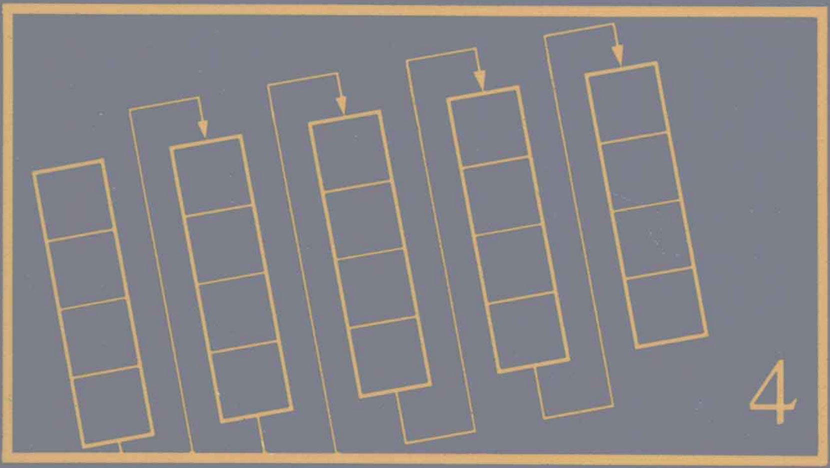

F·R·O·N·T·I·E·R·S
IN APPLIED MATHEMATICS

Handbook for Matrix Computations

Thomas F. Coleman
Charles Van Loan

siam

Handbook for Matrix Computations

Thomas F. Coleman
Charles Van Loan
Cornell University

siam

Philadelphia 1988

Library of Congress Catalog Card Number 88-61637.
ISBN 0-89871-227-0

Second printing July 1989.
Third printing April 1991.

Handbook for Matrix Computations

Frontiers in Applied Mathematics

Frontiers in Applied Mathematics is a series of monographs that present new mathematical or computational approaches to significant scientific problems. Beginning with Volume 4, this series will reflect a change in both philosophy and format. Each volume will focus on a broad application of general interest to applied mathematicians as well as engineers and other scientists.

This unique series will advance the development of applied mathematics through the rapid publication of short, inexpensive monographs that lie on the cutting edge of research. The first volume in the new softcover format is in the Practical Computing series. Future volumes will include

fluid dynamics
solid mechanics
nonlinear dynamical systems
materials science
control theory
mathematical physics.

Potential authors who are now writing or plan to write books within the scope of the series are encouraged to contact SIAM for additional information.

Publisher
SIAM
3600 University City Science Center
Philadelphia, PA 19104-2688
(215) 382-9800

Contents

Preface

This handbook can be used as a reference by those actively engaged in scientific computation. It can also serve as a practical companion text in a numerical methods course that involves a significant amount of linear algebraic computation. The book has four chapters, each being fairly independent of the others.

Our treatment of Fortran 77 in Chapter 1 involves a much stronger emphasis on arrays than is accorded by other authors. We also assume that the reader has experience with some high-level programming language. This might be in the form of a recent course in Pascal or a course in Fortran taken many years ago and now half-forgotten.

The second chapter is about the Basic Linear Algebra Subprograms (BLAS). The elementary linear algebra that underpins the BLAS makes them a good vehicle for acquainting the beginning student with modular programming and the importance of "thinking vector" when organizing a matrix computation.

Chapter 3 is concerned with LINPACK, a highly acclaimed package that is suitable for many linear equation and least square calculations. The last chapter is about MATLAB, an interactive system in which it is possible to couch sophisticated matrix computations at a very high level.

A one-semester course in matrix algebra (or the equivalent) is required to understand most of the text.

Because the book spans several levels of practical matrix computations, it can fit into a number of canonically structured numerical methods courses. At Cornell we use Chapters 1 and 2 in our one-semester introductory numerical methods course. In this course it is assumed that the students are acquainted with Pascal. That is why our treatment of Fortran is brisker than what would be found in a "pure" Fortran text. In our graduate-level numerical analysis courses we use Chapters 2, 3, and 4 heavily, with Chapter 1 serving as a reference.

The BLAS and LINPACK are in the public domain and are distributed at cost through Argonne National Laboratory. MATLAB is available from MGA Inc., 73 Junction Square Dr., Concord, MA 01742.

We are indebted to Nick Higham, Bill Coughran, and Eric Grosse for catching numerous typographical errors and for making many valuable suggestions.

Thomas F. Coleman
Charles Van Loan

Chapter 1

A Subset of Fortran 77

Our treatment of Fortran 77 (F77) assumes that the reader is already familiar with some high-level language, e.g., PASCAL. There are many books devoted to the presentation of the basics of Fortran 77(see, e.g., Zwass [8]). We do not attempt to be exhaustive at this level. For example, we say nothing about the "opening" and "closing" of files, and character manipulation is mentioned only briefly. Rather, our emphasis is on matrix computations and we have attempted to be complete in this regard. We pay special attention to arrays since the implementation of matrix algorithms is an underlying theme of the book.

We believe that it is still important for students with a serious interest in scientific computation to be familiar with Fortran. This is not to say that we are advocating that all scientific computing be done in Fortran. On the contrary, the most appropriate language for a particular application at hand should be used. The language "C" is an increasingly popular choice. MATLAB is well suited to dense matrix computations and graphical work. However, we strongly believe that high-quality subroutines should be used as building blocks whenever possible. For reasons of portability and standardization, this software is almost always Fortran software and will probably continue to be so for many years. Therefore, familiarity with Fortran is essential.

A scientific programmer working in a language other than Fortran can often use Fortran subroutines directly, provided the language and computer system support such an interface. Otherwise a direct translation can be used, provided that extreme care is taken. In either case, Fortran knowledge is required.

We make two final observations about the use of Fortran. First, Fortran should seriously be considered when developing general software for a basic mathematical computation with widespread applicability. Fortran programs that adhere to professionally set "standards" are easily ported to other computing systems. This is not true for other languages. Second, it is important to realize that Fortran is not a "dead" programming language. The "modernization" of Fortran (and the subsequent updating of the "standards") is an ongoing process. Fortran 77 is now widely used and accepted (replacing Fortran 66). Fortran 8X is currently being developed. Each new Fortran basically inherits the old version as a subset to which the extensions add power and flexibility.

1.1 BASICS

We begin the discussion with a Fortran program that computes the surface area of a sphere from the formula $A = 4\pi r^2$:

```
      program area
      real r, area
c
c  This program reads a real number r and
c  prints the surface area of a sphere that
c  has radius r.
c
      read(*,*)r
      area = 4.*3.14159*r*r
      write(*,*)area
      stop
      end
```

The purpose of each line in this program is quite evident. Those lines that begin with a "`c`" are *comments*. The **read** and **write** statements perform input and output. The computation of the surface area takes place in the arithmetic assignment statement "`area = ...`". The beginning and end of the program are indicated by the **program** and the **end** statements. Execution is terminated by the **stop**. The memory locations for the variables used by the program are set aside by the line "`real r, area`".

In this section we elaborate on these and a few other elementary constructs.

Program Organization

A Fortran program generally consists of a main program (or "driver") and several subprograms (or "procedures"). Typically the main program and a few of the subprograms are written by the user. Other subprograms may come from a "library." Subprograms are discussed later.

A main program is structured as follows:

```
program { name }

    { declarations }

    { other statements }

stop
end
```

Each line in a Fortran code must conform to certain column position rules.

Column 1	:	Blank unless the line is a comment.
Columns 2-5	:	Statement label (optional).
Column 6	:	Indicates continuation of previous line (optional).
Columns 7-72	:	The Fortran statement.
Columns 73-80	:	Sequence number (optional).

Comments

A line that begins with a "c" in the first column is a comment. Comments may appear anywhere in a program and are crucial to program readability. Comments should be informative, well written, and sufficiently "set off" from the body of the program. To accomplish the latter begin and end each comment block with a blank comment as in the surface area program above.

Sequence Numbers

Every line in a program can be numbered in columns 73-80. This used to be a common practice but in the age of screen editors sequence numbering has become less useful for debugging.

List-directed "read" and "write"

Until we cover input/output (I/O) in depth in §1.7, we rely upon two elementary I/O constructs:

read(*,*) { *list-of-variables* }

write(*,*) { *list-of-variables* }

Thus,

```
read(*,*) x, y, z
```

reads three items of data and stores them in the variables named x, y, and z. Similarly,

```
write(*,*)a, b
```

prints the contents of the variables named a and b.

Messages (enclosed in quotes) may be interspersed with variable names in a **write** statement. Thus, if a and b contain 2 and 3, respectively, then

```
write(*,*)'a = ', a,'b = ', b
```

produces the output

```
a = 2.0000      b = 3.0000
```

An important detail suppressed in our I/O discussion is where a **read** physically obtains its data and where a **write** physically sends its output. The "asterisk" notation in a **read** or a **write** specifies a default device, e.g., the keyboard, the terminal screen, etc. How the default devices are set depends upon the system and it is necessary to obtain local instructions for their use.

Names in Fortran

Names in Fortran must involve no more than six characters chosen from the alphanumeric set

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789
```

The name must begin with a letter. In our examples we do *not* distinguish between upper and lower case. However, it should be noted that a few F77 compilers accept only uppercase input.

Types and Declarations

Every variable in a Fortran program should be defined in a *declaration*. This establishes the variable's *type*. There are several possibilities:

integer { *list-of-variables* }

real { *list-of-variables* }

double precision { *list-of-variables* }

complex { *list-of-variables* }

logical { *list-of-variables* }

character { *list-of-variables* }

Thus, a program may begin with the following declarations:

```
double precision  a, b, c
integer n
complex z1, z2
double precision length, width
logical test
character*20 name
```

There are a few rules to follow regarding the naming and typing of variables.

- Variable names must begin with a letter and must be no more than six alphanumeric characters in length.
- Each variable should be declared exactly once. Automatic typing occurs otherwise. This means that variables whose names begin with letters `i` through `n` are integers and all others are real.

- It is legal to have more than one declaration per type in a program.

- Declarations must be placed at the beginning of a program, before any executable statement. This is because their purpose is to set aside memory locations for the variables used by the program.

We discuss the integer, real, double precision, and character types now. Logical and complex variables are discussed in §1.2 and §1.8, respectively.

Integer Variables

Numbers stored in integer variables are represented in *fixed point* style. In a typical computer 32 bits, $b_0, b_1, \ldots, b_{31}$, might be allocated for each integer variable x with the convention that x has the value

$$x = (-1)^{b_{31}} \times (b_{30}b_{29}\ldots b_1 b_0)_2 \qquad b_i \in \{0,1\} .$$

The notation $(\cdot)_2$ is amply illustrated by the example

$$(01101)_2 = 1\times 2^0 + 0\times 2^1 + 1\times 2^2 + 1\times 2^3 + 0\times 2^4 = 13$$

Note that because of the finiteness of an integer "word" there is an upper bound on the size of the integers that can be represented. In the 32-bit example, only integers in the interval $[-m, m]$, where $m = 2^{31} - 1 \cong 2 \times 10^9$, can be represented.

Floating Point Variables

Numbers that are stored in real or double precision variables are represented in *floating point* style. The floating point word is partitioned into a mantissa part, *m*, and an exponent part, *e*, with the convention that the value of the variable is specified by $m\,2^e$. The length of a floating point word and how it is partitioned into the exponent and mantissa parts depends upon the computer used. A typical 32-bit floating point number *x* might have a 24-bit mantissa *m* and an 8-bit exponent *e*, with the convention that

$$m = (-1)^{b_{23}} \times (.b_0 b_1 \cdots b_{22})_2$$

and

$$e = (-1)^{b_{31}} \times (b_{30} \cdots b_{24})_2$$

with the convention that $x = m2^e$. Stipulating that $b_0 \neq 0$ makes the representation of a given floating point number unique. (An exception to this rule is required when $x = 0$.)

Double precision variables represent numbers in the same fashion as do real variables, but more space is allocated per variable. For example, if 32-bit words are used for real variables then typically 64-bit words would be used for double precision variables. This leaves more bits for mantissa specification, e.g., 56 bits instead of 24 .

Regardless of precision, there are finitely many floating point numbers, and *rounding errors* generally arise with every arithmetic operation. Moreover, an arithmetic operation (such as a divide by a very small number) may lead to a number that is "too big" to represent in the floating point system. *Overflow* results, a situation that usually leads to program termination. Some of the hazards of floating point computation are discussed in §1.9.

Arithmetic Assignment

An arithmetic assignment statement has the form

{ *variable name* } = { *expression* }

The expression on the right-hand side is evaluated according to precise rules and the result is stored in the memory location corresponding to the variable on the left-hand side. The values of the variables on the right-hand side do not change as a consequence of the assignment. For example,

```
A = pi*r**2
```

would compute the area of the circle and store the result in the variable `A`, assuming that `pi` and `r` are appropriately initialized. An asterisk denotes multiplication, whereas a double asterisk specifies exponentiation.

A more complicated assignment statement is

```
root = -b + sqrt(b*b - 4.*a*c)/2.*a
```

The expression to the right of the "=" involves one of Fortran's numerous "built-in" (or intrinsic) functions, the square root. We introduce various built-in functions throughout the text. A complete list is given in Appendix 1.

Readers familiar with solving quadratic equations will recognize that the above assignment statement does *not* compute a zero of $ax^2 + bx + c = 0$. The order in which the operations are to be performed is not correctly specified. Indeed, the above assignment statement is equivalent to

```
root =  -b + ( (sqrt(b*b - 4.*a*c)/2. )*a )
```

The problem is one of *precedence*. Unless overridden by parentheses, an exponentiation (**) has a higher precedence than a multiplicative operation such as "/" or "*" which in turn has a higher precedence than an additive operation such as "+" or "-". Thus

```
w = x + y/z**2      ⇔      w =   x + ( y/(z**2) )
```

If a choice has to be made between two multiplicative operations or two additive operations, then the leftmost operation in the expression is performed first. On the other hand, repeated exponentiations are processed from right to left. These examples should clarify the possibilities:

```
w = x + y + z       ⇔      w = (x + y ) + z
w = x/y*z           ⇔      w = (x/y)*z
w = x**y**z         ⇔      w = x**(y**z)
```

Parentheses should be used to prescribe the correct order of computation to both the compiler and the reader in ambiguous cases. Thus,

```
root = ( -b + sqrt(b*b - 4.*a*c) )/(2.*a)
```

correctly assigns a zero of $ax^2 + bx + c = 0$ to the variable `root`, provided the argument passed to the square root function is nonnegative at the time of execution.