A Portable Fortran Program To Find the Euclidean Norm of a Vector

JAMES L. BLUE
Bell Laboratories

A successful portable version of a Fortran subprogram to find the Euclidean norm of an n-vector, $\| x \| = (\sum_{i=1}^{n} x_i^2)^{1/2}$, is described. Overflow and underflow are avoided. A program implementing the algorithm is included.

Key Words and Phrases: error bounds, Euclidean norm, machine constants, overflow, portability, underflow
CR Categories: 5.11, 5.14

1. INTRODUCTION

A set of Fortran subprograms for performing the basic operations of linear algebra [4, 5, 6] should include a subprogram to find the Euclidean norm of an n-vector, $\| x \| = (\sum_{i=1}^{n} x_i^2)^{1/2}$. Such a subprogram should be accurate and efficient, and should avoid all overflows and underflows.

The problem appears much easier than it is. Preliminary versions of the subprogram, by several authors, failed at least two of these requirements.

This paper describes a successful version which is also portable. All machine-dependent constants are combinations of the basic machine constants defined by Fox et al. [3]; therefore the programs are portable. A program incorporating the algorithm is included.

To avoid overflow, large x_i must be scaled down. Let R be the largest positive floating-point number representable on the computer being used. Then for any x_i such that $| x_i | > R^{1/2}$, x_i^2 will overflow, although $\| x \|$ may not overflow.

A simple way of avoiding overflow is the following. Let

$$x_{\max} = \max_{i=1,n} | x_i |.$$

Form

$$a = \sum_{i=1}^{n} (x_i/x_{\max})^2.$$

Then $\| x \| = x_{\max}\, a^{1/2}$. This procedure requires two passes over the data vector, which is unnecessarily slow, especially for long vectors on paged machines. (Tested

Author's address: Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974.

on the Honeywell 6000 in single precision, this procedure took 50 percent longer than the procedure to be discussed.) Furthermore, it does not avoid underflow. More complicated two-pass procedures can avoid underflow.

Let r be the smallest positive floating-point number representable on the machine being used. Then for any x_i such that $| x_i | < r^{1/2}$, x_i^2 will underflow, although $\| x \|$ does not underflow. If underflow is replaced by zero, then $\| x \|$ may be computed incorrectly, since the underflowing components may contribute to $\| x \|$. Even if the underflowing components are so small as not to contribute to $\| x \|$, it is desirable to avoid underflow. Thus small numbers must be scaled up in forming the partial sum of squares.

For medium-sized numbers, no scaling is necessary. For efficiency, it is desirable to avoid all unnecessary scaling and to make only one pass over the data.

2. AN ALGORITHM FOR CALCULATING THE EUCLIDEAN NORM OF AN n-VECTOR

All overflows and underflows may be avoided by employing the following algorithm. The algorithm requires constants b, B, s, and S, which are derived in Section 3; B and S are large positive numbers, and b and s are small positive numbers. N is the largest value of n for which the algorithm can be guaranteed, and ϵ is a measure of machine precision; N and ϵ are defined in Section 3. Three accumulators for partial sums of squares will be used. Only one pass over the x-vector is made.

```
if $n = 0$, set $\| x \| = 0$ and return.
if $n < 0$, set an error flag and stop.
if $n > N$, set an error flag and stop.
$a_{\text{sml}} = 0$; $a_{\text{med}} = 0$; $a_{\text{big}} = 0$
for $i = 1$ through $n$
  if $| x_i | > B$, $a_{\text{big}} \leftarrow a_{\text{big}} + (x_i/S)^2$
  else if $| x_i | < b$, $a_{\text{sml}} \leftarrow a_{\text{sml}} + (x_i/s)^2$
  else $a_{\text{med}} \leftarrow a_{\text{med}} + x_i^2$
if $a_{\text{big}}$ is nonzero
  if $a_{\text{big}}^{1/2} > R/S$, $\| x \| > R$ and overflow would occur. Set $\| x \| = R$, set an error flag, and return.
  if $a_{\text{med}}$ is nonzero
    $y_{\text{min}} = \min(a_{\text{med}}^{1/2}, Sa_{\text{big}}^{1/2})$
    $y_{\text{max}} = \max(a_{\text{med}}^{1/2}, Sa_{\text{big}}^{1/2})$
  else set $\| x \| = Sa_{\text{big}}^{1/2}$ and return
else if $a_{\text{sml}}$ is nonzero
  if $a_{\text{med}}$ is nonzero
    $y_{\text{min}} = \min(a_{\text{med}}^{1/2}, sa_{\text{sml}}^{1/2})$
    $y_{\text{max}} = \max(a_{\text{med}}^{1/2}, sa_{\text{sml}}^{1/2})$
  else set $\| x \| = sa_{\text{sml}}^{1/2}$ and return
else set $\| x \| = a_{\text{med}}^{1/2}$ and return.
if $y_{\text{min}} < \epsilon^{1/2} y_{\text{max}}$, set $\| x \| = y_{\text{max}}$.
else set $\| x \| = y_{\text{max}}(1 + (y_{\text{min}}/y_{\text{max}})^2)^{1/2}$
```

3. COMPUTER MODEL

To convert the algorithm of Section 2 into a portable Fortran program, we must define b, B, s, S, N, and ϵ in terms of machine-dependent parameters which are readily available. This requires some assumptions about the computer being used.

In addition, analysis of the correctness of the algorithm requires assumptions about the properties of the arithmetic done by the computer.

Various assumptions could be made, and b, B, s, and S could be defined in terms of N, ϵ, r, R, and β (the floating-point base). Then a proof of correctness would require several relations to hold among N, ϵ, r, R, and β. Other than these relations, only minimal information about the computer would be required.

A more satisfactory solution is obtained by starting from a detailed model of a computer. A useful model is that of Fox et al. [3].

Floating-Point Number Representation

The floating-point representation is defined by four integers: β, t, $e_{\min}$, and $e_{\max}$. Zero and all numbers of the form

$$\begin{aligned} &\pm\beta^e(m_1\beta^{-1} + \cdots + m_t\beta^{-t}), \quad 1 \le m_1 < \beta \\ &0 \le m_i < \beta, \quad i = 2, 3, \ldots, t, \\ &e_{\min} \le e \le e_{\max} \end{aligned}$$

are exactly representable. These numbers are called model numbers; they are a subset of the machine numbers. On some machines not all the machine numbers are model numbers. In particular, numbers that are kept in working registers may have extra bits in the mantissa.

The smallest and largest magnitude model numbers are easily found:

$$\begin{aligned} r &= \beta^{e_{\min}-1}, \\ R &= \beta^{e_{\max}}(1 - \beta^{-t}). \end{aligned}$$

A useful measure of machine precision is the largest relative spacing between model numbers, $\epsilon = \beta^{1-t}$. The largest n for which the algorithm can be proven depends on the arithmetical properties of the computer and on the relative values of $e_{\min}$ $e_{\max}$, and t. Under the assumptions of the theorem in Section 4, the largest safe n, is $N = \beta^{t-1} - 1$. (This may be larger than the largest integer representable on the computer.)

Floating-Point Arithmetic

The success of the algorithm could be assured if Wilkinson's assumptions [8] about floating-point arithmetic were made. Let fl(*expression*) be the value obtained when *expression* is evaluated in floating-point arithmetic on the computer in question. Let *op* denote any of the operations +, −, ×, or ÷. Then the usual assumption is that

$$(1 - \eta)(u \; op \; v) \le \mathrm{fl}(u \; op \; v) \le (1 + \eta)(u \; op \; v)$$

where $\eta = \epsilon$ or $\epsilon/2$; its value depends on the rounding properties of the computer and whether arithmetic is done using guard bits [8]. A similar assumption is needed about the square-root routine.

The standard floating-point error analysis could be used in deriving the proof of correctness and the error analysis of the algorithm. If the computer model of the previous section is used, alternative assumptions about the floating-point arithmetic [2] become attractive and simplify the proofs.

Suppose u and v are any machine numbers. Let

$$w = u \, op \, v,$$
$$\tilde{w} = \mathrm{fl}(u \, op \, v).$$

A reasonable model of the accuracy of floating-point arithmetic is the following [2], which is slightly weaker than perfect rounding:

(F1) If w is a model number, $\tilde{w} = w$.

(F2) Otherwise, if w_1 and w_2 are the two model numbers that bracket w, $w_1 \leq \tilde{w} \leq w_2$.

No assumptions are made about the result if $|w| < r$ or $|w| > R$. Properly designed arithmetic units obey these two assumptions, but guard bits are required. Since the maximum relative spacing between model numbers is ϵ, arithmetic units obeying (F1) and (F2) obey

(F3) $(1 - \epsilon)(u \, op \, v) < \mathrm{fl}(u \, op \, v) < (1 + \epsilon)(u \, op \, v)$,

which is the usual Wilkinson assumption for truncated arithmetic, with $\eta = \epsilon$.

An accuracy assumption is also required for the square-root routine; we assume $(1 - \epsilon)\sqrt{u} < \mathrm{fl}(\sqrt{u}) < (1 + \epsilon)\sqrt{u}$, which is easy to achieve on most machines.

Three simple lemmas will be stated without proof. They follow from (F1) and (F2), but not from (F3) alone. In the following, k is an integer, and u is any model number.

Lemma A. *A model number can be scaled by a power of the base without any error. If* $r \leq |u\beta^k| \leq R$, *then* $fl(u\beta^k) = u\beta^k$. *If* $r \leq |u/\beta^k| \leq R$, *then* $fl(u/\beta^k) = u/\beta^k$.

Lemma B. *If* $u \geq \beta^k$ *and* $r \leq \beta^{2k} \leq R$, *then* $fl(u^2) \geq \beta^{2k}$. *If* $u \leq \beta^k$ *and* $r \leq \beta^{2k} \leq R$, *then* $fl(u^2) \leq \beta^{2k}$.

Lemma C. *If* $u_i \leq \beta^k$, $i = 1, 2, \ldots, n$, *and if* $n \leq N$ *and* $n\beta^k \leq R$, *then*

$$fl\left[\sum_{i=1}^{n} u_i\right] \leq n\beta^k.$$

4. PROOF OF CORRECTNESS

We choose b, B, s, and S as follows. Denote $\lceil u \rceil$ as the smallest integer greater than or equal to u, and $\lfloor u \rfloor$ as the largest integer less than or equal to u.

$$b = \beta^{\lceil (e_{\min}-1)/2 \rceil}$$
$$B = \beta^{\lfloor (e_{\max}-t+1)/2 \rfloor}$$
$$s = \beta^{\lfloor (e_{\min}-1)/2 \rfloor}$$
$$S = \beta^{\lceil (e_{\max}+t-1)/2 \rceil}.$$

If $e_{\min}$ is odd, $b = s$; if $e_{\min}$ is even, $b = \beta s$. If $e_{\max} + t$ is odd, $S = \beta^{t-1}B$; if $e_{\max} + t$ is even, $S = \beta^t B$. (Examples are given in Table I.)

Table I
(SP = single precision; DP = double precision)

	Honeywell 6000/7000		IBM 360/370		CDC 6000/7000	
	SP	DP	SP	DP	SP	DP
β	2	2	16	16	2	2
t	27	63	6	14	48	96
$e_{\min}$	−127	−127	−64	−64	−974	−927
$e_{\max}$	127	127	63	63	1070	1070
$\log_2 b$	−64	−64	−128	−128	−487	−464
$\log_2 B$	50	32	116	100	511	487
$\log_2 s$	−64	−64	−132	−132	−488	−464
$\log_2 S$	77	95	136	152	559	583
$\log_2 \epsilon$	−26	−62	−20	−52	−47	−95
$\log_2(N+1)$	26	35	20	31	47	48

The proof of correctness requires three relations to hold:

$$e_{\min} \leq 1 - 2t \tag{1}$$

$$1 + t \leq e_{\max} \tag{2}$$

$$t \geq 2, \quad \begin{aligned} &\text{if } \beta \leq 4, t \geq 3 \\ &\text{if } \beta = 2, t \geq 5. \end{aligned} \tag{3}$$

For the computers[1] listed in [3], all the above hold. In fact, these relations should hold for any computer suitable for scientific work.

THEOREM. *Suppose the algorithm of Section 2 is implemented on the model computer of Section 3, for which* ($F1$) *and* ($F2$) *hold. Let b, B, s, and S be as given above, and let relations* (1), (2), *and* (3) *hold. Let x be an n-vector with $n \leq N$, with each component x_i, a model number. Then no overflow or underflow will be produced when the algorithm operates on x. If $fl(\| x \|) > R$, an error message will be produced; otherwise*

$$\| x \|(1 - \epsilon)^{6+n/2} < fl(\| x \|) < \| x \|(1 + \epsilon)^{6+n/2}.$$

The proof is based on a series of lemmas. The first two are necessary to establish correctness, even if roundoff, underflow, and overflow are neglected. The next five lemmas establish the absence of overflow and underflow. The final lemma bounds the error in any one of the three accumulators. The proofs of the lemmas require Lemmas A, B, and C and relations (1), (2), and (3), and will be omitted. The details may be found in [1].

LEMMA 1. *The three ranges are disjoint; $b < 1 < B$.*

LEMMA 2. *If a_{big} is nonzero, the contribution of a_{sml} to $\| x \|$ is negligible.*

[1] These are the Burroughs 6700 series, the CDC 6000/7000 series, the Honeywell 600/6000 series, the IBM 360/370 series, the PDP-10 and PDP-11 series, the SEL systems 85/86, the Univac 1100 series, and the Xerox Sigma 5/7/9 series.

LEMMA 3. *There is no overflow or underflow in calculating* a_{big}.

LEMMA 4. *There is no underflow or overflow in calculating* a_{med}.

LEMMA 5. *There is no overflow or underflow in calculating* a_{sml}.

LEMMA 6. *There is no underflow in computing* $\epsilon^{1/2} y_{max}$ *or* $(y_{min}/y_{max})^2$.

LEMMA 7. *There is no overflow in calculating* $y_{max}(1 + (y_{min}/y_{max})^2)^{1/2}$.

LEMMA 8. *Let* $\mathbf{u} = (u_1, u_2, \ldots, u_m)$ *be a vector of model numbers. Suppose* $r \leq fl(u_i^2) \leq R$ *for* $i = 1, 2, \ldots, m$. *If* $fl(\sum_{i=1}^{m} u_i^2)$ *does not overflow, then*

$$(1 - \epsilon)^m \sum_{i=1}^{n} u_i^2 \leq fl(\sum_{i=1}^{n} u_i^2) \leq (1 + \epsilon)^m \sum_{i=1}^{n} u_i^2.$$

PROOF OF THEOREM. The proof of the theorem now follows upon applying Lemma 8 to each of the three accumulators, with $u_i = x_i/s$ for a_{sml}, $u_i = x_i$ for a_{med}, and $u_i = x_i/S$ for a_{big}. Standard floating-point error analysis of the algorithm, with the use of Lemma A, then gives relative error bounds of $(1 + \epsilon)^{1+n/2}$ if only one accumulator contributes and $(1 + \epsilon)^{5+n/2}$ if two adjacent accumulators contribute. If a_{big} and a_{sml} are both nonzero, a_{sml} is ignored, which according to Lemma 2 could contribute another rounding error. Thus the relative error bound is either $(1 + \epsilon)^{2+n/2}$ or $(1 + \epsilon)^{6+n/2}$, depending on whether a_{med} is zero or nonzero.

5. IMPLEMENTATION

A portable implementation of the algorithm, x2norm, written in RATFOR [7], is given in the Appendix; the output of the RATFOR preprocessor is portable Fortran. The initialization of the needed machine-dependent constants is done by subprogram x2init; the remainder of the norm program is machine independent. A first-time switch is used so that x2init is not called every time x2norm is called. To make a version of x2norm for any specific machine, remove the call to x2init, calculate the needed constants according to the prescription in Section 4, and replace the data statements in x2norm. Floating-point constants should be done in binary, octal, or hexadecimal, (whichever is appropriate) to ensure that b, B, s, and S are exactly powers of β.

In this implementation a portable Fortran error-handling facility [3] is used. Execution stops after either of the fatal errors $n < 0$ or $n > N$; the user may elect to continue after the nonfatal error $\| x \| > R$.

For a portable implementation of the initialization routine, portable Fortran machine constant programs [3] are used.

APPENDIX. PROGRAM LISTING

```
real function x2norm(n, x)

# Calculate 2-norm of x vector.
# Avoid all overflows and underflows

integer n,nmax,j
real x(n),ax,abig,amed,asml,b1,b2,s1m,s2m,relerr,overfl,rbig

# This portable version of x2norm uses nmax as a first-time switch and
# calls x2init to calculate needed machine-dependent constants.
# x2init normally is executed only once, to save overhead.
# This is a non-standard, but safe, usage. If x2init
# were executed more than once, as might happen if overlays
# were used, additional overhead would be incurred, but no
# errors would occur.

# For any specific machine, the data statements can be revised
# and the call to x2init removed

data b1,b2,s1m,s2m,overfl,rbig,relerr /7*0.0/
data nmax/0/

if (nmax<=0)
  call x2init(nmax,b1,b2,s1m,s2m,overfl,rbig, relerr)

if (n==0)
  {x2norm=0.0; return}
if (n<0)
   call seterr(' x2norm - n .lt. 0',18,1,2)        # fatal error
if (n>nmax)
   call seterr(' x2norm - n too large',21,2,2)   # fatal error

asml=0.0
amed=0.0
abig=0.0
do j=1,n
  {ax=abs(x(j))
        if (ax>b2) abig=abig+(ax*s2m)**2
   else if (ax<b1) asml=asml+(ax*s1m)**2
   else            amed=amed+ax**2
   }
if (abig>0.0)
  {abig=sqrt(abig)
   if (abig>overfl)
     {x2norm=rbig
      call seterr(' x2norm - overflow',18,3,1)   # non-fatal error
      return
     }
   if (amed>0.0)
     {abig=abig/s2m
      amed=sqrt(amed)
     }
   else
     {x2norm=abig/s2m; return}
  }
else if (asml>0.0)
  {if (amed>0.0)
     {abig=sqrt(amed)
      amed=sqrt(asml)/s1m
     }
   else
     {x2norm=sqrt(asml)/s1m; return}
  }
else
  {x2norm=sqrt(amed); return}   # the standard path

asml=amin1(abig,amed)
abig=amax1(abig,amed)
if (asml<=abig*relerr)
```

```
   x2norm = abig
else
  x2norm = abig*sqrt(1.0+ (asml/abig)**2)
return
end

subroutine x2init(nmax,b1,b2,s1m,s2m,overfl,rbig,relerr)

integer ilmach,nmax,iout,nbig,ibeta,it,iemin,iemax,iexp
real rlmach,bexp,abig,b1,b2,s1m,s2m,eps,relerr,overfl,rbig

# This program calculates the machine-dependent constants
#       b1, b2, s1m, s2m, relerr overfl, nmax
# from the "basic" machine-dependent numbers
#       nbig, ibeta, it, iemin, iemax, rbig.

# The following define the basic machine-dependent constants.
# For portability, the PORT subprograms "ilmach" and "rlmach"
# are used  For any specific computer, each of the assignment
# statements can be replaced

iout  =ilmach(4)   # standard output file for error messages
nbig  =ilmach(9)   # largest integer
ibeta =ilmach(10)  # base for floating-point numbers
it    =ilmach(11)  # number of base-beta digits in mantissa
iemin =ilmach(12)  # minimum exponent
iemax =ilmach(13)  # maximum exponent
rbig  =rlmach(2)   # largest floating-point number

# Check the basic machine-dependent constants.
if (iemin>1-2*it | 1+it>iemax | (it= =2 & ıbeta<5) |
     (it< =4 & ibeta< =3) | it<2)
   {write(iout,1)
    1 format(' x2norm - the algorithm cannot be guaranteed',
       ' on this computer')
   }

ıexp  = -((1-iemin)/2)
b1    =bexp(ibeta, iexp)            # lower boundary of midrange
iexp  =(iemax+1-it)/2
b2    =bexp(ibeta,iexp)             # upper boundary of midrange

iexp  =(2-iemin)/2
s1m   =bexp(ibeta,iexp)             # scaling factor for lower range
iexp  = -((iemax+it)/2)
s2m   =bexp(ibeta,iexp)             # scaling factor for upper range

overfl=rbig*s2m                     # overflow boundary for abig
eps   =bexp(ibeta,1-it)
relerr=sqrt(eps)                    # tolerance for neglecting asml
abig  =1.0/eps-1.0
if (float(nbig)>abig) nmax=abig     # largest safe n
else                  nmax=nbig

return
end
real function bexp(ibeta,iexp)

# bexp=ibeta**iexp by binary expansion of iexp,
# exact if ibeta is the machine base
```

```
integer ibeta,iexp,n
real tbeta

tbeta = float(ibeta)
bexp = 1.0
n = iexp
if (n<0)
  {n = -n
   tbeta = 1.0/tbeta
  }
repeat
  {if (mod(n,2)!=0) bexp = bexp*tbeta
   n = n/2
   if (n==0) return
   tbeta = tbeta*tbeta
  }
return
end
```

ACKNOWLEDGMENT

A.D. Hall originally suggested the idea of three accumulators. W.S. Brown, A.D. Hall, N.L. Schryer, and D.D. Warner provided useful criticisms of various drafts of the manuscript.

REFERENCES

1. BLUE, J.L. A portable Fortran program to find the Euclidean norm of a vector Comptng· Sci Tech. Rep. 45, Bell Laboratories, Murray Hill, N J , July 1976
2. BROWN, W.S. A realistic model of floating point computation In *Mathematical Software III*, J.R. Rice, Ed., Academic Press, New York, 1977.
3. FOX, P.A., HALL, A D., AND SCHRYER, N L. The PORT mathematical subroutine library. Comptng. Sci. Tech. Rep. 47, Bell Laboratories, Murray Hill, N J., Sept. 1976.
4. HANSON, R.J., KROGH, F T., AND LAWSON, C.L. A proposal for standard linear algebra subprograms. Tech. Memo. 33-660, Jet Propulsion Lab , Pasadena, Calif., Nov. 1973.
5. LAWSON, C L Standardization of Fortran callable subprograms for basic linear algebra Proc. Math. Software II, May 1974, p. 261.
6. LAWSON, C.L., HANSON, R.J., KINCAID, D., AND KROGH, F.T Basic linear algebra subprograms for Fortran usage. May 1976 (unpublished).
7. KERNIGHAN, B W. RATFOR—a preprocessor for a rational Fortran *Software—Practice and Experience 5* (Oct. 1975), 395–406
8 WILKINSON, J.H. *Rounding Errors in Algebraic Processes*. Prentice-Hall, Englewood Cliffs, N.J , 1963.

Received July 1976, revised February 1977