Computer System Support for Scientific and Engineering Computation

Lecture 18 - June 30, 1988 (notes revised July 27, 1988)

Copyright ©1988 by W. Kahan and David Goldberg. All rights reserved.

1 Twenty-Five Years with Mathematical Software

This is a summary of the lecture by Cleve Moler, currently of Ardent Computers, about writing portable software for solving linear algebra problems.

1.1 Machine Epsilon

Machine epsilon (hereafter referred to as eps) is defined to be the separation between 1 and the next floating point number, and is often approximated as the smallest floating point number ε so that $1 + \varepsilon > 1$. The earliest program that Moler presented, a linear equation solver from 1963, required the user to specify eps as an input parameter. That program also used assembly language routines ILOG2, DOT, SDOT, and DAD, as well as using fixed size arrays. The programs in *Computer Solution of Linear Algebraic Systems* by Forsythe and Moler (1967) buried eps in the code, setting it equal to 0-0, which the user had to replace with the correct value for his machine. Moler gave an anecdote concerning a program that went into an infinite loop on the Ardent machine. The reason was it used a routine from Bell Labs called rimach, which requires the porter to find the parameters for his machine in the comments, and then "uncomment" that piece of code. The code had come from being run on a Pyramid machine, and it miraculously had run correctly even though the constants in the code were for neither the Pyramid nor the Ardent computers! The translation of *Computer Solution of Linear Algebraic Systems* into Hungarian replaced 0-0 with 1.0E-8.

The volume on linear algebra of Handbook for Automatic Computation by Wilkinson and Reinsch appeared in 1971.¹ The routines in this book not only required giving eps as an argument to procedures, but also a variable tol which was used to guard against underflow. The reason an underflow check was needed, can be seen as follows. Suppose you wanted to scale a vector (a, b, c) to have norm 1. You would compute the norm $s = \sqrt{a^2 + b^2 + c^2}$ and replace the vector with (a/s, b/s, c/s). But suppose that the vector is $10^{-19}(1, 2, 1)$. In IEEE single precision, the underflow threshold is 1.2×10^{-38} . If underflows are flushed to zero, then the computed value of the norm s will be 2×10^{-19} and the normalized vector will be $(\frac{1}{2}, 1, \frac{1}{2})$, which doesn't have norm 1. This situation can be avoided by explicitly checking for underflow using the paramater to1.

¹Wilkinson's earlier book has been referred to as the Bible, this volume as the New Testament.



The book Computer Methods for Mathematical Computations by Forsythe, Malcolm and

Moler (1977) gives the following machine independent code for computing eps

EPS = 1. 10 EPS = 0.5*EPS EPSP1 = EPS + 1 IF (EPSP1 .GT. 1.) GO TO 10

This code appears to compute EPS = $\frac{1}{2}$ ulp(1) on IEEE machines and IBM 370 and EPS = $\frac{1}{4}$ ulp(1) on VAX machines. Beyond that, it may not work correctly for machines with a high precision accumulator, because it might compute EPSP1 in a high precision accumulator, and compare this high precision number with 1, rather than rounding EPSP1 to the precision used to store floating point numbers in memory. Or even worse, an optimizing compiler might change the test EPSP1 .GT. 1. to EPS .GT. 0, which would compute the smallest positive representable number rather then eps. A routine that uses this algorithm for eps is the zero finding program ZEROIN. It requires the user to provide an argument TOL for the amount of error that can be tolerated in the answer. If TOL is zero, than the result is computed to within eps.

One of the tricks in ZEROIN concerns finding the midpoint between B and C. The naive formula (B + C)/2.0 may not work on a non-binary machine. For example in two digit decimal, if B = 9.7 and C = 9.8 then B + C is 19.5 and will be rounded to either 19 or 20, thus (B + C)/2.0 will be either 9.5 or 10, neither of which is between 9.7 and 9.8. The routine ZEROIN uses the formula B + (C-B)/2.0 instead.²

Not all the routines in Computer Methods for Mathematical Computations compute EPS directly. For example, the singular value decompositon program SVD adds ABS(SMALL) to ANORM, where SMALL is a small quantity computed in the algorithm. When the sum equals ANORM, the iteration stops. As with eps, this calculation can be ruined by optimizing compilers, and a truly careful routine would be

```
COMMON FTEST
TEST = ABS(SMALL) + ANORM
CALL FOO(TEST)
IF (FTEST .EQ. ANORM)
...
SUBROUTINE FOO
COMMON FTEST
FTEST = TEST
```

In 1974, EISPACK appeared, which was basically the translation of Wilkinson and Riesch into FORTRAN. The only unportability in EISPACK concerned eps. It was defined by MACHEP = ?, so that the programs wouldn't compile unless ? was replaced with a value. The variable tol was eliminated by scaling the vector (a, b, c) before taking its norm.

In EISPACK III (1983), code for computing eps was provided, namely

	A	=	4.	.01)0,	/3	. OD	0
10	В	*	A	-	1.	. 01	00	
	С	=	В	+	В	+	В	

²This trick is due to Householder and dates from around 1953.

(

J

£

Lecture 18 - June 30, 1988 (notes revised July 27, 1988)

EPS = DABS(C-1.0D0) IF (EPS .EQ. 0.0D)) G0 T0 10

The GO TO 10 is inserted in order to foil optimizing compilers. The reason why this program works, is that 4.0D0/3.0D0 can only be represented exactly on ternary machines, or in other words can't be represented exactly on any known machine, and so will be rounded. This is the only rounding error that occurs in this program, and so C will be slightly different from 1. The subtraction B = A - 1.0D0 guarantees that the last bit of B will be zero, and thus the last bit of C is zero. To illustrate, consider p = 5 and base $\beta = 10$. Then

A = 1.3333 B = .33330 C = .99990 EPS = .00010

The only roundoff error occured when computing A, and .0001 is the distance between 1.0 and the next representable number 1.0001. However, EISPACK III doesn't really use eps directly. Rather it tests for a negligible elements directly as we illustrated above for the routine SVD in *Computer Methods for Mathematical Computations*.

1.2 Iterative Refinement

We earlier studied how to use iterative refinement to improve the accuracy of solutions to linear systems. When doing iterative refinement, it is essential to compute the residual $\vec{b} - A\vec{x}$ in a higher precision than the main calculation. The earliest linear equation solver from 1963 used the assembly coded routines DOT and DAD to compute in double precision. The book *Computer Solution of Linear Algebraic Systems* has its algorithms written in ALGOL, and points out that accumulating sums in double precision can't be written in ALGOL 60. It refers to the routine innerprod, giving a reference for it. However, the FORTRAN version of the algorithm used the fact that FORTRAN compilers could recognize D = D + X*Y and compute the product in double precision. The PL/I version used the statement MULTIPLY(A(I,J), X(J), 12) to accumulate, using the fact that the default precision was 6.

But neither Computer Methods for Mathematical Computations (1976) nor LINPACK (1979) use iterative refinement. Some of the reasons are

- Its hard to write iterative refinement portably in FORTRAN, because when the working precision is double precision (as it usually is when doing scientific computing on all contemporary machines except CRAY and CDC), there is no portable way to efficiently code the extended precision operations.
- The extra accuracy you get using iterative refinement is not usually worth it, because the input data is usually not precise. In fact, the input matrix is often the output of another program.
- One of the uses of iterative refinement is to give a bound on the accuracy of the solution, but this information can be easily obtained by estimating the condition number of the matrix.

1.3 Efficiency

The 1967 book Computer Solution of Linear Algebraic Systems has the ALGOL comment

comment Inner loop. Only column subscript varies. Use machine code if necessary for efficiency.

Ideally, portable software shouldn't require writing in machine code, but rather should be written in such a way that compilers can optimize the code. In the first version of EISPACK, the inner loop of TRED1, the routine for reducing a lower triangular matrix to a tridiagonal one, looks like this

D0 180 K = 1, J 180 G = G + A(J,K) * A(I,K) JP1 = J + 1 IF (L .LT. JP1) G0 TO 220 D0 200 K = JP1, L 200 G = G + A(K,J) * A(I,K)

It contains an IF statement so it can't be vectorized on Cray class machines. In EISPACK III, the loop was rewritten as

```
D0 240 J = 1, L

F = D(J)

G = E(J) + A(J,J) * F

JP1 = J + 1

IF (L .LT. JP1) G0 T0 220

D0 200 K = JP1, L

G + G + A(K,J) * D(K)

-E(K) = E(K) + A(K,J) * F

200 CONTINUE

220 E(J) = G

240 CONTINUE
```

The code has been changed so that the inner loop does not have an IF statement.

The LINPACK codes address efficiency by using the BLAS, the Basic Linear Algebra Subprograms designed by Lawson, Hanson, Kincaid and Krogh (1978). Almost all the inner loops of LINPACK occur inside a BLAS routine, and LINPACK only uses column oriented BLAS. A typical BLAS routine is SAXPY, which performs the operation $\tilde{y} = \tilde{y} + a\tilde{x}$. The BLAS are written in FORTRAN but can be replaced by assembly language coded versions for machines with compilers that can't optimize them. Another advantage of using the BLAS is that for those who are familiar with it, codes which use it are easier to understand.³ Unfortunately, the BLAS tend to get in the way of very high quality compilers and can actually reduce efficiency.

In order to improve portability, LINPACK contains no machine dependent constants, no I/O, no character manipulation, no COMMON or EQUIVALENCE statements, and no mixed-mode arithmetic.



30D

(

(

Ŀ

ł

(

(

³And in fact, familiarity with BLAS is so widespread, that a company was named after SAXPY.

1.4 Comments on EISPACK

EISPACK and LINPACK cost over a million dollars to develop, and are very high quality codes. As mentioned above, EISPACK is now available as EISPACK III, which has improved in portability (no longer have to modify the code to insert your own MACHEP) and in performance (inner loops are column oriented and vectorizable). However, even in codes which are so highly developed, problems can be discovered. For example Guenter Ziegler and Andrew Odlyzko used the routine RS in EISPACK to compute the eigenvalues of the following real symmetric matrix :

(-1	1	1	-1	-1	1	1	-1	-1 \
1	1	-1	-1	1	1	-1	-1	1
1	-1	-1	1	1	-1	-1	1	1
-1	-1	1	1	-1	-1	1	1	-1
-1	1	1	-1	-1	1	1	-1	-1
1	1	-1	-1	1	1	-1	-1	1
1	-1	-1	1	1	-1	-1	1	-1
-1	-1	1	1	-1	-1	1	-1	1
-1	1	1	-1	-1	1	-1	1	1 /

EISPACK reported that 5 eigenvalues were on the order of roundoff error using double precision D format on the VAX. The correct answer is that 4 eigenvalues are 0. EISPACK got the wrong answer due to an underflow bug.

1.5 Pythagorean Sums

The expression $\sqrt{a^2 + b^2}$ occurs quite frequently. It represents the length of a vector and the norm of a complex number among other things. The obvious formula has two potential shortcomings. The first is that it requires a square root function to be available. The second is that it can underflow or overflow. In IEEE single precision, the maximum representable number is about 1.7×10^{38} , so if a or b is much bigger than 1.3×10^{19} , the computation of $\sqrt{a^2 + b^2}$ will overflow, even though the final answer is well within range. The paper *Replacing Square Roots by Pythagorean Sums* by Moler and Morrison gives an iterative algorithm that avoids these problems. If $a \ge b$, it starts by setting p = a and q = b. At each step, $p^2 + q^2 = a^2 + b^2$, but q gets smaller and hence p gets bigger. When q is negligible, then p will be an extremely good approximation to $\sqrt{a^2 + b^2}$. The rule for computing p and q is

$$r = \left(\frac{q}{p}\right)^{2}$$

$$s = \frac{r}{4+r}$$

$$p \leftarrow p + 2p\left(\frac{r}{4+r}\right) = p + 2ps$$

$$q \leftarrow q\left(\frac{r}{4+r}\right) = qs$$

Since r < 1, clearly $s < \frac{1}{3}$, so at each iteration q is at most $\frac{1}{5}$ th the value of the previous iteration (in fact, it decreases even more rapidly). And $p^2 + q^2$ becomes $p^2(1+2s)^2 + (sq)^2 = p^2(4s+4s^2)+p^2+(s^2-1)q^2+q^2 = (s+1)\{4sp^2+(s-1)q^2\}+p^2+q^2$. Since $(s-1)(q/p)^2 = -4s$,

the quantity in braces is 0, so $p^2 + q^2$ is preserved. The algorithm obviously doesn't involve square roots, and since p starts with the value $\max(a, b)$ and grows at each step, it will not underflow nor overflow. The main problem with this algorithm is that it is too slow. It requires 2 divisions per iteration, but hardware implementations of square root take about the same amount of time as a single division step. A more practical algorithm would be $|a|\sqrt{1+(b/a)^2}$, where $a = \max(a, b)$.

One interesting facet of this algorithm is that it is noticeably more accurate on machines supporting gradual underflow. If μ is the smallest positive representable floating point number, and $a = 4\mu$, $b = 3\mu$, then $s \approx .1233$ so $q \approx .37\mu$ underflows. If underflows are flushed to zero, then the algorithm stops after 1 iteration, giving an answer of $p + 2sp \approx 4.986\mu$, compared with the correct answer of 5μ . A machine with gradual underflow would give a much more accurate result. A related situation occurs with the alternate formula $|a|\sqrt{1 + (b/a)^2}$. This is does not benefit from gradual underflow, but it can have an error as large as $1 + 3\beta/4$ ulps on machines that use base β . A more accurate formula is $a+b/((a/b) + \sqrt{1 + (a/b)^2})$, and this modified formula does benefit from gradual underflow when a and b are small.

1.6 Comments on IEEE 754

What is the impact of IEEE 754 on writers of portable software?

- The most common languages for portable software, FORTRAN and C, don't have any language facilities that allow you to exploit the IEEE standard, particularly in the area of exception handling. The development of libraries such as Apple's SANE package may help in the future, although SANE on a Macintosh is quite slow.
- Portable software must work on VAX, Cray and IBM/370 as well as IEEE machines, so portable software can't assume that IEEE facilities will be available.
- The ANSI C and FORTRAN 8x efforts are more important to portable software than the IEEE standard.

A demo of MATLAB indicates how well this particular portable software deals with the IEEE standard. It correctly computes u = 0/0 as Nan and $\max(5, u) =$ Nan, but incorrectly sets $\max(u, 5) = 5$. The impact of IEEE on mini-supercomputer companies like Ardent is

- Gradual underflow is too slow for vector processors.⁴ Both the Weitek chips and all their imitators require extra cycles for processing denormalized numbers, but vector processors require predictable computation times.⁵
- The IEEE standard contains many fine points that are too much trouble to implement. An example of such a fine point is that when square root is implemented in software, that software must correctly set the inexact bit.

£

⁴Gradual underflow was the most controversial part of the standard, and probably accounted for the length of time it took to get adopted. A foreign visitor to the U.S. was advised that the sights not to be missed were Las Vegas, the Grand Canyon, and the IEEE standards committee meeting.

³Kahan suggests that the IEEE standard didn't address vector processors, because CRAY appeared to have a lock on the market, and wasn't interested in changing its arithmetic to conform to the IEEE standard. Hough: earlier drafts did address pipelined implementations via warning mode; at the instigation of some Apple people that was taken out to simplify the standard, subsequently complicating everybody's life to such an extent that Hough regrets supporting that simplication.

25 YEARS

WITH

MATHEMATICAL SOFTWARE

-Cleve Moler Ardent Computer June 30, 1988

•

• • •

1963

•

•

<u>C</u>			-	'- r	<u>י</u> רר		' TRAI				- r			
-		ו ו הוההחתי	ا 		1 1 	<u>A.</u>	N. C.		V. V. 11	<u>.</u>				
12	ઝ	JERUU I	1112	SULY	E(181) 7	SHS DS		39 j i riA 7 - 7	N N I I I	7	,			
•		•	-		•	•••		• •		•				
2	 EP	SAND	IТМ	AX A	RF 'P(RAMET	ERS .	N THE	ITEDAT	: אהוז	•		•	
				5 1	5	5 ï 5	3	i 19		·•				
	=	-	•	-			•		-					
С	CA	LLSF	AP S	וסצפח	JT I HE	ES ILO	1 <u>52</u> , 1	DOT, SI	dika too	פאם כ		•		
ŗ.		ï	1	ñ	1	17			• • •	1 1 .5				
		• • ••	-	•••				-						
- • • • •			NU A	ren. e	ດີໄອ	(50).	47501		ນ. ຮດນ. ກ	W(503.	D/501	• •		
	רה רה	ະມະນາ 3 1	71 FR 77				<u> </u>	THHUU		N(J0)3	RCJU	,		
			•							i 12				
	• -		•				a		• ··		-			
	• -	·			• •				• • • ··		-			
				. e 📜 (
0000 1234 1111	0080 3 6 7 6 1 1 1 1	880 8 • • • • • • • •		. C [] () 0 0 0 20223	00 2222 11111	- 							3880 88 08 572222377 1111111
6 8 6 8 1 2 3 4 1 1 1 1 7 7 7 7	3000 3070 1111 7777	880 8 ******		. C) 0 0 20 x 2 x 1 1 1 1 1	00 23333 11111 77777	8'8 2325 11111 77777			••••••••••••••••••••••••••••••••••••••				3 8 8 0 0 8 8 0 8 n 2 2 3 3 3 7 1 1 1 1 1 1 1 1 7 7 7 7 7 7 7 7 7
6 8 6 8 1 2 3 4 1 1 1 1 2 2 2 2 2	3 8 8 8 3 8 7 8 1 1 1 1 2 2 2 2 2			.C	0 0 0 10 10 11 11111 12222	0 0 73334 11111 22222		888 87 1 1 1 1 1						388000000 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
6008 1111 2222 3333	30000 3070 1111 2222 3333			0 () (0 0 10 0 0 0 0 0 0 0 1 1 1 1 2 2 2 2 2 3 3 3 _ 1) 0 0 20 x 2 x 2 x 2 x 2 x 2 x 2 x 2 x 2 x 2 x	0 0 73333 11111 22222 3333	8 -				- 0 0 8 11111 22222 33333			3 8 8 0 8 8 0 8 9 7 7 7 9 9 7 7 7 7 1 1 1 1 1 1 1 1 2 7 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3
6 0 0 8 1 1 1 1 1 2 2 2 2 2 3 3 3 3 4 4 4 4	3000 3171 1111 2222 3333 44 <u>4</u> 4	880 8 11111 222 2 33333 4444	2 2 2 2 2 2 3 3 3 3 3 4 4 4 4	. C [] C 	0 0 0 0 11111 2222 133 3	0 0 73777 11111 22222 3333 44 4		• • • • • • • • • • • • • • • • • • •		111111 222222 33333	- 11111 22222 33333 4.44			3 8 8 0 8 8 0 8 5 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7
6 0 0 8 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5	2222 3333 44:4 5555	000 0 111111 222 2 33333 4444 5 55	0000 111 22222 3333 4444 5 5 5 5	0 () 0 () 1 1 1 1 2 2 2 2 2 3 3 3 _ 3 _ 4 4 4 4 5 _ 5 5) 0 0 2 2 2 2 2 1 3 3 3 1 4 4 4 5 5 5 5	00 11111 22222 3333 444 5555		8888 999 1 111 222222 33333 444_44			- 0 0 8 11111 22222 33333 4:_44 55_55	_0000 111111 222222 _33333 44444 55555	0 0 0 0 0 0 1 1 1 1 1 1 2 2 2 2 2 2 1 3 3 3 3 3 1 4 4 4 4 4 5 5 5 5 5	3 8 8 0 8 8 0 8 1 2 7 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4
	3 8 8 8 8 1 1 1 1 1 2 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 5 5 5 5	000 0 11111 222 2 33333 4444 5 55	2222 3333 4444 5 55	0 () () 1 1 1 1 1 2 2 2 2 2 3 3 3 _ 1 4 4 4 4 5 _ 5 5) 0 0 0 22 2 2 2 1 3 3 3 1 4 4 4 5 5 5 5	8 8 1 1 1 1 1 1 2 2 2 2 2 3 3 3 3 4 4 4 5 5 5 5		0 0 0 0 1 111 2 2 2 2 2 2 2 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4		111111 222222 33333 4 44 4 55 555		00000 111111 222222 33333 44444 55555		
0 0 0 0 1 7 3 1 1 1 1 1 2 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 5 6 6 6	0000 1111 2222 3333 44:4 5555 5655	00000 111111 22222 33333 4444 5555 66666	0000 111 22222 33333 6444 5_55 .6666	. 0 0 	0 0 0 2 2 2 2 1 1 3 3 3 1 4 4 4 5 5 5 5 6 6 6 6 8	0 0 2 2 2 2 2 2 3 3 3 3 4 4 4 5 5 5 5 5 6 6 6 6 6		00 0 1 111 222222 33333 444_44 5555 666666		111111 222222 33333 4 44 4 55 555		00000 111111 222222 33333 44444 55555 66666		
6 0 0 0 1 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4 5 5 5 5 5 6 6 6 7 7 7 7	3 8 8 8 8 3 7 7 1 1 1 1 1 2 2 2 2 3 3 3 3 4 4 . 4 5 5 5 5 5 6 6 6 7 7 7 7	00000 111111 22222 33333 4444 5555 66666 77777	2 2 2 2 2 2 3 3 3 3 3 5 4 4 4 4 5 _ 5 5 5 4 6 6 6 6 7 7 7 7 7	. 6 6 	0 0 0 11111 2222 133 3 14 44 5555 5555 5555	8 8 7 7 7 7 7		8 8 1 11 2 2 3 3 4 4 5 5 6 6 7 7				00000 111111 222222 33333 44444 55555 55555 55555 55555 55555 55555		3 8 8 0 0 8 8 0 8 1 1 1 1 1 1 1 1 1 2 7 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5 5 5 6 6 6 6 6 6 6 6 7 7 7 7 7 7 7 7
600 1111 2222 3333 4444 5555 8666 7777 8888	2222 3333 44.4 5555 6666 7777 8888	000 0 11111 222 2 33333 4444 5 55 5 55 5 6 6 6 6 7 7 7 7 7 8 8 8 8 8	2 2 2 2 2 2 3 3 3 3 3 3 4 4 4 4 5 5 5 5 5 5 5 5 5 5	6 1 6 1 1 1 1 1 2 2 2 2 2 3 3 3 _ 3 4 4 4 4 5 _ 5 5 5 6 6 6 6 7 7 7 7 1 8 8 [1	0 0 0 0 0 1 1 1 1 2 2 2 2 1 3 3 3 3 4 4 4 5 5 5 5 6 6 6 6 7 7 7 7 8 8 1 4			0 0 0 1 1 1 2 2 2 3 3 3 4 4 4 5 5 5 6 6 6 7 7 7				00000 111111 222222 33333 44444 55555 66666 77777		

•

.

•

•

•

.

.

.

• *

1967

COMPUTER SOLUTION OF LINEAR ALGEBRAIC SYSTEMS

GEORGE E. FORSYTHE

Professor of Computer Science Stanford University

ſ

ſ

1

.....

à

1

; (____

.

: C.

CLEVE B. MOLER

Assistant Professor of Mathematics University of Michigan

PRENTICE-HALL, INC.

:•• ·

. .

ENGLEWOOD CLIFFS, N.J.

المربوبة المتعادين والمراجع

16. ALGOL 60 PROGRAM

Computer programs that use Gaussian elimination or one of its variants have been written in many programming languages and used on many computers. Several of these programs have also used some form of iterative improvement. Together with William McKeeman we have developed the set of four ALGOL 60 procedures now to be given as program (16.1). Earlier versions of this program are found in Forsythe (1960) and McKeeman (1962). See Baumann *et al.* (1964) and Naur *et al.* (1963) for an introduction to and a definition of the ALGOL 60 language. Several pages of explanation follow our program.

(16.1) ALGOL 60 program for solving linear systems

begin
comment Linear system package, ALGOL 60 version;
integer array $ps[1:100]$; comment Global pivot index array. We assume $n \le 100$;
<pre>procedure DECOMPOSE(n, A, LU); value n; integer n;</pre>
real array A, LU; comment A, LU[1:n, 1:n]; comment Uses global integer array ps;
comment Computes triangular matrices L and U and per- mutation matrix P so that $LU = PA$. Stores $L - I$ and U in LU. Array ps contains permuted row indices:
comment DECOMPOSE(n. A. A) overwrites A with LU:
begin
real array scales[1:n];
integer i, j, k, pivotindex;
real normrow, pivot, size, biggest, mult;
comment Initialize ps, LU and scales;
for $i := 1$ step 1 until n do
begin
ps[i] := i;
normrow :== 0;
for $j := 1$ step 1 until n do
begin
LU[i,j] := A[i,j];
<pre>if normrow < abs(LU[i, j]) then normrow := abs(LU[i, j]); end;</pre>

ł

58

•



SEC. 16

. . .

.

ALGOL 60 PROGRAM 59

if normrow $\neq 0$ then scales[i] := 1/normrowelse begin scales[i] := 0; SINGULAR(0) end; end; comment Gaussian elimination with partial pivoting; for k := 1 step 1 until n - 1 do begin biggest := 0;for i := k step 1 until π do begin size := $abs(LU[ps[i], k]) \times scales[ps[i]];$ if biggest < size then begin biggest := size; pivotindex := i end; end: if biggest = 0 then begin SINGULAR(1); go to endkloop end; if pivotindex $\neq k$ then begin j := ps[k]; ps[k] := ps[pivotindex]; ps[pivotindex] := j end: pirot := LU[ps[k], k];for i := k + 1 step 1 until n do begin LU[ps[i], k] := mult := LU[ps[i], k]/pivot;if $mult \neq 0$ then lor j := k + 1 step 1 until n do $LU[ps[i], j] := LU[ps[i], j] - mult \times LU[ps[k], j];$ comment Inner loop. Only column subscript varies. Use machine code if necessary for efficiency; end; endkloop: end; if LU[ps[n], n] = 0 then SINGULAR(1); end DECOMPOSE: procedure SOLVE(n, LU, b, x); value n; integer n; real array LU, b, x; comment LU[1:n, 1:n], b, x[1:n]; comment Uses global integer array ps; comment Solves Ax = b using LU from DECOMPOSE; begin integer i, J; real dot;

```
for i := 1 step 1 until n do
 begin
    dot := 0:
    for j := 1 step 1 until i - 1 do
      dot := dot + LU[ps[i], j] \times x[j];
    x[i] := b[ps[i]] - dot;
  end;
 for i := n step -1 until 1 do
  begin
    dot := 0;
    for j := i + 1 step 1 until \pi do
      dot := dot + LU[ps[i], j] \times x[j];
    x[i] := (x[i] - dot)/LU[ps[i], i];
  end:
  comment As in DECOMPOSE, the inner loops involve only the
           column subscript of LU and may be machine coded
           for efficiency;
end SOLVE:
```

SEC. 16



.

.

61 ALGOL 60 PROGRAM SEC. 16 The body of the procedure cannot be written in ALGOL 60: comment The body of accumdotprod could be written as follows in terms of the code procedure innerprod on p. 206 of Martin, Peters, and Wilkinson (1966): begin real d1, d2 integer k merprod(1, 1, n, extraterm, 0, A[i, k], x[k], k, d1, d2)accumdotproa := a1 end; begin comment (code); accumdotprod := 0-0; comment 0-0 indicates code result; end accumdatorod : eps := 0-0; comment Machine-dependent round-off level; itmax := 0-0; comment Use approximately $2 \times \log(1/eps)$; normx := 0,for i := 1 step 1 until π do if norm x < abs(x[i]) then norm x := abs(x[i]); if normx = 0 then begin digits := -log(eps); go to converged end; for iter := 1 step 1 until itmax do begin for i := 1 step 1 until n do r[i] := -accumdotprod(n, A, i, x, -b[i]);SOLVE(n, LU, r, dx);normdx := 0; for i := 1 step 1 until n do begin t := x[i]; $\mathbf{x}[i] := \mathbf{x}[i] + d\mathbf{x}[i];$ if norm dx < abs(x[i] - t) then norm dx := abs(x[i] - t); end; if iter = 1 then digits := $-\log(\text{if normdx} \neq 0 \text{ then normdx}/normx)$ else eps); if norm $dx \leq eps \times norm x$ then go to converged; end iter: comment Iteration did not converge; SINGULAR(2); converged: end IMPROVE:

.

.

:

:

ŧ

62 ALGOL 60 PROGRAM



Notes on the ALGOL program: DECOMPOSE (n, A, LU) uses elimination to find *n*-by-*n* triangular matrices L and U so that LU = PA, where PA is the matrix A with its rows interchanged. The interchange information is stored in the global array ps, and the matrices L - I and U are stored in LU.

SOLVE (n, LU, b, x) uses the LU factorization from DECOMPOSE to find an approximate solution to a single system of equations, Ax = b.

IMPROVE (n, A, LU, b, x, digits) requires a copy of the original matrix A, its LU decomposition, a right-hand side b, and the approximate solution x computed by SOLVE. It carries out the iterative improvement process until, if possible, x is accurate to machine precision. It also provides an estimate digits of the accuracy of the first approximation. The value of digits is, roughly, the number of decimal digits of x which are not changed by the iteration. This is a measure of the condition of A.

SINGULAR (why) is used by the other procedures to indicate the occurrence of an error condition.

In practice, these procedures are used by another procedure or executive program written to handle a specific class of problems. As an example, we have included in Sec. 18 a procedure which inverts a matrix.

DECOMPOSE uses elimination, basically in the form described in Sec. 9. Temporarily ignoring scaling and pivoting, we can express the central calculation, the elimination, by

(16.2)

for j := k + 1 step 1 until n do

$$a_{i,j} := a_{i,j} - (a_{i,k}/a_{k,k}) \times a_{k,j}.$$

SEC. 16

17. FORTRAN, KITERJESZTETT ALGOL ÉS PL/1 PROGRAMOK

Az előző fejezetben leírt eljárások legtöbb részlete közvetlenül lefordítható más algoritmikus számítógépnyelvre. Ezt tesszük most a FORTRAN egy elfogadott standardizálásával, egy kiterjesztett ALGOL konkrét realizálásával és a PL/1 egy előzetes specifikációjával kapcsolatban. Mindegyik program illusztrálja maguknak az eljárásoknak bizonyos részeit, valamint a felhasznált nyelveket és számítógépeket. (Javasoljuk az Olvasónak, hogy tájékozódjon azokról a nyelvekről és számítógépekről, amelyekkel nem ismerős.)

Úgy gondoljuk, hogy az általunk használt FORTRAN nyelv megfelel az American Standard Association (1964) által leírt, legtöbbször ASA FORTRAN-nak nevezett nyelvnek. Amennyire ez lehetséges, magában foglalja három FORTRAN dialektus: az IBM 7090/94-re készített FORTRAN IV, a CDC 1604-re készített FORTRAN 63, és az IBM System/360-ra készített Basic Programming Support FORTRAN közös vonásait (lásd International Business Machines (1965a), Control Data Corp. (1963), és International Business Machines (1965b)). Elkerültük az olyan vonásokat, mint a típusdeklarációk, relációs kifejezések, címkés közös tárolás és változtatható tömbdimenziók, amelyek hasznosak lehetnének, de egyúttal különböző formájúak is, és egy vagy több rendszerben nem is léteznek. Néhány kisebb összeegyeztethetetlenség előfordul: a JAVÍT-ban a kettős pontosság deklarációjának a formája, a FEL-BON és JAVÍT-ban az ABS, AMAX1 és ALOG10 függvénynevek, valamint az output egységszár a KIIR-ban. E pontokon esetleges változtatásokat eszközölve, a szubrutinok más FORTRAN rendszerekre is átvihetők.

(17.1) FORTRAN program lineáris egyenletrendszerek megoldására



5 Lineáris algebrai problémák

.

С	•
č	GAUSS FÉLE KIKÚSZÖBÖLÉS RÉSZLEGES FŐELEM-
č	KIVÁLASZTÁSSAL
•	NMI = N - I
	DO 17 K = 1 NM1
	NAGY=00
	MÉRET = ABS (III. (IP. K)) + SK ÅI ÅK (IP)
	IE (MÉRET-NAGY) 11 11 10
	$10 \qquad NAGY = MÉRET$
	IDXFOR-I
	$17 \qquad (1401) 13, 14, 13$
	$\frac{12}{CO} = \frac{17}{17}$
	$12 IE (IDYEOE-K) \ 14 \ 15 \ 14$
	$13 \qquad 1 \qquad$
	IPS(K) - IPS(IDYEOE)
	$\frac{1}{1} \frac{1}{1} \frac{1}$
	IS (IDAFOE) = J
	$\frac{1}{1} = \frac{1}{1} $
	FOELEM = OL(RF, R)
	$\mathbf{A}\mathbf{F}\mathbf{I}=\mathbf{A}+\mathbf{I}$
	$\frac{1}{10} \frac{1}{10} \frac$
	IF = IFS(I)
	= -OL(IP, K)/FOELEM
	OL(IP, K) = -EM
	DU = I I I I I I I I I I I I I I I I I I
~	
C	BELSU CIKLUS. HASZNALJUNK GEPI KUDUI
C	HA A COMPILER NEM AD HATEKUNT PROG-
C	KAMUI.
	16 CONTINUE
	I/ CONTINUE
	KP = IPS(N)
	IP(UL(KP, N)) 19, 18, 19
	18 CALL KIIK (2)
	IY KETUKN
	End

SUBROUTINE MEGOLD (NN, UL, B, X) DIMENSION UL (30, 30), B (30), X (30), IPS (30) COMMON IPS N=NNNP1=N+1С IP=IPS (1) X (1)=B (IP)

66 .

.....

•

:

i

!

ŧ



(:-

• . •

67

С	• • • LÉNYEGES HOGY A (I, J) • X (J) KETTŐS PON-
С	TOSSÁGÚ EREDMÉNYT ADJON ÉS A FENTI
Č	+ ÉS – KETTŐS PONTOSSÁGÚ LEGYEN. • • •
•	· CALL MEGOLD (N. UL. R. DX)
	DXNORM = 0.0
	DO 6I=1 N
	$\mathbf{T} = \mathbf{Y}(\mathbf{I})$
	$\mathbf{Y} = \mathbf{X} (\mathbf{i})$
	$\Delta (I) = \Delta (I) + D \Delta (I)$ $D Y = D Y = A M A Y I (D Y = D M A P C (Y (I) = T))$
	D X N O K M = A M A X I (D X N O K M, A D S (X (I) - I))
	6 CONTINUE
	IF (IIER-I) 8, 7, 8
	7 JEGYEK = - ALOGIO (AMAXI (DXNORM/
	(XNORM, EPS))
	8 IF (DXNORM-EPS • XNORM) 10, 10, 9
•	9 CONTINUE
С	AZ ITERÁCIÓ NEM KONVERGÁLT
	CALL KIIR (3)
	10 RETURN
	END
	· · · · · · · · · · · · · · · · · · ·

۱



Figyeljük meg, hogy az ALGOL programbeli LU-t a FORTRAN programokban UL-el fejeztük ki.

Előfordulhat, hogy egyetlen ALGOL utasítást — különösen indexeket, valamint logikai vagy Boole-kifejezéseket tartalmazót — csak több FORTRAN utasítással tudunk kifejezni. Viszont a fordítóprogram bizonyos értelemben kárpótolhat ezért, batékonyabb gépi kód előállításával. Esetünkben ez különösen így van. A FORTRANban a FELBON-beli belső ciklus

$$(17.2)$$
 DO 16 J = KP1, N

16 UL (IP, J) = UL (IP, J) + $EM \bullet UL$ (KP, J)

76 FORTRAN, EXTENDED ALGOL, AND PL/I PROGRAMS

: C

.

••••

....

IMPROVE: PROCEDURE (N. A. LU. B. X. DIGITS) : DECLARE A(+,+) /+ URIGINAL MATRIX +/, LU(+,+) FLNAT /+ DFCOMPOSITION OF A +/, B(+) /+ RIGHT MAND SIDE +/, X(+) /+ RIGHT MAND SIDE +/, DIGITS /+ WILL BE SET TO ACCURACY OF IMPUT X +/ 1 DECLARE (R.OX) (N), (NURMX, MORMDX, T) FLOAT, (I, J, ITER) FIXED BINARY, EPS INITIAL (1.8-6) /* MACHINE DEPENDENT ROUMDOFF LEVEL */, ITMAX INITIAL (12) /* USE 2*LOG10(1/EPS) APPROXIMATELY */; DECLARE OPSUM FLOAT (12) /* IT IS SSENTIAL "MAT PRECISION OF DPSUM AND ARGUMENT OF MULTIPLY Used below be twice default precision. Default precision of 6 Assumed Here. */ ; NGRMX = 0 ; 00 [= 1 TO N ; NGRMX = MAX(NORMX, A65(X([)]) ; FND : IF NORMX = 0 THEN 00; DIGITS = -LOGIO(EPS); GO TO CONVERGED; END ; DO ITER = 1 TO ITMAX : DO I = 1 TO N : DPSUM = 0 : DPSUM = DPSUM + MULTIPLY(A(I,J), X(J), 12) : END : DPSUM = B(11 - DPSUM ; R(I) = DPSUH : END : CALL SOLVE(N+LU+R+DX1 ; NORMOX = 0 : DO I = 1 TO N ; T = X(1) : x(1) = x(1) + Dx(1) ; NORHOX - MAX (NORMOX, A65(X(I)-T)) : END : IF ITER = 1 THEN DIGITS = -LOGIOIMAXINORMOX/NORMX.EPSII ; IF NORMOX <= EPS+NURMX THEN GO TO CUNVERGED : END : CALL SINGULAR('CON') ; CONVERGED: END IMPROVE ;

SINGULAR: PROCEDURE (WHY) : DEGLARE WHY CHARACTER(3) : IF WHY-ROW' THEN PUT SKIP(2) LIST (*ZERO ROW IN DECOMPOSE') : IF WHY=*PIV* THEN PUT SKIP(2) LIST ('SINGULAR WATKIX IN DECOMPOSE. SOLVE WILL DIVIDE BY ZERO.'); IF HMY='COM' THEN PUT SKIPIZI LIST ('ND COMVERGENCE IN IMPROVE. MATRIX IS NEARLY SINGULAR.'); END SINGULAR;

•

- -

· • • •

SEC. 17

...

--

1976

COMPUTER METHODS FOR MATHEMATICAL COMPUTATIONS

GEORGE E. FORSYTHE

MICHAEL A. MALCOLM

Department of Computer Science University of Waterloo

316

••

CLEVE B. MOLER

Department of Mathematics and Statistics University of New Mexico

PRENTICE-HALL, INC.

ENGLEWOOD CLIFFS, N. J. 07632

. . . .

and $L \le e \le U$. If for every nonzero x in F, $d_1 \ne 0$, then the floating-point number system F is said to be normalized. The integer e is called the exponent, and the number $f = (d_1/\beta + \ldots + d_i/\beta^2)$ is called the fraction. Usually the integer $\beta' \cdot f$ is stored using a common integer representation scheme such as signed magnitude, one's complement, or two's complement.

Actual computer implementations of floating-point representations may differ in detail from the ideal ones discussed here, but the differences are minor and can almost always be ignored when dealing with fundamental problems of roundoff errors.

The following table gives some examples of floating point systems. The quantity β^{1-r} is an estimate of the relative accuracy of the arithmetic. We do not give the precise value of machine epsilon because it depends upon complicated details, such as the form of rounding.

If the number of digits, t, is not an integer, it means that $\beta = 2^k$ and $k \cdot t$ bits are available for binary representation of the fraction.

Computer	ß	t	L	U	β^{1-i}
Univac 1108	2	27	-128	127	1.49 × 10 ⁻¹
Honeywell 6000	2	27	-128	127	1.49 × 10⁻∎
PDP-11	2	24	-128	127	1.19 × 10-7
Control Data 6600	2	48	976	1,070	7.11 × 10 ⁻¹⁵
Cray-1	2	48	-16,384	8,191	7.11 × 10-13
Illiac-IV	2	48	-16,384	16,383	7.11 × 10-15
Setun (Russian)	3	18	?	?	7.74 × 10-*
Burroughs B5500	8	13	-51	77	1.46 × 10-11
Hewlett Packard HP-45	10	10	-98	100	1.00 × 10-9
Texas Instruments SR-5x	10	12	-98	100	1.00×10^{-11}
IBM 360 and 370	16	6	-64	63	9.54 × 10-7
IBM 360 and 370	16	14	-64	63	2.22 × 10-16
Telefunken TR440	16	91	-127	127	5.84 × 10-11
Maniac II	65536	2	-7	7	7.25 × 10-9

Some computers use more than one floating-point number system. For example, the IBM 360 uses the two base-16 systems listed above. These two different systems are called *short precision* and *long precision*.

The set F is not a continuum, or even an infinite set. It has exactly $2(\beta - 1)\beta^{t-1}(U - L + 1) + 1$ numbers in it. These are not equally spaced throughout their range but only between successive powers of β . Figure 2.1 shows the 33-point set F for the small illustrative system $\beta = 2, t = 3, L = -1, U = 2$.

Because F is a finite set, there is no possibility of representing the continuum of real numbers in any detail. Indeed, real numbers in absolute value

14 FLOATING-POINT COMPUTATION

318

CHAP. 2

value) of ϵ . That is, a program can discover the available precision for the machine it is executing on *at execution time*. The method we use for computing an approximation which differs from ϵ by at most a factor of 2 is illustrated by the following segment of a Fortran program:

EPS = 1. 10 EPS = 0.5•EPS EPSP1 = EPS + 1. IF (EPSP1 .GT. 1.) GO TO 10

.

. ÷.

P2-5. (Kahan) (a) How are the numbers ½, ¾, and ⅓ represented internally in your computer. Use an appropriate notation, i.e., binary, octal, hexadecimal, etc. How are these numbers represented in the floating-point number systems of other computers such as the IBM 360, CDC 6600, Univac 1108, Honeywell 6000, PDP-11, Burroughs 6500, etc.?

(b) Consider the following Fortran program:

H = 1./2. X = 2./3. - H Y = 3./5. - H E = (X+X+X) - H F = (Y+Y+Y+Y+Y) - H Q = F/EWRITE (6,10) Q STOP 10 FORMAT(1H, G20.10) END

The variable Q can take on several different values depending on the floating-point arithmetic hardware used by the computer. Try to figure out the value of Q for computers you are familiar with. Run the program on as many computers as you can to check your results. Explain your results.

P2-6. Consider the following two Fortran programs:

```
EPS = 1.

10 EPS = EPS/2.

WRITE (6,20) EPS

20 FORMAT.(1H, G20.10)

EPSP1 = EPS + 1

IF (EPSP1 .GT. 1.) GO TO 10

STOP

END

EPS = 1.

10 EPS = EPS/2.

WRITE (6,20) EPS

20 FORMAT (1H, G20.10)

IF (EPS .GT. 0.) GO TO 10

STOP
```

Run the programs on your system, and explain the results.

END

P2-7. What output is produced when the following Fortran program is run on various computers with which you are familiar? Try to predict the output before actually running the program; then run it to confirm your answer.

REAL FUNCTION ZEROIN(AX.BX.F.TOL) REAL AX, BX, F, TOL

A ZERO OF THE FUNCTION F(X) IS COMPUTED IN THE INTERVAL AX.BX .

- REAL C A ZI C A ZI C INPUT.. C AX I C AX I C BX I C F F C T LEFT ENDPOINT OF INITIAL INTERVAL RIGHT ENDPOINT OF INITIAL INTERVAL FUNCTION SUBPROGRAM WHICH EVALUATES F(X) FOR ANY X IN THE INTERVAL AX, AX DESIRED LENGTH OF THE INTERVAL OF UNCERTAINTY OF THE C C
 - TOL FINAL RESULT (.GE. 0.0)

C C OUTPUT. ¢

ē

С

C С

С

Ĉ

ZEROIN ABCISSA APPROXIMATING A ZERO OF F IN THE INTERVAL AX.8X Ċ

IT IS ASSUMED THAT F(AX) AND F(BX) HAVE OPPOSITE SIGNS WITHOUT A CHECK. ZEROIN RETURNS A ZERO X IN THE GIVEN INTERVAL AX.BX TO WITHIN A TOLERANCE 4°MACHEPS*ABS(X) + TOL, WHERE MACHEPS IS THE RELATIVE MACHINE PRECISION. Ĉ С ¢

THIS FUNCTION SUBPROGRAM IS A SLIGHTLY MODIFIED TRANSLATION OF THE ALGOL 60 PROCEDURE ZERO GIVEN IN RICHARD BRENT, ALGORITHMS FOR MINIMIZATION WITHOUT DERIVATIVES, PRENTICE - HALL, INC. (1973). ĊCC C

REAL A.B.C.D.E.EPS.FA.FB.FC.TOLI.XM.P.O.R.S

С С COMPUTE EPS, THE RELATIVE MACHINE PRECISION

EPS = 1.0 10 EPS = EPS/2.0 TOLI = 1.0 + EPS IF (TOLI .GT. 1.0) GO TO 10

CHAP. 7

•.•

• • •

1

(

C C INITIALIZATION

A = AX8 = 8X FA = F(A)FB = F(B) Г.

C

•

्र ल

• • • •

÷.

.

. . . .

۰.

C BEGIN STEP C 20 C = A FC = FA D = 8 · A E = D 30 IF (ABS(FC) .GE. ABS(FB)) GO TO 40 $\begin{array}{c} \mathbf{A} = \mathbf{B} \\ \mathbf{B} = \mathbf{C} \end{array}$ Č = Å FA = FB FB = FC FC = FA C C CONVERGENCE TEST С 40 TOLI = 2.0"EPS" ABS(B) + 0.3"TOL XM = .3"(C - B) IF (ABS(XM) .LE TOLI) GO TO 90 IF (FB .EQ. 0.0) GO TO 90 C C IS BISECTION NECESSARY č IF (ABS(E) .LT. TOLI) GO TO 70 IF (ABS(FA) .LE. ABS(FB)) GO TO 70 C IS QUADRATIC INTERPOLATION POSSIBLE IF (A .NE. C) GO TO 50 C LINEAR INTERPOLATION S = FB/FAP = 2.0"XM"S Q = 1.0 - S GO TO 60 C C INVERSE QUADRATIC INTERPOLATION С SO Q = FA/FC R = FB/FC S = FB/FA $P = S^{\circ}(2.0^{\circ}XM^{\circ}Q^{\circ}(Q - R) - (B - A)^{\circ}(R - 1.0))$ $Q = (Q - 1.0)^{\circ}(R - 1.0)^{\circ}(S - 1.0)$

. . .

•

166 SOLUTION OF NONLINEAR EQUATIONS

CHAP. 7

<u>si</u>

• • :-

• • •

÷

1

.

• • •

؛

÷ .

Ř sec. 5.5 SUBROLITINE QUANCS 05 C C C C LOCATE NEXT INTERVAL. 72 IF (NIM .EQ. 2*(NIM/2)) GO TO 75 NIM = NIM/2 LEV = LEV-1 GO TO 72 75 NIM = NIM + 1 IF (LEV .LE. 0) GO TO BO C ASSEMBLE ELEMENTS REQUIRED FOR THE NEXT INTERVAL. C C QPREV = QRIGHT(LEV) X0 = X(16)F0 = F(16) DO 78 1 = 1, 8 F(2*1) = FSAVE(I.LEV) X(2*1) = XSAVE(I.LEV) 78 CONTINUE GO TO 30 С с с *** STAGE & *** FINALIZE AND RETURN \$0 RESULT = RESULT + CORII С MAKE SURE ERREST NOT LESS THAN ROUNDOFF LEVEL. Ċ IF (ERREST .EQ. 0.0) RETURN 82 TEMP = ABS(RESULT) + ERREST IF (TEMP .NE. ABS(RESULT)) RETURN ERREST = 2.0°ERREST GO TO 82 00000 END

÷ С

ſ

Ì

1

1

į

1

<u>ا</u>، _____

-

. .

:..;

Ł

- ·.

· . . - '

. .

••

• •

325

.

SEC. 9.5 SUBROUTINE SVD 233 С DO 500 II = 1. MN I = MN + I - II L = I + I G = W(I)IF (I .EQ. N) GO TO 430 С DO 420 J = L N 420 U(1,J) = 0.0С 430 IF (G .EQ. 0.0) GO TO 475 IF (I .EQ. MN) GO TO 460 С DO 450 J = L N S = 0.0 С DO 440 K = L M DO 440 K = L M S = S + U(K.I) • U(K.J) DOUBLE DIVISION AVOIDS POSSIBLE UNDERFLOW 440 С С DO 450 K = J. M U(KJ) = U(KJ) + F · U(KJ) 450 С 460 DO 470 J = 1. M 470 U(J.I) = U(J.I) / G С GO TO 490 С 475 DO 480 J = I. M 480 U(J,I) = 0.0 С 490 U(1.1) = U(1.1) + 1.0 500 CONTINUE DIAGONALIZATION OF THE BIDIAGONAL FORM С С ITS = 0 TEST FOR SPLITTING. с с IESI FOR SPLITTING. FOR L=K STEP -I UNTIL I DO - DO 530 LL = I. K LI = K - LL 520 L = LI + I IF (ABS(RVI(L)) + ANORM .EQ. ANORM) GO TO 565 ... RVI(I) IS ALWAYS ZERO. SO THERE IS NO EXIT THROUGH THE BOTTOM OF THE LOOP IF (ABS(W(LI)) + ANORM .EQ. ANORM) GO TO 540 DATING с с 5.30 CONTINUE

t

1

ĺ

ŧ

•••••

.

•

.

1971

Handbook for Automatic Computation

C.

<u>_</u>

• • •

Edited by F. L. Bauer · A. S. Householder · F. W. J. Olver H. Rutishauser † · K. Samelson · E. Stiefel

Volume II

J.H.Wilkinson · C. Reinsch

Linear Algebra

Chief editor F. L. Bauer



Springer-Verlag New York Heidelberg Berlin 1971

•••••

4. ALGOL Programs

procedure trcd1 (n, tol) trans: (a) result: (d, e, e2);

value n, tol; integer n; real tol; array a, d,.e, c2;

comment This procedure reduces the given lower triangle of a symmetric matrix. A, stored in the array a[1:n, 1:n], to tridiagonal form using Householder's reduction. The diagonal of the result is stored in the array d[1:n] and the sub-diagonal in the last n-1 stores of the array e[1:n](with the additional element e[1]=0). e2[i] is set to equal $e[i]\uparrow 2$. The strictly lower triangle of the array a, together with the array e, is used to store sufficient information for the details of the transformation to be recoverable in the procedure trbak 1. The upper triangle of the array a is left unaltered;

begin

integer i, j, k, l; real /, g, h; for i := 1 step 1 until n do d[i] := a[i, i];for $i := \pi$ step -1 until 1 do begin l:=i-1; h:=0;for k := 1 step 1 until l do $h := h + a[i, k] \times a[i, k];$ comment if k is too small for orthogonality to be guaranteed, the transformation is skipped; if $h \leq tol$ then begin e[i] := e2[i] := 0; go to skip end: e2[i] := h; f := a[i, i-1]; $e[s] := g := if f \ge 0$ then -sqrt(h) else sqrt(h); $h:=h-/\times g; a[i,i-1]:=f-g; f:=0;$ for j := 1 step 1 until l do begin g := 0; **comment** form element of $A \times u$; for k := 1 step 1 until j do $g := g + a[j, k] \times a[i, k];$ for k := j + 1 step 1 until l do $g := g + a[k, j] \times a[i, k];$ comment form element of p; $g := e[f] := g/h; f := f + g \times a[i, f]$ end j; comment form K; h := f/(h+h);comment form reduced A; for j := 1 step 1 until l do begin $f := a[i, j]; g := e[j] := e[j] - h \times f;$ for k := 1 step 1 until j do $a[j, k] := a[j, k] - f \times c[k] - g \times a[i, k]$ end *i*: skip: h := d[i]; d[i] := a[i, i]; a[i, i] := hend i

end trad 1;

ł

```
procedure lql1 (n, macheps) trans: (d, e) exit: (fail);
value n, macheps; integer n; real macheps; array d, e; label /ail;
comment This procedure finds the eigenvalues of a tridiagonal matrix, T, given
            with its diagonal elements in the array d[1:n] and its subdiagonal
            elements in the last n-1 stores of the array e[1:n], using QL trans-
            formations. The eigenvalues are overwritten on the diagonal elements
            in the array d in ascending order. The procedure will fail if any one
            eigenvalue takes more than 30 iterations;
            integer i, j, l, m;
            real b, c, f, g, h, p, r, s;
            for i := 2 step 1 until n do e[i-1] := e[i];
            c[n] := b := f := 0;
            for l := 1 step 1 until \pi do
            begin j := 0; h := macheps \times (abs(d[l]) + abs(e[l]));
                   if b < h then b := h;
                   comment look for small sub-diagonal element;
                   for m := l step 1 until n do
                   if abs(e[m]) \leq b then go to cont1;
            cont1: if m = l then go to root;
            nextil: if j = 30 then go to [ail;
                   i := i + 1;
                   comment form shift;
                   g := d[l]; p := (d[l+1] - g)/(2 \times e[l]); r := sqrt(p^{+}2 + 1);
                   d[l] := c[l]/(\text{if } p < 0 \text{ then } p - r \text{ else } p + r); h := g - d[l];
                   for i:=l+1 step 1 until n do d[i]:=d[i]-h;
                   f := f + h;
                   comment QL transformation;
                   p := d[m]; c := 1; s := 0;
                   for i := m - 1 step - 1 until l do
                   begin g := c \times c[i]; h := c \times p;
                          if abs(p) \ge abs(e[i]) then
                          begin c := c[i]/p; r := sqrl(c\uparrow 2+1);
                                 e[i+1] := s \times p \times r; s := c/r; c := 1/r
                          end
                          else
                          begin c := p/e[i]; r := sqrt(c^{+}2+1);
                                 e[i+1] := s \times e[i] \times r; \ s := 1/r; \ c := c/r
                          end;
                          p := c \times d[s] - s \times g;
                          d[i+1] := h + s \times (c \times g + s \times d[i])
                   end i;
                   e[l] := s \times p; \ d[l] := c \times p;
                   if abs(e[l]) > b then go to nextit;
            rool: p := d[l] + /;
                   comment order eigenvalue;
                   for i := l step -1 until 2 do
                   if p < d[i-1] then d[i] := d[i-1] else go to cont?;
                   i := 1:
            cont2: d[i] := p
            end l
end lql1;
```

begin



1974

Lecture Notes in Computer Science

Edited by G. Goos, Karlsruhe and J. Hartmanis, Ithaca

PROPERTY OF: CLEVE MOLER DEPARTMENT OF MATHEMATICS UNIVERSITY OF NEW MEXICO

6

B. T. Smith \cdot J. M. Boyle \cdot B. S. Garbow Y. Ikebe \cdot V. C. Klema \cdot C. B. Moler

Matrix Eigensystem Routines – EISPACK Guide



Springer-Verlag Berlin · Heidelberg · New York 1974

:

7.1-201

: **(**

1

í

τ

÷

·

.

Ł

.

i

SUBROUTINE TRED1 (NM, N, A, D, E, E2) C INTEGER I,J,K,L,N,II,NM,JP1 REAL A(NM,N),D(N),E(N),E2(N) REAL F,G,H,SCALE REAL SQRT,ABS,SIGN С DO 100 I = 1, N 100 D(I) = A(I,I) FOR I=N STEP -1 UNTIL 1 DO --С DO 300 II = 1, N I = N + 1 - II L = I - 1 H = 0.0 C 120 С IF (SCALE .NE. 0.0) GO TO 140 E(I) = 0.0 E2(I) = 0.0 GO TO 290 130 С DO 150 K = 1, L A(I,K) = A(I,K) / SCALE H = H + A(I,K) * A(I,K) 140 CONTINUE 150 С E2(I) = SCALE * SCALE * H F = A(I,L) G = -SIGN(SQRT(H),F) E(I) = SCALE * G H = H - F * G A(I,L) = F - G IF (L .EQ. 1) GO TO 270 F = 0.0 С DO 240 J = 1, L G = 0.0 ******** FORM ELEMENT OF A*U ********* С DO 180 K = 1, J G = G + A(J,K) * A(I,K) 180 С JP1 = J + 1 IF (L .LT. JP1) GO TO 220 С DO 200 K = JP1, L G = G + A(K,J) * A(I,K)****** FORM ELEMENT OF P E(J) = G / H F = F + E(J) * A(I,J)CONTINUE 200 С 220 CONTINUE 240

345

.

7.1-202

<u>'</u> .

!

C H = F / (H + H) C H = F / (H + H) DO 260 J = 1, L F = A(I,J) G = E(J) - H * F E(J) = G C DO 260 K = 1, J A(J,K) = A(J,K) - F * E(K) - G * A(I,K) C 260 CONTINUE 270 DO 280 K = 1, L 280 A(I,K) = SCALE * A(I,K) C 290 H = D(I) D(I) = A(I,I) A(I,I) = H 300 CONTINUE C RETURN END

346

• • • • . .

· · · · · ·

.

• • • • • •

С,

. .

331

SUBROUTINE TQL1 (N, D, E, IERR) С INTEGER I,J,L,M,N,II,L1,MML,IERR REAL D(N),E(N) REAL B,C,F,G,H,P,R,S,MACHEP REAL SQRT,ABS,SIGN CCCCC ********** MACHEP IS A MACHINE DEPENDENT PARAMETER SPECIFYING THE RELATIVE PRECISION OF FLOATING POINT ARITHMETIC. ******** MACHEP = ?C IERR = 0 IF (N .EQ. 1) GO TO 1001 С DO 100 I = 2, N 100 E(I-1) = E(I)С F = 0.0B = 0.0E(N) = 0.0С DO 290 L = 1, N J = 0C C C CONTINUE 110 С 120 130 С L1 = L + 1G = D(L)P = (D(L1) - G) / (2.0 * E(L)) R = SQRT(P*P+1.0) D(L) = E(L) / (P + SIGN(R,P)) H = G - D(L) С DO 140 I = L1, N D(I) = D(I) - H 140 С F = F + HС ******* QL TRANSFORMATION ********* P = D(M)C = 1.0S = 0.0MML = M - L

328

•••••••••••

.

ł

• # 332

• . • . ••••

· · .

:

....

۵

```
7.1-185
ARARAMAN FOR I-M-1 STEP -1 UNTIL L DO -- ARARAMAN
```

.

С

: •

1

ſ

DO 200 II = 1, MNL I = M - II G = C * E(I) H = C * P • H = C * P IF (ABS(P) .LT. ABS(E(I))) GO TO 150 C = E(I) / P R = SQRT(C*C+1.0) E(I+1) = S * P * R S = C / R C = I 0 / P C = 1.0 / RC = 1.0 / R GO TO 160 C = P / E(I) R = SQRT(C*C+1.0) E(I+1) = S * E(I) * R S = 1.0 / R C = C * S P = C * D(I) - S * G D(I+1) = H + S * (C * G + S * D(I)) NTINUE 150 160 200 CONTINUE С E(L) = S * P D(L) = C * P IF (ABS(E(L)) .GT. B) GO TO 130 P = D(L) + F ********** ORDER EIGENVALUES ********* IF (L .EQ. 1) GO TO 250 ********** FOR I=L STEP -1 UNTIL 2 DO -- ********** DO 230 II = 2, L I = L + 2 - II IF (P .GE. D(I-1)) GO TO 270 D(I) = D(I-1) CONTINUE 210 С С 230 CONTINÚE С 250 $\begin{array}{c} I = 1 \\ D(I) = P \end{array}$ 270 290 CONTINUE С GO TO 1001 C C EIGENVALUE AFTER 30 ITERATIONS ********* 1000 IERR - L 1001 RETURN END

329

. .

ين اور والمراجع والمراجع من المراجع المراجع والمراجع

-

1

983

DOUBLE PRECISION FUNCTION EPSLON (X) DOUBLE PRECISION X С ESTIMATE UNIT ROUNDOFF IN QUANTITIES OF SIZE X. С С DOUBLE PRECISION A, B, C, EPS С С THIS PROGRAM SHOULD FUNCTION PROPERLY ON ALL SYSTEMS С SATISFYING THE FOLLOWING TWO ASSUMPTIONS, 1. THE BASE USED IN REPRESENTING FLOATING POINT С NUMBERS IS NOT A POWER OF THREE. С С 2. THE QUANTITY A IN STATEMENT 10 IS REPRESENTED TO THE ACCURACY USED IN FLOATING POINT VARIABLES С С THAT ARE STORED IN MEMORY. С THE STATEMENT NUMBER 10 AND THE GO TO 10 ARE INTENDED TO С FORCE OPTIMIZING COMPILERS TO GENERATE CODE SATISFYING С ASSUMPTION 2. С UNDER THESE ASSUMPTIONS, IT SHOULD BE TRUE THAT, С A IS NOT EXACTLY EQUAL TO FOUR-THIRDS, с с B HAS A ZERO FOR ITS LAST BIT OR DIGIT, C IS NOT EXACTLY EQUAL TO ONE, Ċ EPS MEASURES THE SEPARATION OF 1.0 FROM С THE NEXT LARGER FLOATING POINT NUMBER. С THE DEVELOPERS OF EISPACK WOULD APPRECIATE BEING INFORMED С ABOUT ANY SYSTEMS WHERE THESE ASSUMPTIONS DO NOT HOLD. С С THIS VERSION DATED 4/6/83. С A = 4.0D0/3.0D010 B = A - 1.0D0C = B + B + BEPS = DABS(C-1.0D0)IF (EPS .EQ. 0.0D0) GO TO 10 EPSLON = EPS * DABS(X)RETURN END

. .

.

1

SUBROUTINE TRED1 (NM, N, A, D, E, E2) С INTEGER I, J, K, L, N, II, NM, JP1 DOUBLE PRECISION A (NM, N), D (N), E (N), E2 (N) DOUBLE PRECISION F, G, H, SCALE С THIS SUBROUTINE IS A TRANSLATION OF THE ALGOL PROCEDURE TRED1, С NUM. MATH. 11, 181-195(1968) BY MARTIN, REINSCH, AND WILKINSON. С HANDBOOK FOR AUTO. COMP., VOL.II-LINEAR ALGEBRA, 212-226(1971). С С THIS SUBROUTINE REDUCES A REAL SYMMETRIC MATRIX C TO A SYMMETRIC TRIDIAGONAL MATRIX USING С С ORTHOGONAL SIMILARITY TRANSFORMATIONS. С С ON INPUT С NM MUST BE SET TO THE ROW DIMENSION OF TWO-DIMENSIONAL С ARRAY PARAMETERS AS DECLARED IN THE CALLING PROGRAM С С DIMENSION STATEMENT. С С N IS THE ORDER OF THE MATRIX. С С A CONTAINS THE REAL SYMMETRIC INPUT MATRIX. ONLY THE С LOWER TRIANGLE OF THE MATRIX NEED BE SUPPLIED. С ON OUTPUT С С С A CONTAINS INFORMATION ABOUT THE ORTHOGONAL TRANS-С FORMATIONS USED IN THE REDUCTION IN ITS STRICT LOWER С TRIANGLE. THE FULL UPPER TRIANGLE OF A IS UNALTERED. С С D CONTAINS THE DIAGONAL ELEMENTS OF THE TRIDIAGONAL MATRIX. С С E CONTAINS THE SUBDIAGONAL ELEMENTS OF THE TRIDIAGONAL С MATRIX IN ITS LAST N-1 POSITIONS. E(1) IS SET TO ZERO. С С E2 CONTAINS THE SQUARES OF THE CORRESPONDING ELEMENTS OF E. С E2 MAY COINCIDE WITH E IF THE SQUARES ARE NOT NEEDED. С С QUESTIONS AND COMMENTS SHOULD BE DIRECTED TO BURTON S. GARBOW, С MATHEMATICS AND COMPUTER SCIENCE DIV, ARGONNE NATIONAL LABORATORY С С THIS VERSION DATED APRIL 1983. С С С DO 100 I = 1, N D(I) = A(N, I)A(N,I) = A(I,I)**100 CONTINUE** С DO 300 II = 1, N I = N + 1 - IIL = I - 1H = 0.0D0SCALE = 0.0D0IF (L .LT. 1) GO TO 130 С SCALE ROW (ALGOL TOL THEN NOT NEEDED) DO 120 K = 1, L 120 SCALE = SCALE + DABS (D(K)) С IF (SCALE .NE. 0.0D0) GO TO 140 С DO 125 J = 1, LD(J) = A(L, J)

tredl.f ·

: **r**:

Ľ

.

.

A(L,J) = A(I,J)A(I,J) = 0.0D0125 CONTINUE С 130 E(I) = 0.0D0E2(I) = 0.0D0GO TO 300 С 140 DO 150 K = 1, L D(K) = D(K) / SCALEH = H + D(K) + D(K)CONTINUE 150 С E2(I) = SCALE * SCALE * HF = D(L)G = -DSIGN(DSQRT(H), F)E(I) = SCALE * G $H = H - F \star G$ D(L) = F - GIF (L .EQ. 1) GO TO 285 С FORM A*U DO 170 J = 1, L170 E(J) = 0.0D0С DO 240 J = 1, LF = D(J)G = E(J) + A(J,J) * FJP1 = J + 1IF (L .LT. JP1) GO TO 220 С DO 200 K = JP1, L G = G + A(K, J) + D(K) $E(K) = E(K) + A(K,J) \star F$ 200 CONTINUE С 220 E(J) = G240 CONTINUE FORM P С F = 0.0D0С DO 245 J = 1, L E(J) = E(J) / H $\mathbf{F} = \mathbf{F} + \mathbf{E}(\mathbf{J}) + \mathbf{D}(\mathbf{J})$ 245 CONTINUE С H = F / (H + H)C FORM Q DO 250 J = 1, L 250 E(J) = E(J) - H + D(J)С FORM REDUCED A DO 280 J = 1, LF = D(J)G = E(J)С DO 260 K = J, L 260 A(K,J) = A(K,J) - F * E(K) - G * D(K)С 280 CONTINUE С 285 DO 290 J = 1, LF = D(J)D(J) = A(L, J)A(L,J) = A(I,J)A(I,J) = F * SCALE290 CONTINUE

· · .

335

:

•

tredl.f	Tri May	1 17:23:47 1987	3

C 300 CONTINUE C RETURN END

• .

• • •

.

ć

С

С С

С

С

С С

С С С

С

•••

. . . .

. . .

· · · ---

SUBROUTINE TQL1 (N, D, E, IERR) С INTEGER I, J, L, M, N, II, L1, L2, MML, IERR DOUBLE PRECISION D(N), E(N) DOUBLE PRECISION C,C2,C3,DL1,EL1,F,G,H,P,R,S,S2,TST1,TST2,PYTHAG THIS SUBROUTINE IS A TRANSLATION OF THE ALGOL PROCEDURE TQL1, NUM. MATH. 11, 293-306(1968) BY BOWDLER, MARTIN, REINSCH, AND WILKINSON. HANDBOOK FOR AUTO. COMP., VOL.II-LINEAR ALGEBRA, 227-240(1971). THIS SUBROUTINE FINDS THE EIGENVALUES OF A SYMMETRIC TRIDIAGONAL MATRIX BY THE QL METHOD. ON INPUT С N IS THE ORDER OF THE MATRIX. С D CONTAINS THE DIAGONAL ELEMENTS OF THE INPUT MATRIX. С С С E CONTAINS THE SUBDIAGONAL ELEMENTS OF THE INPUT MATRIX IN ITS LAST N-1 POSITIONS. E(1) IS ARBITRARY. С С С ON OUTPUT С С D CONTAINS THE EIGENVALUES IN ASCENDING ORDER. IF AN С ERROR EXIT IS MADE, THE EIGENVALUES ARE CORRECT AND С ORDERED FOR INDICES 1,2,...IERR-1, BUT MAY NOT BE С THE SMALLEST EIGENVALUES. С С E HAS BEEN DESTROYED. С С IERR IS SET TO С ZERO FOR NORMAL RETURN, С J IF THE J-TH EIGENVALUE HAS NOT BEEN С DETERMINED AFTER 30 ITERATIONS. С CALLS PYTHAG FOR DSQRT (A*A + B*B) . С С С QUESTIONS AND COMMENTS SHOULD BE DIRECTED TO BURTON S. GARBOW, С MATHEMATICS AND COMPUTER SCIENCE DIV, ARGONNE NATIONAL LABORATORY С С THIS VERSION DATED APRIL 1983. С С С IERR = 0IF (N .EQ. 1) GO TO 1001 С DO 100 I = 2, N 100 E(I-1) = E(I)С F = 0.0D0TST1 = 0.0D0E(N) = 0.0D0С \dot{DO} 290 L = 1, N J = 0H = DABS(D(L)) + DABS(E(L))IF (TST1 .LT. H) TST1 = H C LOOK FOR SMALL SUB-DIAGONAL ELEMENT DO 110 M = L, NTST2 = TST1 + DABS(E(M))IF (TST2 .EQ. TST1) GO TO 120 C E(N) IS ALWAYS ZERO, SO THERE IS NO EXIT

. .

1

•

tql1.f

338

2

ļ

7

: •

•••••

```
С
                 THROUGH THE BOTTOM OF THE LOOP .....
  110
         CONTINUE
C
         IF (M .EQ. L) GO TO 210
  120
         IF (J .EQ. 30) GO TO 1000
  130
         J = J + 1
        ..... FORM SHIFT .....
С
         L1 = L + 1
         L2 = L1 + 1
         G = D(L)
         P = (D(L1) - G) / (2.0D0 * E(L))
         R = PYTHAG(P, 1.0D0)
         D(L) = E(L) / (P + DSIGN(R, P))
         D(L1) = E(L) + (P + DSIGN(R,P))
         DL1 = D(L1)
         H = G - D(L)
         IF (L2 .GT. N) GO TO 145
С
         DO 140 I = L2, N
         D(I) = D(I) - H
  140
С
  145
         \mathbf{F} = \mathbf{F} + \mathbf{H}
       ..... QL TRANSFORMATION .....
С
         P = D(M)
         C = 1.0D0
         C2 = C
         EL1 = E(L1)
         S = 0.0D0
         MML = M - L
       ..... FOR I=M-1 STEP -1 UNTIL L DO -- .....
С
         DO 200 II = 1, MML
            C3 = C2
            C2 = C
             S2 - S
             I = M - II
             G = C \star E(I)
            H = C \star P
            R = PYTHAG(P, E(I))
            E(I+1) = S * R
             S = E(I) / R
             C = P / R
             P = C + D(I) - S + G
             D(I+1) = H + S * (C * G + S * D(I))
  200
         CONTINUE
С
          P = -S + S2 + C3 + EL1 + E(L) / DL1
         E(L) = S * P
         D(L) = C \star P
          TST2 = TST1 + DABS(E(L))
          IF (TST2 .GT. TST1) GO TO 130
  210
          P = D(L) + F
С
         ..... ORDER EIGENVALUES .....
          IF (L .EQ. 1) GO TO 250
       ..... FOR I=L STEP -1 UNTIL 2 DO -- .....
С
          DO 230 II = 2, L
             I = L + 2 - II
             IF (P .GE. D(I-1)) GO TO 270
             D(I) = D(I-1)
   230
          CONTINUE
С
   250
          I = 1
   270
          D(I) = P
   290 CONTINUE
С
       GO TO 1001
```

•••••••

.

.

. .

. . . .

.

• • •

tql1.f Fri May 1 17:23:46 1987 .3

Г

Ċ

Ŧ

é

;

c C	• • • • • • • • • • •	SET ERROR · EIGENVALUE	NO CON AFTER 30	VERGENCE TO ITERATIONS	AN
1000 1001	ierr = l Return End		•		

. . . .

1

SUBROUTINE TOLRAT (N, D, E2, IERR) С INTEGER I, J, L, M, N, II, L1, MML, IERR DOUBLE PRECISION D(N), E2(N) DOUBLE PRECISION B, C, F, G, H, P, R, S, T, EPSLON, PYTHAG character*20 string С THIS SUBROUTINE IS A TRANSLATION OF THE ALGOL PROCEDURE TOLRAT, С ALGORITHM 464, COMM. ACM 16, 689(1973) BY REINSCH. С C THIS SUBROUTINE FINDS THE EIGENVALUES OF A SYMMETRIC C TRIDIAGONAL MATRIX BY THE RATIONAL QL METHOD. С С С ON INPUT С С N IS THE ORDER OF THE MATRIX. С С D CONTAINS THE DIAGONAL ELEMENTS OF THE INPUT MATRIX. С С E2 CONTAINS THE SQUARES OF THE SUBDIAGONAL ELEMENTS OF THE С INPUT MATRIX IN ITS LAST N-1 POSITIONS. E2(1) IS ARBITRARY. С C ON OUTPUT С С D CONTAINS THE EIGENVALUES IN ASCENDING ORDER. IF AN С ERROR EXIT IS MADE, THE EIGENVALUES ARE CORRECT AND С ORDERED FOR INDICES 1,2,... IERR-1, BUT MAY NOT BE С THE SMALLEST EIGENVALUES. С С E2 HAS BEEN DESTROYED. С С IERR IS SET TO С ZERO FOR NORMAL RETURN, С J IF THE J-TH EIGENVALUE HAS NOT BEEN С DETERMINED AFTER 30 ITERATIONS. С С CALLS PYTHAG FOR DSORT (A*A + B*B) . С С QUESTIONS AND COMMENTS SHOULD BE DIRECTED TO BURTON S. GARBOW, С MATHEMATICS AND COMPUTER SCIENCE DIV, ARGONNE NATIONAL LABORATORY С С THIS VERSION DATED APRIL 1983. C С С IERR = 0IF (N .EQ. 1) GO TO 1001 С DO 100 I = 2, N 100 E2(I-1) = E2(I)С F = 0.0D0T = 0.0D0 E2(N) = 0.0D0С DO 290 L = 1, N J = 0H = DABS(D(L)) + DSQRT(E2(L))IF (T .GT. H) GO TO 105 T = HB = EPSLON(T)C = B * B..... LOOK FOR SMALL SQUARED SUB-DIAGONAL ELEMENT С DO 110 M = L, N 105

4

t

(

(

IF (E2(M) .LE. C) GO TO 120

tgirat.f

: **(**

1

t

. (

E

2

34

```
..... E2 (N) IS ALWAYS ZERO, SO THERE IS NO EXIT
С
                THROUGH THE BOTTOM OF THE LOOP .....
С
 110
        CONTINUE
С
  120
        IF (M .EQ. L) GO TO 210
        IF (J .EQ. 30) GO TO 1000
  130
         \mathbf{J}=\mathbf{J}+\mathbf{1}
      ..... FORM SHIFT .....
С
        L1 = L + 1
         S = DSQRT(E2(L))
         G = D(L)
         P = (D(L1) - G) / (2.0D0 * S)
         R = PYTHAG(P, 1.0D0)
        D(L) = S / (P + DSIGN(R, P))
         H = G - D(L)
С
        DO 140 I = L1, N
  140
         D(I) = D(I) - H
С
         \mathbf{F} = \mathbf{F} + \mathbf{H}
        ..... RATIONAL QL TRANSFORMATION .....
С
         G = D(M)
         IF (G .EQ. 0.0D0) G = B
         H = G
         S = 0.0D0
         MML = M - L
        ..... FOR I=M-1 STEP -1 UNTIL L DO -- .....
С
         DO 200 II = 1, MML
            I = M - II
            P = G * H
            R = P + E2(I)
            E2(I+1) = S * R
            S = E2(I) / R
            D(I+1) = H + S * (H + D(I))
            G = D(I) - E2(I) / G
            IF (G .EQ. 0.0D0) G = B
            H = G * P / R
  200
         CONTINUE
С
         E2(L) = S * G
         D(L) = H
       ...... GUARD AGAINST UNDERFLOW IN CONVERGENCE TEST .....
С
         IF (H .EQ. 0.0D0) GO TO 210
         IF (DABS(E2(L)) .LE. DABS(C/H)) GO TO 210
         E2(L) = H + E2(L)
         IF (E2(L) .NE. 0.0D0) GO TO 130
         P = D(L) + F
  210
С
       ..... ORDER EIGENVALUES .....
         IF (L .EQ. 1) GO TO 250
       ..... FOR I=L STEP -1 UNTIL 2 DO -- .....
С
         DO 230 II = 2, L
            I = L + 2 - II
            IF (P .GE. D(I-1)) GO TO 270
            D(I) = D(I-1)
  230
         CONTINUE
C
  250
         I = 1
  270
         D(I) = P
  290 CONTINUE
С
      GO TO 1001
          ..... SET ERROR -- NO CONVERGENCE TO AN
С
                 EIGENVALUE AFTER 30 ITERATIONS .....
 С
 1000 \text{ IERR} = L
 1001 RETURN
```

.



1

1

į

UNDERFLOW IN EISPACK

987

by Eric Grosse, Bell Labs, Murray Hill, NJ and Cleve Moler, Dana Computer, Sunnyvale, CA

We recently came across an interesting case where EISPACK fails to give the correct eigenvalues for what appears to be an easy matrix. The difficulties can be traced to floating point underflow. They are most insidious in double precision arithmetic on the VAX [*] where the "D" floating point format has an unfortunately small exponent range. However, a scaled version of the example can fail on any machine, including ones which fully conform to the IEEE floating point standard. We recommend a simple change to the EISPACK top level routine "RS" which should protect most users from the problem.

The example is due to Guenter Ziegler of the University of Augsburg in West Germany and Andrew Odlyzko of AT&T Bell Laboratories. They were investigating a question raised by Amir Dembo of Brown University regarding the distribution of rank in real symmetric Hankel matrices whose elements are +1 and -1. (A Hankel matrix is constant along each anti-diagonal, but that's irrelevant for what concerns us here.) One of their matrices is 9-by-9:

-1	1	1	-1	-1	1	1	-1	-1
1	1	-1	-1	1	1	-1	-1	1
1	-1	-1	1	1	-1	-1	1	1
-1	-1	1	1	-1	-1	1	1	-1
-1	1	1	-1	-1	1	1	-1	-1
1	1	-1	-1	1	1	-1	-1	1
1	-1	-1	1	1	-1	-1	1	-1
-1	-1	1	1	-1	-1	1	-1	1
-1	1	1	-1	-1	1	-1	1	1

It is not obvious, but this matrix happens to have four eigenvalues equal to zero, and hence its rank is five. From the many possible ways to compute the rank of such matrices, Zeigler and Odlyzko chose to use the EISPACK routine RS (for Real Symmetric) and count the number of negligible computed eigenvalues. For this example, running on a VAX in D format double precision, EISPACK incorrectly claimed there were five eigenvalues on the order of roundoff error. The same program, running on almost any other computer, would produce the correct answer, which is only four negligible eigenvalues.

The problem turns out to be a catastrophic underflow in the EISPACK routine TQLRAT. This is a square-root-free variant of the QR algorithm for finding eigenvalues of a symmetric tridiagonal matrix. It operates on the squares of off-diagonal elements. On the VAX, the square of double precision roundoff error is roughly $10^{(-34)}$ and the underflow limit is only $10^{(-38)}$. There is not enough room between those two numbers for TQLRAT to operate properly. On other computers, similar difficulties will occur if the example is scaled by a factor on the order of the square root of the underflow limit. For IEEE machines, the scale factor would have to be about $10^{(-150)}$, so such examples are much less likely in practice, but TQLRAT might not properly handle any which do turn up.

1

The easiest solution is to replace

. CALL TOLRAT (N, ALPHA, BETA, IERR)

in EISPACK routine RS by

CALL TOL1 (N, ALPHA, BETA, IERR) .

Since TQL1 does not work with the squares of the tridiagonal elements, it is much less prone to underflow trouble. No change is needed in the case when eigenvectors are being computed, since RS then calls TQL2 rather than TQLRAT.

An alternate solution, an improved version of TQLRAT, is available from the authors. But its range of applicability is still limited to a smaller portion of the floating point exponent range than TQL1 and TQL2.

Ironically, advances in floating point hardware make the need for square-root-free algorithms less pressing. On one recent chip, the builtin square root is even slightly faster than division!

[*] VAX is a trademark of Digital Equipment Corporation.

.

LINPACK Users' Guide

J. J. Dongarra

Argonne National Laboratory

J. R. Bunch University of California, San Diego C. B. Moler University of New Mexico

G. W. Stewart

University of Maryland



and formats Fortran programs to clarify their structure. It also generates variants of programs. The "master versions" of all the LINPACK subroutines are those which use complex arithmetic; versions which use single precision, double precision, and double precision complex arithmetic have been produced automatically by TAMPR. A user may thus convert from one type of arithmetic to another by simply changing the declarations in his program and changing the first letter of the LINPACK subroutines being used.

Anyone reading the Fortran source code for LINPACK subroutines should find the loops and logical structures clearly delineated by the indentation generated by TAMPR.

The BLAS are the Basic Linear Algebra Subprograms designed by Lawson, Hanson, Kincaid and Krogh (1978). They contribute to the speed as well as to the modularity and clarity of the LINPACK subroutines. LINPACK is distributed with versions of the BLAS written in standard Fortran which are intended to provide reasonably efficient execution in most operating environments. However, a particular computing installation may substitute machine language versions of the BLAS and thereby perhaps improve efficiency.

LINPACK is designed to be completely machine independent. There are no machine dependent constants, no input/output statements, no character manipulation, no COMMON or EQUIVALENCE statements, and no mixed-mode arithmetic. All the subroutines (except those whose names begin with Z) use the portable subset of Fortran defined by the PFORT verifier of Ryder (1974).

There is no need for machine dependent constants because there is very little need to check for "small" numbers. For example, candidates for pivots in Gaussian elimination are checked against an exact zero rather than against some small quantity. The test for singularity is made instead by estimating the condition of the matrix; this is not only machine independent, but also far more reliable. The convergence of the iteration in the singular value decomposition is tested in a machine independent manner by statements of the form

> TEST1 = something not small TEST2 = TEST1 + something possibly small IF (TEST1 .EQ. TEST2) ...

The absence of mixed-mode arithmetic implies that the single precision subroutines do not use any double precision arithmetic and hence that the double precision subroutines do not require any kind of extended precision. It also implies that LINPACK does not include a subroutine for iterative improvement; however, an example in Chapter 1 indicates how such

I.4

a subroutine could be added by anyone with easy access to mixed-mode arithmetic. (Some of the BLAS involve mixed-mode arithmetic, but they are not used by LINPACK.)

Floating point underflows and overflows may occur in some of the LINPACK subroutines. Any underflows which occur are harmless. We hope that the operating system sets underflowed quantities to zero and continues operation without producing any error messages. With some operating systems, it may be necessary to insert control cards or call special system subroutines to achieve this type of underflow handling.

Overflows, if they occur, are much more serious. They must be regarded as error situations resulting from improper use of the subroutines or from unusual scaling. Many precautions against overflow have been taken in LINPACK, but it is impossible to absolutely prevent overflow without seriously degrading performance on reasonably scaled problems. It is expected that overflows will cause the operating system to terminate the computation and that the user will have to correct the program or rescale the problem before continuing.

Fortran stores matrices by columns and so programs in which the inner loop goes up or down a column, such as

```
DO 20 J = 1, N
DO 10 I = 1, N
A(I,J) = ...
10 CONTINUE
20 CONTINUE
```

generate sequential access to memory. Programs in which the inner loop goes across a row cause non-sequential access. Sequential access is preferable on operating systems which employ virtual memory or other forms of paging. LINPACK is consequentially "column oriented". Almost all the inner loops occur within the BLAS and, although the BLAS allow a matrix to be accessed by rows, this provision is never used by LINPACK. The column orientation requires revision of some conventional algorithms, but results in significant improvement in performance on operating systems with paging and cache memory.

All square matrices which are parameters of LINPACK subroutines are specified in the calling sequences by three arguments, for example

CALL SGEFA(A,LDA,N,...)

Here A is the name of a two-dimensional Fortran array, LDA is the leading dimension of that array, and N is the order of the matrix stored in the array or in a portion of the array. The two parameters LDA and N have different meanings and need not have the same value. The amount of storage reserved for the array A is determined by a declaration in

С С MAIN ITERATION LOOP FOR THE SINGULAR VALUES. С MM = MITER = 0360 · CONTINUE С С QUIT IF ALL THE SINGULAR VALUES HAVE BEEN FOUND. С С ...EXIT IF (M .EQ. 0) GO TO 620 С С IF TOO MANY ITERATIONS HAVE BEEN PERFORMED, SET С FLAG AND RETURN. С IF (ITER .LT. MAXIT) GO TO 370 INFO = MС .EXIT GO TO 620 370 CONTINUE С THIS SECTION OF THE PROGRAM INSPECTS FOR NEGLIGIBLE ELEMENTS IN THE S AND E ARRAYS. С Ĉ ON COMPLETION THE VARIABLES KASE AND L ARE SET AS FOLLOWS. C C C C KASE = 1IF S(M) AND E(L-1) ARE NEGLIGIBLE AND L.LT.M KASE = 2IF S(L) IS NEGLIGIBLE AND L.LT.M IF E(L-1) IS NEGLIGIBLE, L.LT.M, AND S(L), ..., S(M) ARE NOT NEGLIGIBLE (QR STEP). IF E(M-1) IS NEGLIGIBLE (CONVERGENCE). KASE = 3Ĉ С KASE = 4С DO 390 LL = 1, M L = M - LL...EXIT С IF (L .EQ. 0) GO TO 400 TEST = ABS(S(L)) + ABS(S(L+1))ZTEST = TEST + ABS(E(L))IF (ZTEST .NE. TEST) GO TO 380 E(L) = 0.0E0СEXIT GO TO 400 380 CONTINUE 390 CONTINUE 400 CONTINUE IF (L .NE. M - 1) GO TO 410 KASE = 4GO TO 480 410 CONTINUE LP1 = L + 1MP1 = M + 1DO 430 LLS = LP1, MP1 LS = M - LLS + LP1С ...EXIT IF (LS .EQ. L) GO TO 440 TEST = 0.0E0IF (LS .NE. M) TEST = TEST + ABS(E(LS))IF (LS .NE. L + 1) TEST = TEST + ABS(E(LS-1)) ZTEST = TEST + ABS(S(LS)) IF (ZTEST .NE. TEST) GO TO 420 S(LS) = 0.0E0СEXIT GO TO 440 420 CONTINUE 430 CONTINUE 440 CONTINUE IF (LS .NE. L) GO TO 450 KASE = 3 GO TO 470 450 CONTINUE IF (LS .NE. M) GO TO 460

Cleve Moler Donald Morrison

1983

Replacing Square Roots by Pythagorean Sums

An algorithm is presented for computing a "Pythagorean sum" $a \oplus b = \sqrt{a^2 + b^2}$ directly from a and b without computing their squares or taking a square root. No destructive floating point overflows or underflows are possible. The algorithm can be extended to compute the Euclidean norm of a vector. The resulting subroutine is short, portable, robust, and accurate, but not as efficient as some other possibilities. The algorithm is particularly attractive for computers where space and reliability are more important than speed.

1. Introduction

It is generally accepted that "square root" is a fundamental operation in scientific computing. However, we suspect that square root is actually used most frequently as part of an even more fundamental operation which we call Pythagorean addition:

 $a \oplus b = \sqrt{a^2 + b^2}.$

The algebraic properties of Pythagorean addition are very similar to those of ordinary addition of positive numbers. Pythagorean addition is also the basis for many different computations:

Polar conversion:

$$r = x \oplus y$$

Complex modulus:

 $|z| = real(z) \oplus imag(z);$

Euclidean vector norm:

 $[v] = v_1 \oplus v_2 \oplus \cdots \oplus v_n;$

Givens rotations:

$$\begin{pmatrix} c & s \\ -s & c \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} r \\ 0 \end{pmatrix};$$

where $r = x \oplus y$, c = x/r, s = y/r.

The conventional Fortran construction

R = SQRT(X - 2 + Y - 2)

may produce damaging underflows and overflows even though the data and the result are well within the range of the machine's floating point number system. Similar constructions in other programming languages may cause the same difficulties.

The remedies currently employed in robust mathematical software lead to code which is clever, but unnatural, lengthy, possibly slow, and sometimes not portable. This is even true of the recently published approaches to the calculation of the Euclidean vector norm by Blue [1] and by the Basic Linear Algebra Subprograms group, Lawson et al. [2].

In this paper we present an algorithm pythog(a,b) which computes $a \oplus b$ directly from a and b, without squaring them and without taking any square roots. The result is robust, portable, short, and, we think, elegant. It is also potentially faster than a square root. We recommend that the algorithm be considered for implementation in machine language or microcode on future systems.

One of our first uses of pythag and the resulting Euclidean norm involved a graphics minicomputer which has a sophisticated Fortran-based operating system, but only about 32K bytes of memory available to the user. We implemented

• Copyright 1983 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions, of this paper may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor.

577

IBM J. RES. DEVELOP. . VOL. 27 . NO. 6 . NOVEMBER 1983

CLEVE MOLER AND DONALD MORRISON

MATLAB [3], an interactive matrix calculator based on LINPACK and EISPACK. In this setting, the space occupied by both source and object code was crucial. MATLAB does matrix computations in complex arithmetic, so pythag is particularly useful. We are able to produce robust, portable software that uses the full range of the floating point exponent.

2. Algorithm pythag

The algorithm for computing $pythag(a,b) = a \oplus b$ is

real function pythag(a,b) real a,b,p,q,r,s p := max(|a|,|b|) q := min(|a|,|b|)while (q is numerically significant) do $r := (q/p)^2$ s := r/(4+r)

s := r/(4+r) p := p+2*s*p

q :**≕ s***q od

pythag := p

The two variables p and q are initialized so that

 $p \oplus q = a \oplus b \text{ and } 0 \leq q \leq p.$

The main part of the algorithm is an iteration that leaves $p \oplus q$ invariant while increasing p and decreasing q. Thus when q becomes negligible, p holds the desired result. We show in Section 4 that the algorithm is cubically convergent and that it will never require more than three iterations on any computer with 20 or fewer significant digits. It is thus potentially faster than the classical quadratically convergent iteration for square root.

There are no square roots involved and, despite the title of this paper, the algorithm cannot be used to compute a square root. If either argument is zero, the result is the absolute value of the other argument.

Typical behavior of the algorithm is illustrated by pythag(4,3). The values of p and q after each iteration are

The most important feature of the algorithm is its robust-

ness. There will be no overflows unless the final result

overflows. In fact, no intermediate results larger than $a \oplus b$

iteration	p.	9
0	4.00000000000	3.000000000000
1	4.986301369863	0.369863013698
2	4.99999 99974188	0.000508052633
3	5.00000000000	0.00000000001

578

are involved. There may be underflows if |b| is much smaller than |a|, but as long as such underflows are quietly set to zero, no harm will result in most cases.

ł

There can be some deterioration in accuracy if both |a| and |b| are very near μ , the smallest positive floating point number. As an extreme example, suppose $a = 4\mu$ and $b = 3\mu$. Then the iterates shown above should simply be scaled by μ . But the value of q after the first iteration would be less than μ and so would be set to zero. The process would terminate early with the corresponding value of p, which is an inaccurate, but not totally incorrect, result.

3. Euclidean vector norm

A primary motivation for our development of pythag is its use in computing the Euclidean norm or 2-norm of a vector. The conventional approach, which simply takes the square root of the sum of the squares of the components, disregards the possibility of underflow and overflow, thereby effectively halving the floating point exponent range. The approaches of Blue [1] and Lawson et al. [2] provide for the possibility of accumulating three sums, one of small numbers whose squares underflow, one of large numbers whose squares overflow, and one of "ordinary-sized" numbers. Environmental inquiries or machine- and accuracy-dependent constants are needed to separate the three classes.

With pythag available, computation of the 2-norm is easy:

real function norm2(x)
real vector x
real s
st= 0
for i := 1 to (number of elements in x)
 s := pythag(s_x(i))
norm2 := x

This algorithm has all the characteristics that might be desired of it, except one. It is robust—there are no destructive underflows and no overflows unless the result must overflow. It is accurate—the round-off error corresponds to a few units in the last digit of each component of the vector. It is portable—there are no machine-dependent constants or environmental inquiries. It is short—both the source code and the object code require very little memory. It accesses each element of the vector only once, which is of some importance in virtual memory and other modern operating systems.

The only possible drawback is its speed. For a vector of length n, it requires n calls to pythag. Even if pythag were implemented efficiently, this is roughly the same as n square roots. The approaches of [1] and [2] require only n multipli-

CLEVE MOLER AND DONALD MORRISON

cations for the most frequent case where the squares of the vector elements do not underflow or overflow. However, in most of the applications we are aware of, speed is not a major consideration. In matrix calculations, for example, the Euclidean norm is usually required only in an outer loop. The time-determining calculations do not involve pythag. Thus, in our opinion, all the advantages outweigh this one disadvantage.

4. Convergence analysis

When the iteration in pythag is terminated and the final value of p accepted as the result, the relative error is

$$e = (p \oplus q - p)/(p \oplus q)$$
$$= (\sqrt{1+r} - 1)/\sqrt{1+r},$$

where $r = (q/p)^2$. (We assume throughout this section that initially p and q are positive.)

The values of e and r are closely related, and the values of their reciprocals are even more closely related. In fact,

$$\frac{1}{e} = \frac{1}{r} + 1 + \frac{\sqrt{1+r}}{r}$$

Since $1 < \sqrt{1+r} < 1 + r/2$, it follows that

$$\frac{2}{r} + 1 < \frac{1}{e} < \frac{2}{r} + \frac{3}{2}.$$

Thus 1/e exceeds 2/r by at least 1 and at most 1.5.

To see how 2/r and hence the relative error varies during the iteration, we introduce the variable

$$u=\frac{4}{r}$$
.

The values of u taken in successive iterations are given by

$$u:=u(u+3)^{\prime}.$$

If the initial value of u is outside the interval $-4 \le u \le -2$, then u increases with each iteration. Hence $u \to \infty$, $r \to 0$, and $p \to a \oplus b$. The fact that u is more than cubed each iteration implies the cubic convergence of the algorithm. Since initially we have $0 < q \le p$, it follows that

$$0 < r \leq 1 \text{ and } 4 \leq u,$$

and u increases rapidly from the very beginning. If the initial value of q/p happens to be an integer, then u takes on integer values.

The most slowly convergent case has initial values p = qand r = 1. The iterated values of μ are

iteration 0 1 2 3 4
$$u$$
 4 196 7761796 >4+10²⁰ >10⁴²

It follows that after three iterations

$$e < \frac{r}{2} - \frac{2}{u} < 0.5 \cdot 10^{-20}.$$

If the arithmetic were done exactly, after three iterations the value of p would agree with the true value of $p \oplus q$ to 20 decimal digits. If there were further iterations, each one would at least triple the number of correct digits. Initial values with q < p produce even more rapid convergence.

With quadratically convergent iterations such as the classical square root algorithm, it is often desirable to use special starting procedures to produce good initial approximations. Our choice of initial values with $q \leq p$ can be regarded as such a starting procedure since the algorithm will converge even without this condition. However, since the convergence is so rapid, it seems unlikely that any more elaborate starting mechanism would offer any advantage.

5. Round-off error and stopping criterion

In addition to being robust with respect to underflow and overflow, the performance of pythag in the presence of round-off error is quite satisfactory. It is possible to show that after each iteration the computed value of the variable pis the same as the value that would be obtained with exact computation on slightly perturbed starting values. The rapid convergence guarantees that there is no chance for excessive accumulation of rounding errors.

The main question is when to terminate the iteration. If we stop too soon, the result is inaccurate. If we do not stop soon enough, we do more work than is necessary. There are several possible types of stopping criteria.

1. Take a fixed number of iterations.

The appropriate number depends upon the desired accuracy: two iterations for 6 or fewer significant digits, three iterations for 20 or fewer significant digits, four iterations for 60 or fewer significant digits. There is thus a very slight machine and precision dependence. Moreover, fewer iterations are necessary for pythag(a,b) with b much smaller than a.

2. Iterate until there is no change.

This can be implemented in a machine-independent manner with something like

ps := p p := p + 2*s*p

IBM J. RES. DEVELOP. . VOL. 27 . NO. 6 . NOVEMBER 1983

if p = ps then exit . .

This is probably the most foolproof criterion, but it always uses one extra iteration, just to confirm that the final iteration was not necessary.

3. Predict that there will be no change.

The idea is to do a simple calculation early in the step that will indicate whether or not the remainder of the step is necessary. If we use $f(x) \neq y$ to mean that the computed value of f(x) equals y, then the condition we wish to predict is

 $p + 2sp \pm p$.

When r is small, then s = r/(4 + r) is less than and almost equal to r/4. Consequently, a sufficient and almost equivalent condition is

 $p + rp/2 \doteq p$.

It might seem that this is equivalent to

2 + r - 2.

However, this is not quite true. Let β be the base of the floating point arithmetic. For any floating point number p in the range $1 \le p < \beta$, the set of floating point numbers d for which

p + d = p

is the same as the set of d for which

 $1+d \doteq 1$.

In other words, the conditions p + dp = p and 1 + d = 1 are precisely equivalent only when p is a power of β .

We have chosen to stop when

 $4 + r \pm 4$.

There are three reasons for this choice. The quantity 4 + r is available early in the step and is needed in computing s. The condition is almost equivalent to predicting no change in p. The variables p and q have already been somewhat contaminated by round-off error from previous steps.

The satisfactory error properties of pythag are inherited by norm2. It is possible to show that the computed value of norm2(x) is the exact Euclidean norm of some vector whose individual elements are within the round-off error of the corresponding elements of x. 6. Some related algorithms

It is possible to compute $\sqrt{a^2 - b^2}$ by replacing the statement

kin pythag with

 $\mathbf{r} := -(\mathbf{q}/\mathbf{p})^2.$

The convergence analysis in Section 4 still applies, except that r and u take on negative values. In particular, when a = b, the initial value of u is -4 and this value does not change. The iteration becomes simply

p := p/3, q := -q/3.

The variable p approaches zero as it should, but the convergence is only linear. If $a \neq b$, the convergence is eventually cubic, but many iterations may be required to enter the cubic regime.

The iteration within pythag effectively computes $p\sqrt{1+r}$. The related cubically convergent algorithm for square root is

function sqrt(z) real z,p,r,s p := 1r := z - 1

while (r is numerically significant)

Although this algorithm will converge for any positive z, it is most effective for values of z near 1. The algorithm can be derived from the approximation

$$\sqrt{1+r}\simeq\frac{4+3r}{4+r},$$

which is accurate to second order for small values of r. The classical quadratically convergent iteration for square root can be derived from the approximation

 $\sqrt{1+r}\simeq 1+\frac{r}{2}.$

which is accurate only to first order. The cubically convergent algorithm requires fewer iterations, but more operations per iteration. Consequently, its relative efficiency depends upon the details of the implementation.

CLEVE MOLER AND DONALD MORRISON

580

IBM J. RES. DEVELOP. + VOL. 27 + NO. 6 + NOVEMBER 1983

The Euclidean norm of a vector can also be computed by a generalization of pythag(a,b) to allow a vector argument with any number of components in place of (a,b), a vector argument with only two components:

vector-pythag(x)
real vector x,q
real p.r.s,t
p := (any nonzero component of x, preferably the largest)
q := (x with p deleted)
while (q is numerically significant)
do
r := (dot product of q/p with itself)
s := r/(4+r)
p := p+2****
q := s**q
od
vector-pythag := p

The convergence analysis of Section 4 applies to this algorithm, but the initial value of u may be less than 4. The convergence is cubic, but the accuracy attained after a fixed number of iterations will generally be less than that of the scalar algorithm. Moreover, it does not seem possible to obtain a practical implementation which retains the simplicity of pythag and norm2.

References

1. J. L. Blue, "A Portable Fortran Program to Find the Euclidean Norm of a Vector," ACM Trans. Math. Software 4, 15-23 (1978).

- C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprograms for Fortran Usage," ACM Trans. Math. Software 5, 308-323 (1979).
- Cleve Moler, "MATLAB Users' Guide," Technical Report CS81-1, Department of Computer Science, University of New Mexico, Albuquerque.

Received June 6, 1983; revised July 15, 1983

Cleve B. Moler Department of Computer Science, University of New Mexico, Albuquerque, New Mexico 87131. Professor Moler has been with the University of New Mexico since 1972. He is currently chairman of the Department of Computer Science. His research interests include numerical analysis, mathematical software, and scientific computing. He received his Ph.D. in mathematies from Stanford University, California, in 1965 and taught at the University of Michigan from 1966 to 1972. Professor Moler is a member of the Association for Computing Machinery and the Society for Industrial and Applied Mathematics.

Donald R. Morrison Department of Computer Science, Uniwersity of New Mexico, Albuquerque, New Mexico 87131. Professor Morrison has been with the University of New Mexico 87131. Professor Morrison has been with the University of New Mexico 87131. He received his Ph.D. in mathematics from the University of Wisconsin in 1950. He taught at Tulane University. New Orleans, Louisiana, from 1950 to 1955, and was a staff member, supervisor, and department manager at Sandia Laboratory from 1955 to 1971. He has published several papers in abstract algebra, computation, information retrieval, and cryptography. Professor Morrison is a member of the Association for Computing Machinery and the Mathematical Association of America.

CLEVE MOLER AND DONALD MORRISON

.

.

. .

. .

.

•

.

.

1