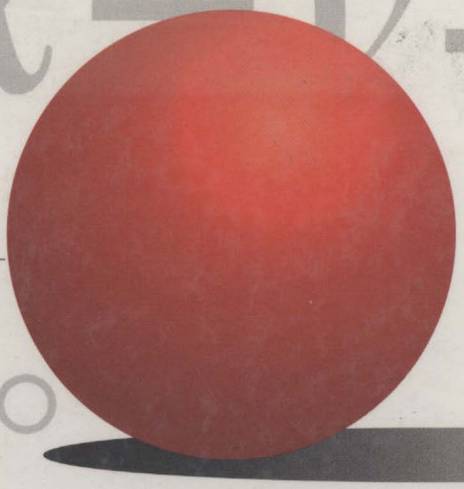
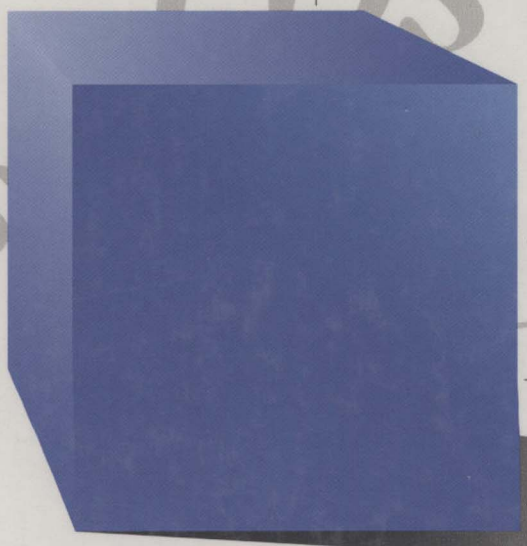
Computer Routines Included on Disk

SOLVE IT!



COMPUTER-AIDED MATHEMATICS FOR SCIENCE AND ENGINEERING



SAMUEL DOUGHTY

Reduce your dependence on special-purpose, commercial software

SOLVE IT!

COMPUTER-AIDED MATHEMATICS FOR SCIENCE AND ENGINEERING

SAMUEL DOUGHTY

Gulf Publishing Company
Houston, London, Paris, Zurich, Tokyo

SOLVE IT!

Computer-Aided Mathematics for Science and Engineering

Gulf Publishing Company
Book Division
P.O. Box 2608 ☐ Houston Texas 77252-2608

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

Library of Congress Cataloging-in-Publication Data
Doughty, Samuel.
Solve it : computer-aided mathematics for science and engineering / Samuel Doughty.
p. cm.
Includes bibliographical references and index.
ISBN 0-88415-266-9 (alk. paper)
1. Engineering mathematics—Data processing.
2. Science—Mathematics—Data processing.
TA345.D68 1995
519.4'0285'51—dc20 95-30439
CIP

Printed on Acid-Free Paper (∞).

SOLVE IT!

COMPUTER-AIDED MATHEMATICS FOR SCIENCE AND ENGINEERING

For Ann-Marie, John, and Jenni

Foreword

The landscape of applied mathematics has been greatly changed by the computer, and more particularly, by the microcomputer that is found on virtually every desk. Before the computer, many solution techniques led to *formal solutions*, solutions that showed the form of the result but often only indicated some major computational effort required to actually evaluate the result. The implication was, "Here is what it will look like and what is required, but it will only be usable if you are willing to do this rather substantial amount of computation to implement it." Today, that "rather substantial amount of computation" is often readily done with the computer.

Even with the availability of the computer, there is still the need for detailed computation instructions known as *algorithms*. For example, many formal solutions involve an indicated matrix inverse, but the details of the matrix inversion are not specified. There are numerous methods for matrix inversion, each preferred for a particular situation. Before the formal solution can be turned into a numerical solution, a specific method of inversion must be selected and the detailed computational steps must be specified, that is, an algorithm must be established.

This book is not intended to be a comprehensive book on numerical analysis. Rather, it is intended to be a companion, a knowledgeable friend, offering help in those areas where it has experience. That experience comes from the writer's approximately thirty years of numerical solutions for problems in science and engineering. The topics included are those the writer has found to be useful, but which are not readily available in a single source. These matters range from topics that will be useful to a college freshman, such as curve fitting, to topics more likely to useful to the practicing professional, such as applications of Green's theorem.

It is the author's hope that this book will become a friend of the user early in his college career and remain with the user into his professional life. The term *user* is employed, rather than *reader*, because it is the author's intent that this book be *used*, not simply read. The book is written with this purpose in mind, and it frequently includes sections of sample computer code, often subroutines or main programs, which the user is encouraged to first program as they stand, then check, debug, and adapt the program to the user's problems.

The author's intent has been to present the topics as independent units. The exception is the discussion on matrices and matrix notation that appears quite early. This material is used rather freely throughout the remainder of the book, and the user will do well to read and understand the use of matrices to this extent before plunging into the remainder of the book.

Considerable effort has been expended to eliminate errors, but no doubt some remain. The author will be grateful to those users who pass on corrections, comments, and suggestions for improvement.

Samuel Doughty

SOLVE IT!

COMPUTER-AIDED MATHEMATICS FOR SCIENCE AND ENGINEERING

Contents

Chapter 1

Philosophy

Make It Easy For Whom?

There are many reasons to use digital computation today including labor reduction, elimination of random, careless mistakes, and increased accuracy. The major reason, in most cases is the first one, labor reduction. This includes the opportunity to use methods that would be simply too tedious for manual computation and are only accessible through the digital computer. If labor reduction is the primary concern, then it is important to assure that human labor is really reduced. All too often, human effort is expended to make the work easy for the machine, and this is usually contrary to the goal of reducing human labor.

Some people will make the elementary mistake of trying to save effort in the execution of the program by precomputing various constants to be inserted into the code directly. A common example of this is the calculation of the value of 2π and the programming of a decimal approximation to this value. If the programming language in use has an internal value for π, then the reasonable approach is to let the machine make the multiplication by 2 whenever the program is executed. This will give a value with as many significant figures as are being calculated within the program. If the language does not have an internal value for π, then such a value should be programmed once, near the beginning of the code, and used by name thereafter. The repeated programming of a decimal approximation is both tedious and a frequent source of errors.

A more subtle waste of time occurs in seeking more and more complicated programming techniques, usually for reduced execution time. In many technical applications, the final program will be executed only once or a few times to obtain the desired results, and this is the sort of computation discussed here. For applications such a real-time control, there is clearly a need for minimum execution time, but that is outside the scope of this book. There is obviously a trade-off to be made here, because it is human time that is often wasted while waiting for execution to be completed. Even so, there is little gained by spending an extra hour of programming time in order to reduce execution time by a few milliseconds total. Execution time becomes a problem when it is measured in hours or even days, but relatively few programs approach such times.

A second waste of time often involved with more complicated programming comes from the fact that the resulting code is more difficult to debug. By developing complicated code, particularly long, complicated individual statements, the time required to search out errors is often multiplied many times.

Write It Yourself or Go Commercial?

There is a dazzling array of commercial software available today to do everything from managing your social calendar to arranging musical scores. The software vendors would like for you to believe that their product is superior to anything that you could write, and that to do a satisfactory job, you have no choice but to use their product. For a computing problem for which you do not understand the process, they may very well be correct. For any task that you understand how to do well, there is often no need to go to a commercial program. There certainly is no need to look for a commercial code for any sort of calculation where you are the expert and they are the amateurs!

One of the principal advantages of writing the code yourself is that you know exactly what it does, exactly what assumptions are built into the process. If there is a need to revise the solution, you can certainly do so if you wrote the code in the first place. If you are using a commercial code and need to make a revision, there is really very little that you can do other than to petition the software vendor for a revision or look for another vendor. The security of knowing exactly how the program works, and the associated flexibility to make changes as needed, often outweigh the short term advantages of using a commercial code, not to mention the cost of the commercial code. And remember, commercial codes cost twice: once for the software and again for the time and effort required to develop proficiency!

Programming Language

If you are going to write the code yourself, you must chose a programming language. The choice of programming language is often dictated by what is available. For most scientific and engineering computation (not real-time applications), a language that facilitates program development is the best choice. Depending on the nature of the work being programmed, features such as matrix operations and high level plotting commands are often significant.

Most microcomputers offer a version of BASIC, and this language lends itself to program development. There are, however, many different versions of BASIC, and they are not all equal. One of the most powerful versions is True BASIC, a product of True BASIC, Inc., West Lebanon, NH. True BASIC includes matrix operations and high level plotting commands, and execution is fairly fast on most problems. The author has found this language to have an optimum combination of programming commands, speed of execution, and ease of usage for most of his engineering consulting work. This is the language used for all of the examples in this book.

Chapter 2

Matrices

A user of this book might wonder why the opening discussion is on a topic as abstract as matrices. The reasons for this choice are

- although often presented abstractly, matrices are very useful for a wide variety of applied mathematics used in science and engineering, and
- matrix notation is quite close to what is needed for computer program development in many situations. Consequently, a problem formulated in matrix notation is usually rather easily programmed for computer numerical solution.

With these two thoughts in mind, matrices are a very good place to begin this presentation. It should be emphasized that very little matrix theory will be used; the emphasis is primarily on the notation and arithmetic of matrices.

Matrix Notation

The terms "matrix" or "matrix notation" as used in applied mathematics refer to a rectangular array of numbers. The word *matrix* is singular; to refer to more than one matrix, the ending changes to become *matrices*. Thus it is possible to have *one matrix* that is the sum of *two matrices*. The numbers in the array are called *elements* of the matrix, and the elements are arranged in rows and columns. A typical matrix, $[A]$ with n_r rows and n_c columns represents the following array:

$$[A] = \begin{bmatrix} a_{11} & a_{12} & a_{13} & \cdots & a_{1n_c} \\ a_{21} & a_{22} & a_{23} & \cdots & a_{2n_c} \\ a_{31} & a_{32} & a_{33} & \cdots & a_{3n_c} \\ \vdots & \vdots & \vdots & \ddots & \\ a_{n_r1} & a_{n_r2} & a_{n_r3} & & a_{n_rn_c} \end{bmatrix}$$

The typical element is denoted as a_{ij} where the first subscript is the *row index* and the second subscript is the *column index*. In this way, the indices on each element indicate its location within the array. All of the elements in the first row, have a 1 as the first index. Similarly, all of the elements in the second row have a 2

as the first index, etc. All of the elements in the first column have a 1 as the second index. All of the elements in the second column have a 2 for the second index, etc. The range of the row index is from 1 to n_r and is called the *row dimension* of the array. The range of the column index is from 1 to n_c, and this is called the *column dimension* of the array.

We are interested in two types of matrices. The first is called a square matrix. A matrix is square, provided that the row and column dimensions are equal, $n_r = n_c$. It is spoken of as an $(n \times n)$ array. In form, a square matrix looks just like the matrix $[A]$ above and requires no further comment at this point. The second form of interest is called a *column (or row) vector.* For a column vector, the column dimension is 1, so the array has the form

$$\{B\} = \begin{Bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{Bmatrix}$$

The braces, { }, are used to denote a column vector, in contrast with the brackets [ ] used above for a rectangular or square matrix. Because it is a one dimensional array, only a single index is required and there is only a single dimension, a row dimension. This same array can be written in row form as

$$(B) = (b_1, b_2, \cdots b_n)$$

where the parentheses ( ) denote the row format.

If the idea of a matrix as a two dimensional array is to be maintained, then the column vector should be considered as a $(n \times 1)$ array. Similarly, the row vector is a $(1 \times n)$ array. The value of this formal consistency with the rectangular array description will become more evident shortly.

One of the basic matrix operations is called the *transpose.* The transpose consists of interchanging the rows and columns in a rectangular matrix so that the elements comprising a row in the original matrix become a column in the transposed matrix. The transpose of the matrix $[A]$ above, written with indices that denote position in the original array, is

$$[A]^t = \begin{bmatrix} a_{11} & a_{21} & a_{31} & \cdots & a_{n_r 1} \\ a_{12} & a_{22} & a_{32} & \cdots & a_{n_r 2} \\ a_{13} & a_{23} & a_{33} & \cdots & a_{n_r 3} \\ \vdots & \vdots & \vdots & \ddots & \\ a_{1n_c} & a_{2n_c} & a_{3n_c} & & a_{n_c n_r} \end{bmatrix}$$

The superscript **t** on the matrix $[A]$ is not an exponent, but rather is the notation for the transpose of the matrix. When the transpose operation is applied to a column vector, the result is the row vector form for the same array, thus:

$$\{B\}^t = \begin{Bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{Bmatrix}^t = (b_1, b_2, \cdots b_n) = (B)$$

and similarly, the row form is converted to the column form by transposition:

$$(B)^t = (b_1, b_2, \cdots b_n)^t = \left\{ \begin{array}{c} b_1 \\ b_2 \\ \vdots \\ b_n \end{array} \right\} = \{B\}$$

Note that $[A]^{t^t} = [A]$, that is, two applications of the transpose operation simply restore a matrix to its original form.

At this point, it is appropriate to define two special matrix forms. A diagonal matrix is a square matrix with all elements zero, except for those on the *main diagonal*, which is the group of elements lying along the diagonal from the upper left corner to the lower right corner. Since the only nonzero values are found on the main diagonal, only those values need to be expressed if it is known that a particular matrix is diagonal. This is often written as

$$[D] = Diag\,(d_1, d_2, \cdots d_n)$$

where the notation $Diag\,(\cdots)$ conveys the fact that the matrix is diagonal and the d_i are the values of the diagonal elements.

The second definition required is that of the *identity* matrix. An identity matrix is a square, diagonal matrix with unit values on the main diagonal. Thus, for a dimension of 3, the identity matrix is

$$[I] = Diag\,(1,1,1) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The notation $[I]$ is widely used for the identity matrix, but caution should be used where this may be confused with other matrices such as the inertia matrix, which is often denoted by the same symbol. The identity matrix is in some ways comparable to a 1 in scalar arithmetic; it is the unit element in matrix arithmetic.

Addition and Subtraction of Matrices

Either addition or subtraction of two matrices produces a resultant matrix where each element of the result is the sum or difference of the corresponding terms in the two matrices being added or subtracted. Such a description makes sense only when the two original matrices have the same row and column dimensions. Further, the resultant matrix will have the same row and column dimensions as the two original matrices. If $[A]$ and $[B]$ are rectangular arrays having dimensions $(n_r \times n_c)$,

$$[A] = \begin{bmatrix} a_{11} & a_{12} & \cdots \\ a_{21} & a_{22} & \\ \vdots & & \ddots \end{bmatrix} \qquad [B] = \begin{bmatrix} b_{11} & b_{12} & \cdots \\ b_{21} & b_{22} & \\ \vdots & & \ddots \end{bmatrix}$$

the sum and difference are $[S]$ and $[D]$, respectively,

$$[S] = \begin{bmatrix} s_{11} & s_{12} & \cdots \\ s_{21} & s_{22} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} = \begin{bmatrix} a_{11}+b_{11} & a_{12}+b_{12} & \cdots \\ a_{21}+b_{21} & a_{22}+b_{22} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

$$[D] = \begin{bmatrix} d_{11} & d_{12} & \cdots \\ d_{21} & d_{22} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} = \begin{bmatrix} a_{11}-b_{11} & a_{12}-b_{12} & \cdots \\ a_{21}-b_{21} & a_{22}-b_{22} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}$$

Multiplication of Matrices

If two matrices are multiplied together, the result is another matrix. The elements of the result are the sums of products of the elements of the two matrix factors. To be more specific, and using the notation for the two matrices $[A]$ and $[B]$ defined above, the elements of the product matrix, $[P]$, are computed as:

$$p_{ik} = \sum_j a_{ij} b_{jk}$$

If the sum on j is to make sense, the range of values for j in the $[A]$ matrix (the column dimension of $[A]$) must be the same as the range of values for j in the $[B]$ matrix (the row dimension of $[B]$). Evidently i can take all of the values it was allowed in the $[A]$ matrix, while k can assume all values that were allowed for it in the $[B]$ matrix. The result is a square matrix having the row dimension from the first factor and the column dimension from the second factor:

$$\underbrace{[P]}_{n_1 \times n_3} = \underbrace{[A]}_{n_1 \times n_2} \underbrace{[B]}_{n_2 \times n_3}$$

Note that this notation indicates a common value for the summed index, n_2, as shown. It also shows the row dimension of the product as the row dimension of the first factor and the column dimension of the product as the column dimension of the second factor. If any of this condition is not met, the matrix product is not defined.

It is necessary to point out that matrix multiplication is *noncommutative*, which is to say that the order of the factors is significant. This means that, in general $[A][B] \neq [B][A]$, even if both products are properly defined. There may be particular instances in which the order of the factors may be reversed, but this must be verified for the particular situation. One example of a case where matrices do commute is found when the identity matrix is one of the factors:

$$[A][I] = [I][A] = [A]$$

Multiplying by the identity matrix is comparable to scalar multiplication by 1.

It will often be useful to multiply matrices within a computer program. The following code segment, based on the expression for the typical term given above, illustrates how this is done.

Example Code for Matrix Product

```
for i = 1 to n1
   for k = 1 to n3
       sum = 0
       for j = 1 to n2
           sum = sum + a(i,j) * b(j,k)
       next j
   next k
next i
```

This example code assumes that the arrays `a(,)`, `b(,)` and `c(,)` have been previously defined and that `n1`, `n2`, and `n3` have the appropriate values to match the array dimension statements.

Matrix Inverse

Thus far, matrices appear to behave much like scalars, at least for addition, subtraction, and multiplication operations. (The notable exception is that matrix multiplication does not commute whereas scalar multiplication is commutative.) There is, however, no division operation for matrices, so the analogy fails at this point. For many matrices there is a second matrix said to be the *inverse* of the first, but this is not equivalent to a scalar reciprocal.

Assume that $[A]$ is a square matrix having an inverse (more will be said later about when an inverse exists). *If the matrix product* $[A]\,[V]$ *results in the identity matrix, then* $[V]$ *is called the inverse of* $[A]$. The inverse of $[A]$ is often written as $[A]^{-1}$, where the superscript -1 suggests the exponent -1 denoting a reciprocal in scalar arithmetic. While some would prefer to distinguish between a left inverse satisfying $[A]^{-1}\,[A] = [I]$ and a right inverse satisfying $[A]\,[A]^{-1} = [I]$, as a practical matter and for computational purposes, they are the same. Thus,

$$[A]\,[A]^{-1} = [A]^{-1}\,[A] = [I]$$

The analogy with scalar arithmetic is obvious, but note again that there is no division operation defined for matrices, only multiplication. This is true, even when one of the factors in the product is described as an "inverse."

Inverse matrices exist only for square matrices, but not every square matrix has an inverse. A matrix for which there is no inverse is called *singular*, meaning exceptional or out of the ordinary. A later section will deal further with the question of when an inverse exists and what failure to have an inverse indicates about the matrix. While the description above provides a means by which a possible inverse matrix can be tested, it does not really provide a means for determining the inverse if it exists.