Cleve Moler
Donald Morrison

Replacing Square Roots by Pythagorean Sums

An algorithm is presented for computing a "Pythagorean sum" a ⊕ 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 *pythag(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

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)
  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

iteration	p	q
0	4.000000000000	3.000000000000
1	4.986301369863	0.369863013698
2	4.999999974188	0.000508052633
3	5.000000000000	0.000000000001

The most important feature of the algorithm is its robustness. There will be no overflows unless the final result overflows. In fact, no intermediate results larger than $a \oplus b$ 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.

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
s := 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-

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

$$u := u(u+3)^2.$$

If the initial value of u is outside the interval $-4 \leq u \leq -2$, 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 $0 < q \leq p$, 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 $p = q$ and $r = 1$. The iterated values of u are

iteration	0	1	2	3	4
u	4	196	7761796	$>4*10^{20}$	$>10^{62}$

It follows that after three iterations

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

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 p is 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
```

```
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 $f(x) \doteq y$ to mean that the computed value of $f(x)$ equals y, then the condition we wish to predict is

$$p + 2sp \doteq p.$$

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

$$2 + r \doteq 2.$$

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 $1 \leq p < \beta$, the set of floating point numbers d for which

$$p + d \doteq p$$

is the same as the set of d for which

$$1 + d \doteq 1.$$

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

We have chosen to stop when

$$4 + r \doteq 4.$$

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

```
r := (q/p)^2
```

in pythag with

```
r := -(q/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 := 1
r := z-1

while (r is numerically significant)

do
  s := r/(4+r)
  p := p+2*s*p
  r := r*(s/(1+2*s))^2
od
sqrt := p
```

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.

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*s*p
  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).
2. 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).
3. 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 mathematics 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, University of New Mexico, Albuquerque, New Mexico 87131.* Professor Morrison has been with the University of New Mexico since 1971. 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.