergence criteria and organisation of calculations. Out of such differences, I have usually managed to learn something of value in improving my subsequent work, either by accepting new ideas or by being reassured that what I was doing had been through some criticism and had survived.

There are a number of people who deserve thanks for their contribution to this book and who may not be mentioned explicitly in the text:
(i) in the United Kingdom, the many members of the Numerical Algorithms Group, of the Numerical Optimization Centre and of various university departments with whom I discussed the ideas from which the algorithms have condensed;

(ii) in the United States, the members of the Applied Mathematics Division of the Argonne National Laboratory who have taken such an interest in the algorithms, and Stephen Nash who has pointed out a number of errors and faults; and
(iii) in Canada, the members of the Economics Branch of Agriculture Canada for presenting me with such interesting problems to solve, Kevin Price for careful and detailed criticism, Bob Henderson for trying out most of the algorithms, Richard Wang for pointing out several errors in chapter 8, John Johns for trying (and finding errors in) eigenvalue algorithms, and not least Mary Nash for a host of corrections and improvements to the book as a whole.

It is a pleasure to acknowledge the very important roles of Neville Goodman and Geoff Amor of Adam Hilger Ltd in the realisation of this book.

J. C. Nash

Ottawa, 22 December 1977

CONTENTS

Chapter 1

A STARTING POINT

1.1. PURPOSE AND SCOPE

This monograph is written for the person who has to solve problems with (small) computers. It is a handbook to help him or her obtain reliable answers to specific questions, posed in a mathematical way, using limited computational resources. To this end the solution methods proposed are presented not only as formulae but also as algorithms, those recipes for solving problems which are more than merely a list of the mathematical ingredients.

There has been an attempt throughout to give examples of each type of calculation and in particular to give examples of cases which are prone to upset the execution of algorithms. No doubt there are many gaps in the treatment where the experience which is condensed into these pages has not been adequate to guard against all the pitfalls that confront the problem solver. The process of learning is continuous, as much for the teacher as the taught. Therefore, the user of this work is advised to think for himself and to use his own knowledge and familiarity of particular problems as much as possible. There is, after all, less than a human generation of experience with automatic computation and it should not seem surprising that satisfactory methods do not exist as yet for many problems. Throughout the sections which follow, this underlying novelty of the art of solving numerical problems by automatic algorithms finds expression in a conservative design policy. Reliability is given priority over speed and, from the title of the work, space requirements for both the programs and the data are kept low.

Despite this policy, it must be mentioned immediately and with some emphasis that the algorithms may prove to be surprisingly efficient from a cost-of-running point of view. In two separate cases where explicit comparisons were made, programs using the algorithms presented in this book cost less to run than their large-machine counterparts. More recent tests of execution times for algebraic eigenvalue problems, roots of a function of one variable and function minimisation showed that the eigenvalue algorithms were by and large 'slower' than those recommended for use on large machines, while the other test problems were solved with notable efficiency by the compact algorithms. That 'small' programs may be more frugal than larger, supposedly more efficient, ones based on different algorithms to do the same job has at least some foundation in the way today's computers work.

Firstly, most machines are controlled by operating systems which charge for main memory storage and for transfers between this and backing store (disc, tape, etc). In both compilation (translation of the program into machine code) and

execution, a smaller program will probably attract fewer of these charges. On top of this, the time required to compile the program should be reduced.

Secondly, once the program begins to execute, there are housekeeping operations which must be paid for:

(i) to keep programs separate in a time-sharing environment, and
(ii) to access the various parts of the program and data within the space allocated to a single user.

Some recent studies by Dr Maurice Cox of the National Physical Laboratory, England, show that (ii) above requires about 90% of the time that the computer spends with a typical scientific computation. Only about 10% of the time goes to actual arithmetic. It is not unreasonable that a small program has simpler structures such as address maps and decision tables than a larger routine, and it is tempting to suggest that the computer may be able to perform useful work with the small program while deciding what to do with the larger one. This explanation is largely speculation at the moment of writing, since hard evidence is not yet available. Moreover, this book appears to be the first attempt to select and present algorithms which have low storage requirements. This is in part due to the very recent appearance of small computers and programmable calculators. Of course, the very early electronic computers were small in the sense of memory space. However, they were also very unlike modern small computers in the manner in which they were programmed and operated. The demands on the programmer himself to handle such fundamental operations as floating-point arithmetic and simple mathematical functions have largely disappeared, though there is unfortunately still a need to watch for errors in the manufacturers' floating-point arithmetic and special functions.

Besides the motivation of cost savings or the desire to use an available and possibly under-utilised small computer, this work is directed to those who share my philosophy that human beings are better able to comprehend and deal with small programs and systems than large ones. That is to say, it is anticipated that the costs involved in implementing, modifying and correcting a small program will be lower for small algorithms than for large ones, though this comparison will depend greatly on the structure of the algorithms. By way of illustration, I implemented and tested the eigenvalue/vector algorithm (algorithm 13) in under half an hour from a 10 character/second terminal in Aberystwyth using a Xerox Sigma·9 computer in Birmingham. The elapsed time includes my instruction in the use of the system which was of a type I had not previously encountered. I am grateful to Mrs Lucy Tedd for showing me this system. Dr John Johns of the Herzberg Institute of Astrophysics was able to obtain useful eigensolutions from the same algorithm within two hours of delivery of a Hewlett–Packard 9825 programmable calculator. He later discovered a small error in the prototype of the algorithm.

The topics covered in this work are numerical linear algebra and function minimisation. Why not differential equations? Quite simply because I have had very little experience with the numerical solution of differential equations except by techniques using linear algebra or function minimisation. Within the two broad

areas, several subjects are given prominence. Linear equations are treated in considerable detail with separate methods given for a number of special situations. The algorithms given here are in fact very much the same as those used on large machines. The algebraic eigenvalue problem of symmetric matrices is discussed quite extensively, while that of non-symmetric and complex matrices is only touched upon briefly. This is largely a reflection of the comparative occurrence of these problems in real-world situations, though it must be pointed out that algorithms for the general square-matrix eigenproblem are complicated considerably by the inherent difficulties which attend the problem. Furthermore, one must deal with complex numbers on machines where the type COMPLEX, if it exists, may be handled unreliably. Constrained optimisation is likewise given only brief attention because the development of methods to handle this problem is proving a difficult and tortuous task. When the number of constraints is large, it is commonly called the mathematical programming problem. In practice it usually has a very simple, if large, structure but is nonetheless generally attacked by a method which ignores such simplifying structure because all too often the simplification possible in one problem does not occur in another.

Since the aim has been to give a problem-solving person some tools with which to work, the mathematical detail in the pages that follow has been mostly confined to that required for explanatory purposes. No claim is made to rigour in any 'proof', though a sincere effort has been made to ensure that the statement of theorems is correct and precise.

1.2. MACHINE CHARACTERISTICS

For the purpose of the discussion a *small computer* will mean any computing device, or share of such, which allows the user about 6000 characters of memory space to hold his program and data. The program will not include input–output routines, basic mathematical functions or system utilities, nor any compiler or interpreter. This definition, then, is of the logical structure which is available to the user. It corresponds reasonably well to the minimum configuration of a number of desk-top computers or programmable calculators as well as to the pieces of larger machines which are made available to users by time-sharing or student-teaching systems. The term 'small computer' will henceforth apply to this logical structure which faces the user. This machine will be programmable in some language or other, usually high-level, to facilitate its use. Increasingly the languages available on such systems are nominally the same as those on large machines. However, if standardisation has proved an elusive goal in the domain of large computers, it has yet to be sighted in the realm of small ones. Indeed, this has dictated the form of the algorithms presented later. Unfortunately, most of us who must solve problems with computers seem to have almost no say over the characteristics of the machines and systems with which we must work. Too often, we ride piggy-back on some or other data-processing (as opposed to computing) operation. It is no use listening to the programming language specialist who chides, 'No one in his right mind programs in XXX', where XXX happens to be the only language on the system available to us. Since there is a job to be done,

we must do the best with the tools we have unless we have the wherewithal to change them.

This should not lead to complacency in dealing with the machine but rather to an active wariness of any and every feature of the system. A number of these can and should be checked by using programming devices which force the system to reveal itself in spite of the declarations in the manual(s). Others will have to be determined by exploring every error possibility when a program fails to produce expected results. In most cases programmer error is to blame, but I have encountered at least one system error in each of the systems I have used seriously. For instance, trigonometric functions are usually computed by power series approximation. However, these approximations have validity over specified domains, usually $[0, \pi/4]$ or $[0, \pi/2]$ (see Abramowitz and Stegun 1965, p 76). Thus the argument of the function must first be transformed to bring it into the appropriate range. For example

$$\sin(\pi - \phi) = \sin\phi \tag{1.1}$$

or

$$\sin(\pi/2 - \phi) = \cos\phi. \tag{1.2}$$

Unless this range reduction is done very carefully the results may be quite unexpected. On one system, hosted by a Data General NOVA, I have observed that the sine of an angle near π and the cosine of an angle near $\pi/2$ were both computed as unity instead of a small value, due to this type of error. Similarly, on some models of Hewlett-Packard pocket calculators, the rectangular-to-polar coordinate transformation may give a vector 180° from the correct direction. (This appears to have been corrected now.)

Since most algorithms are in some sense iterative, it is necessary that one has some criterion for deciding when sufficient progress has been made that the execution of a program can be halted. While, in general, I avoid tests which require knowledge of the machine, preferring to use the criterion that no progress has been made in an iteration, it is sometimes convenient or even necessary to employ tests involving tolerances related to the structure of the computing device at hand.

The most useful property of a system which can be determined systematically is the machine precision. This is the smallest number, eps, such that

$$1 + \text{eps} > 1 \tag{1.3}$$

within the arithmetic of the system. Two programs in FORTRAN for determining the machine precision, the radix or base of the arithmetic, and machine rounding or truncating properties have been given by Malcolm (1972). The reader is cautioned that, since these programs make use of tests of conditions like (1.3), they may be frustrated by optimising compilers which are able to note that (1.3) in exact arithmetic is equivalent to

$$\text{eps} > 0. \tag{1.4}$$

Condition (1.4) is not meaningful in the present context. The Univac compilers

have acquired some notoriety in this regard, but they are by no means the only offenders.

To find the machine precision and radix by using arithmetic of the computer itself, it is first necessary to find a number q such that $(1+q)$ and q are represented identically, that is, the representation of 1 having the same exponent as q has a digit in the $(t-1)$th radix position where t is the number of radix digits in the floating-point mantissa. As an example, consider a four decimal digit machine. If $q = 10\,000$ or greater, then q is represented as (say)

$$0{\cdot}1 * 1E5$$

while 1 is represented as

$$0{\cdot}00001 * 1E5.$$

The action of storing the five-digit sum

$$0{\cdot}10001 * 1E5$$

in a four-digit word causes the last digit to be dropped. In the example, $q = 10\,000$ is the smallest number which causes $(1+q)$ and q to be represented identically, but any number

$$q > 9999$$

will have the same property. If the machine under consideration has radix R, then any

$$q \geqslant R^t \tag{1.5}$$

will have the desired property. If, moreover, q and R^{t-1} are represented so that

$$q < R^{t-1} \tag{1.6}$$

then

$$q - R > q. \tag{1.7}$$

In our example, $R = 10$ and $t = 4$ so the largest q consistent with (1.6) is

$$q = 10^5 - 10 = 99\,990 = 0{\cdot}9999 * 1E5$$

and

$$99\,990 - 10 = 100\,000 = 0{\cdot}1000 * 1E6 > q.$$

Starting with a trial value, say $q = 1$, successive doubling will give some number

$$q = 2^k$$

such that $(q+1)$ and q are represented identically. By then setting r to successive integers 2, 3, 4, . . . , a value such that

$$q - r > q \tag{1.8}$$

will be found. On a machine which truncates, r is then the radix R. However, if the machine rounds in some fashion, the condition (1.8) may be satisfied for $r < R$. Nevertheless, the representations of q and $(q+r)$ will differ by R. In the example,

doubling will produce $q = 16\,384$ which will be represented as

$$0{\cdot}1638 * 1E5$$

so $q+r$ is represented as

$$0{\cdot}1639 * 1E5$$

for some $r \leqslant 10$. Then subtraction of these gives

$$0{\cdot}0001 * 1E5 = 10.$$

Unfortunately, it is possible to foresee situations where this will not work. Suppose that $q = 99\,990$, then we have

$$0{\cdot}9999 * 1E5 + 10 = 0{\cdot}1000 * 1E6$$

and

$$0{\cdot}1000 * 1E6 - 0{\cdot}9999 * 1E5 = R'.$$

But if the second number in this subtraction is first transformed to

$$0{\cdot}0999 * 1E6$$

then R' is assigned the value 100. Successive doubling should not, unless the machine arithmetic is extremely unusual, give q this close to the upper bound of (1.6).

Suppose that R has been found and that it is greater than two. Then if the representation of $q+(R-1)$ is greater than that of q, the machine we are using *rounds*, otherwise it *chops* or *truncates* the results of arithmetic operations.

The number of radix digits t is now easily found as the smallest integer such that

$$R^t + 1$$

is represented identically to R^t. Thus the machine precision is given as

$$\text{eps} = R^{1-t} = R^{-(t-1)}. \tag{1.9}$$

In the example, $R = 10$, $t = 4$, so

$$R^{-3} = 0{\cdot}001.$$

Thus

$$1 + 0{\cdot}001 = 1{\cdot}001 > 1$$

but $1 + 0{\cdot}0009$ is, on a machine which truncates, represented as 1.

In all of the previous discussion concerning the computation of the machine precision it is important that the representation of numbers be that in the memory, not in the working registers where extra digits may be carried. On a Hewlett–Packard 9830, for instance, it is necessary when determining the so-called 'split precision' to store numbers specifically in array elements to force the appropriate truncation.

While the subject of machine arithmetic is still warm, note that the mean of two numbers may be calculated to be smaller or greater than either! An example in four-figure decimal arithmetic will serve as an illustration of this.

	Exact	Rounded	Truncated
a	5008	5008	5008
b	5007	5007	5007
$a+b$	10015	1002 * 10	1001 * 10
$(a+b)/2$	5007·5	501·0 * 10	500·5 * 10
		= 5010	= 5005

That this can and does occur should be kept in mind whenever averages are computed. For instance, the calculations are quite stable if performed as

$$(a+b)/2 = 5000 + [(a-5000)+(b-5000)]/2.$$

Taking account of every eventuality of this sort is nevertheless extremely tedious.

Another annoying characteristic of small machines is the frequent absence of extended precision, also referred to as double precision, in which extra radix digits are made available for calculations. This permits the user to carry out arithmetic operations such as accumulation, especially of inner products of vectors, and averaging with less likelihood of catastrophic errors. On equipment which functions with number representations similar to the IBM/360 systems, that is, six hexadecimal ($R = 16$) digits in the mantissa of each number, many programmers use the so-called 'double precision' routinely. Actually $t = 14$, which is not double six. In most of the calculations that I have been asked to perform, I have not found such a sledgehammer policy necessary, though the use of this feature in appropriate situations is extremely valuable. The fact that it does not exist on most small computers has therefore coloured much of the development which follows.

Finally, since the manufacturers' basic software has been put in question above, the user may also wonder about their various application programs and packages. While there are undoubtedly some 'good' programs, I must report that my own experience is that nothing is often preferable to something in this field, since badly written and poorly documented programs take longer to learn and understand and cause more trouble when they go wrong than a homegrown program implemented from scratch. Hopefully, this situation will soon change, but until programs are available which are proud enough to give their pedigree, that is, author/implementor and any modifications, together with performance evaluations (not just 'I ran it'), my advice to the user is 'beware'.

1.3. SOURCES OF PROGRAMS

There are at least three groups, at the time of writing, which are trying to provide high-quality mathematical software on a continuing basis. The Numerical Algorithms Group, based in Oxford, England, supplies a large subroutine library in FORTRAN, ALGOL and ALGOL 68 to a university and institutional (and more recently private) user community. I worked with NAG for two periods of several weeks in 1975–76, and some of the material in this book was to have been born of condensations of NAG routines. As it has turned out, the ideas of the many

workers who have contributed to the NAG library were more important than the routines individually, so that the algorithms I have included are more a distillate than a condensation.

The Applied Mathematics Division of the Argonne National Laboratory, near Chicago, is the centre of the US National Activity for Testing Software. Argonne is the home of EISPACK, a package designed to handle nearly every imaginable algebraic eigenproblem; FUNPACK, a set of special function routines; and MINPACK, a study of minimisation programs. LINPACK, a suite of programs to solve various linear-equation and least-squares problems, should be released for distribution shortly. These are all in FORTRAN. The Argonne staff are the most thorough I know when it comes to testing. Up to the time of writing, their interest has been in medium to large machines and their philosophy has been very different from that which I present here. Where I use a heuristic device to obtain a result, my opinion is that Argonne workers will determine the exact causes and effects of each situation and program accordingly; certainly a more correct but usually a less compact approach.

International Mathematical and Statistical Libraries (IMSL) of Houston, Texas, is a private corporation which provides a FORTRAN subroutine library to users of medium- to large-scale machines. The IMSL collection is quite strong in statistical routines, and my colleagues who have used it praised the quality of the algorithms but disliked the somewhat cumbersome subroutine calls necessary to use them (a similar criticism has been made of NAG routines).

Finally on sources of software, readers should be aware of the Association for Computing Machinery (ACM) Transactions on Mathematical Software which publishes research papers and reports algorithms.

1.4. PROGRAMMING LANGUAGES USED AND STRUCTURED PROGRAMMING

The machines on which the algorithms given in the following pages are designed to run are programmed in a diverse collection of programming languages. It was my original intention to give the algorithms in BASIC which is the language I mainly, but by no means exclusively, use for program development. However, in addition to the absence of standardisation in BASIC, there are many machines with FORTRAN or APL, ALGOL, ALGOL 68 or even some exotic language of their very own, such as the Hewlett–Packard 9810 and 9820 series. There is no explicit barrier to languages such as PL/1 or COBOL either, though I have yet to find small computers (under the definition of §1.2) which can be programmed in these languages. However, new ones are being devised all the time and one of BCPL, C or PASCAL is very likely to become common in the future.

In recent years, the concepts of structured and modular programming have become very popular. The interested reader is referred to Kernighan and Plauger (1974) or Yourdon (1975) for an exposition of these topics. I have found both structured and modular programming to be useful in my own work and recommend them to any practitioner who wishes to keep his debugging and re-programming efforts to a minimum. Nevertheless, while modularity is relatively easy to

impose at the level of individual tasks such as the decomposition of a matrix or the finding of the minimum of a function along a line, it is not always reasonable to insist that the program avoid GOTO instructions. After all, in aiming to keep memory requirements as low as possible, any program code which can do double duty is desirable. If possible, this should be collected into a subprogram. In a number of cases this will not be feasible, since the code may have to be entered at several points. Here the programmer has to make a judgement between compactness and readability of his program. I have opted for the former goal when such a decision has been necessary and have depended on comments and the essential shortness of the code to prevent it from becoming incomprehensible.

One of the unfortunate aspects of the most popular of the programming languages is their lack of constructions which permit the structuring of programs. In particular, the form

IF . . . X . . . THEN . . . A . . . ELSE . . . B

is usually absent. This can be replaced by

```
IF . . . X . . . THEN GOTO A
B
GOTO  Rest of program
A
Rest of program.
```

In BASIC, particularly, the IF . . . GOTO . . . statement is mandatory.

1.5. CHOICE OF ALGORITHMS

The algorithms which appear in this work have been chosen for their utility and for their particular suitability for implementation on small computers. Many topics I would have liked to discuss are deliberately left out because there has been insufficient development to make them suitable for tackling on a small machine. For instance, over the last four years I have spent a considerable amount of human and computer time trying to devise a method for the solution of mathematical programming problems which does not involve the construction of a simplex tableau. Other workers are no doubt pursuing the same objective, spurred on by the enticing simplicity of many mathematical programming problems in the form in which they are stated and the desire to avoid the increased memory requirement imposed by slack, surplus and artificial variables. Alas, while some techniques based on the conjugate gradients minimisation with simple penalty functions for the constraints seem to be able to compute good approximations to the solutions, the rate of convergence is measurable in years rather than seconds. At the time of writing, the simplex algorithm of Dantzig appears the most space conservative to implement (Gass 1964, Hadley 1962). If a good backing store is available on a small computer, the Revised Simplex algorithm may be better, but few small computer systems have fast scratch file capability. Neither of these algorithms should be confused with the Nelder Mead simplex algorithm for general function minimisation.