A C Program for Solving Banded, Symetric, Positive Definite Systems

by

Richard L. Branham, Jr.*

Abstract

Banded, symmetric, positive definite systems occur frequently. Although they can be solved by any program that incorporates Cholesky decomposition, or even LU decomposition, for greatest efficiency account should taken of the specialized nature of banded, symmetric, positive definite systems: elements outside the band need not be stored; because of symmetry nearly half of the non–null elements are superfluous and likewise need not be stored; positive definiteness obviates row and column interchanging. A C program, employing square root–free Cholesky decomposition, is given for solving such systems. For sparse matrix processing C offers advantages over FORTRAN: when the two–dimensional array is represented by a vector, C's conditional operator permits a compact way to index elements in the vector; the vector itself may be allocated dynamically and stores no null elements; divisions by powers of two may be replaced by bit shifts to the right; and preprocessor macros, substituting in–line code for subroutine calls, implement functions. An example, a 40 x 40 pentadiagonal system, illustrates use of the program. In this example the C code for banded, symmetric, positive definite systems is four hundred times faster than code for general Cholesky decomposition. Although accumulation of matrix inner products to high precision is usually not required for banded systems, a function subroutine is nevertheless presented to sum floating-point numbers with cascaded accumulators.

I. Introduction.

Despite their specialized nature banded, symmetric, positive definite linear systems occur frequently. When one replaces a partial differential equation by a finite difference approximation, for example, the system assumes this form. Any method for linear systems, such as LU decomposition, could be used to solve the equations but for greatest efficiency one should take advantage of techniques that exploit the nature of banded, symmetric, positive definite matrices. Algorithms have been published for this purpose, such as Golub and Van Loan's ([1]) pseudocode and Martin and Wilkinson's ([2]) ALGOL code. This paper furnishes another algorithm that, I believe, incorporates characteristic features.

The algorithm uses C, contributing to the growing literature on scientific algorithms in this language; C offers attractive features for sparse matrix processing not available in FORTRAN. The algorithm uses a vector, whose size is determined at execution time and stores no null elements, to represent the matrix. For greatest efficiency square root–free Cholesky decomposition solves the system.

II. Cholesky Decomposition.

Given a symmetric, positive definite linear system,

$$Ax = b \qquad (1),$$

* Centro Regional de Investigaciones, Cientificas y Tecnológicas, C.C. 131, 5500 - Mendoza, Argentina

where A is $n \times n$ and x and b n–vectors, the solution is calculated by Cholesky decompositon:

$$\begin{aligned} A &= S^T S, \\ y &= Sx, \\ S^T y &= b, \end{aligned} \tag{2}$$

where S is upper triangular. Because A is positive definite, row and column interchanges, required in LU decomposition, become unnecessary. An operation count, where an operation refers to a multiplication or a division, reveals $n^{3/6}$ operations to calculate S and $n^{2/2}$ operations to solve each triangular system for y and then x; n square roots must also be calculated. See Golub and Van Loan ([1]) for details.

One can avoid the calculation of square roots by expressing the decomposition as:

$$\begin{aligned} A &= G^T DG, \\ y &= DGx, \\ G^T y &= b, \end{aligned} \tag{3}$$

where G is upper triangular with unit principal diagonal and D is diagonal. The operation count remains the same, but square roots become unnecessary.

III. Banded Cholesky Decomposition.

Define the bandwidth of a matrix as $2m + 1$, where m is the number of superdiagonals or subdiagonals. Eq. (2) or Eq. (3) could be used to solve a banded system, but the null elements outside the band should be taken in account. Just checking for null elements – in reality elements of the order of the machine ϵ of the arithmetic being used – and performing no operations with them is insufficient; the algorithm should do arithmetic only within the band.

To store a symmetric band matrix some, such as Golub and Van Loan ([1]) and Martin and Wilkinson ([2]), use an array of dimension $(m + l)xn$. I prefer a vector with proper indexing to locate the element. The matrix

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & \\ a_{21} & a_{22} & a_{23} & a_{24} & 0 & \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & \cdots \\ 0 & a_{42} & a_{43} & a_{44} & a_{45} & \\ 0 & 0 & a_{53} & a_{54} & a_{55} & \\ 0 & 0 & 0 & a_{64} & a_{65} & \\ 0 & 0 & 0 & 0 & a_{75} & \\ \vdots & \vdots & \vdots & \vdots & \vdots & \end{pmatrix}$$

is stored in column order as the vector

$$(a_{11} \quad a_{12} \quad a_{22} \quad a_{13} \quad a_{23} \quad a_{33} \quad a_{24} \quad a_{34} \quad a_{44} \quad a_{35} \quad a_{45} \quad a_{55} \quad \ldots)$$

With this representation we need

$$n(n+1)/2 - \sum_{i=m+1}^{n-1} (i-m) = (m+1)n - \frac{m(m+1)}{2}$$

locations, for a saving over the two dimensional array, with its $(m+l)n$ requirement, of $1 - m/2n$. For $m \ll n$ little is gained, but as m increases the saving becomes significant, ten per cent for $m = 0.2n$. (Because banded Cholesky decomposition involves more overhead than ordinary Cholesky decomposition, it should not be used for large bandwidths. The exact cutoff depends on coding details and the presence of zeros within the band, bitt my experience suggests $b\mathrm{approx}0.3n$ a practical guide.)

How do we locate a specific element in the vector? Up to column $m+2$ the mapping from the indices i, j to a single index k is the same as that for an upper triangular matrix. When an $n \times n$ matrix is stored by column the mapping to find element a_{ij} is $k = n(j-1) + i$. When the matrix is upper triangular we must allow for the missing $n-1$ elements in the first column, $n-2$ in the second, and so forth up to the $n-th$ column, which is complete. We find for the mapping, therefore, $k = n(j-1) + i - \sum_{k=1}^{j-1}(n-k) = j(j-1)/2 + i$. To allow for the missing elements outside the band when we exceed column $m+1$ we have

$$k = j(j-l)/2 + i \sum_{k=m+2}^{j} [k - (m+1)] = mj + i - m(m+l)/2.$$

The mapping function k is thus multivalued. Up to and including column $m+l \quad k = j(j-1)/2 + i$ and beyond $k = mj + i - m(m+l)/2$. C easily incorporates this mapping with the conditional operator,

$$k = (j <= m+l) ? j * (j-1)/2 + i : m * j + i - m * (m+1)/2. \tag{4}$$

In FORTRAN one has to use the IF-THEN-ELSE construction

```
IF        (J.LE.M + 1)THEN
              K + J * (J - 1)/2 + I
ELSE
              K = M * J + I - M * (M + 1)/2
ENDIF
```

C's notation is much more compact.

When A of Eq. (3) is $n \times n$ one may easily derive formulas for the square root–free Cholesky decomposition:

$$d_i = A_{ii} - \sum_{k'=1}^{i-1} d_{k'} G_{k'i}^2, \quad i = 1, \ldots, n \tag{5a}$$

$$G_{ij} = (A_{ij} - \sum_{i-1}^{k'=1} \quad d_{k'} G_{k'i} G_{k'j})/d_i, j = i+1, \ldots, n, \quad i = 1, \ldots n. \tag{5b}$$

To avoid accessing elements outside the band, for banded systems k' should start at $\max(1, i-m)$ rather than 1 in Eq. (5a) and at $\max(1, j-m)$ in Eq. (5b); j should run up to $\min(i+m, n)$. These limits may be deduced upon our setting elements outside the band in Eq. (5) to zero.

If Eq. (5) were coded as written each sum over k' would involve two multiplications instead of one as with ordinary Cholesky decomposition. Because the extra multiplication is embedded, as Eq. (5b) shows, in a loop nested three deep, the operation count for the decomposition would be approximately doubled. It is preferable to sacrifice a little space to gain efficiency. If one assigns an array of dimension $n-1$ to hold the products $d_{k'} G_{k'i}$ in Eq. (5a), the same products may be used in Eq. (5b). Because one multiplication has been absorbed in a two–deep loop, the three–deep loop requires no additional multiplication, and the operation count remains about the same as with ordinary Cholesky decomposition.

Given that the operation count for both versions of the decomposition remains the same, but the ordinary Cholesky decomposition requires the calculation of n square roots and the square root–free version none, the latter will be more efficient. For a dense $n \times n$ matrix the saving is negligible, but becomes significant for $m \ll n$.

IV. Banded Cholesky Decomposition in C.

Although Eq. (4) provides a notationally elegant way to express the mapping, one must wonder whether k's calculation may be inefficient. To map the two–dimensional array $m \times n$, stored in row order, to the linear storage required by computer memory, C uses $k = ni + j$ for an operation count of one multiplication; the addition, being more rapid, can be neglected. Eq. (4) involves a comparison and then either a multiplication and a division or two multiplications and a division, two or three operations.

But the situation admits improvement. The term $m(m+1)/2$ can be calculated and stored as a constant as soon as the bandwidth is specified. $m+1$ may likewise be stored as a constant, and the integer comparison $j <= m+1$ will be fast. On binary computers the division by two in the term $j(j-1)/2$ may be replaced by a right shift of one bit; shifting is more rapid than division. In C we replace the term $j*(j-1)/2+i$ by $(j*(j-1) \gg 1)+i$. The extra parentheses are necessary because the shift operator has lower priority than additions and subtractions. With these modifications one multiplication suffices to calculate k.

By using a vector to store the elements of a band matrix we obviate having to reserve space for any null elements. But such an advantage would hardly be practical if we had to redimension the vector, necesitating recompilation, for every run of the program. Fortunately, C's dynamic allocation function malloc() permits avoiding the difficulty. malloc(X) obtains X bytes of memory (on byte addressable computers) and assigns the address of the beginning of the block of X bytes to a pointer. If we are working in double–precision, eight bytes on most computers, we reserve space for the vector as malloc$(((m + l) * n - m * (m + l)/2) * 8)$ and pass the address to a pointer; it is unnecessary to store any null elements.

But a slight inconvenience presents itself. In C the name of an array is a pointer. Having obtained space by malloc() and assigned an address to a pointer – suppose that the pointer has been declared as double $*$ a – we can treat the pointer as an array, for example a (5). But in C arrays start with index 0, not 1 as in FORTRAN. When working with vectors it is more convenient to have their indices run from 1 to n rather than 0 to $n - 1$. As Press et al. ([3]) mention the difficulty can be circumvented by subtracting one from the pointer (one unit of storage of whatever class the pointer has been declared as). We do not use a [0] and a [1] now coincides with the beginning of the block allocated by malloc().

For work with banded systems C possesses the advantage over FORTRAN that functions like $\max(1, j - m)$ can be coded as preprocessor macros rather than subroutine calls. Upon expansion by the preprocessor the macro generates in–line code and avoids the overhead of a subroutine call, a significant saving when the function is embedded in loops, although we pay the price of more in–line code.

Table I gives a C program, incorporating the ideas sketched in this section and Sec. III, for solving a banded, symmetric, positive definite system. The preprocessor definitions $\mathrm{MAX}(X, y)$ and $\mathrm{MIN}(X, y)$ implement the max and min functions. Storage for the matrix of coeficients A, represented as a vector and using the mapping Eq. (4), the solution vector x, the ancillary vector y, and array hold to contain the products $d_{k\prime}G_{\prime}i$ is allocated dynamically. For greater flexibility the program uses the size of operator to determine the size of a double–precision number rather than to assume that double–precision occupies eight bytes. The diagonal matrix D of Eq. (3) is unnecessary because the $d_i\prime$s usurp the diagonal of G in place of the unneeded unit elements. But the program does assume that the computer is binary, the base by far the most frequent in scientific computing, and replaces divisions by two in the mapping Eq. (4) by shifts to the right of one bit. The program is considerably more compact than an equivalent FORTRAN program, ninety–seven lines of code, excluding comments, compared with 150 lines for the FORTRAN program.

To have some idea of the program's operation count, I performed an analysis of the multiplications and divisions within loops. The presence of the max and min functions complicates the analysis, but for the situation $m \ll n$ we can take for the quantity $\mathrm{MAX}(1, i - m)$, for example, simply $i - m$. Upon doing this I find that

$$\sum_{i=2}^{m+1} 2 + \sum_{i=2}^{n} \left(\textstyle\sum_{k\prime=i-m}^{i-1} 2 + \sum_{j=i+1}^{i+m} \sum_{k\prime=j-m}^{i-1} 1 \right) + \sum_{i=2}^{n} \sum_{k\prime=i-m} 1 + \sum_{i=1}^{n-1} \sum_{k\prime=i+1}^{i-m} 1 + \sum_{i=1}^{n} 1$$

operations are required. Expanding the sums and omitting terms containing only m, by assumption much smaller than n, I arrive at

$$m^2n/2 + 7mn/2 + n$$

operations.

The operation count shows the efficiency of banded Cholesky decomposition over its unbanded cousin. For small m the cubic term $m^2n/2$ is much less than the cubic $n^3/6$ term of general Cholesky decomposition. To solve a tridiagonal system, for example, necesitates only $5n$ operations, an $0(n)$ rather than $0(n^3)$ algorithm.

Because of the significantly reduced operation count, accumulation of vector scalar products and matrix inner products to high precision by use of extended precision or cascaded accumulators – see Branham ([4]) for a discussion of the latter – should be unnecessary. There may, nevertheless, arise occasions, perhaps for poorly conditioned A and bandwidth a significant fraction of n, when it would be beneficial to use higher precision. Rather than leave the matter undiscussed I present in Table II a C function to implement cascaded accumulators. Because cascaded accumulators are machine dependent, what is given is specific to the VAX series of computers. See my previous publication for an explanation of how cascaded accumulators work.

To use the function one must define an additional array – call it temp – allocated and dimensioned the same way as hold. Then replace a sum statement, for example,

$$\text{sum+} = \text{hold}[krp] * a[kj];$$

within the

$$\text{for}(kpr = \text{MAX}(1, j-m) : kpr <= i-1; ++kpr)$$

loop by

$$\text{temp}[kpr] = \text{hold}[kpr] * a[kj];$$

within the loop followed by

$$\text{cascade}(\text{temp}, n);$$

outside of the loop. temp should be initialized to zero before a call to cascade.

V. An Example.

I have chosen the same example presented in Martin and Wilkinson ([2]),

$$\begin{pmatrix} 5 & -4 & 1 & 0 & 0 & & & & \\ -4 & 6 & -4 & 1 & 0 & & & & \\ 1 & -4 & 6 & -4 & 1 & & & & \\ 0 & 1 & -4 & 0 & 0 & \dots & & & \\ & & & & 1 & -4 & 6 & -4 & 1 \\ & & & & 0 & 1 & -4 & 6 & -4 \\ & & & & 0 & 0 & 1 & -4 & 5 \end{pmatrix} \begin{pmatrix} X_1 \\ X_2 \\ X_3 \\ \vdots \\ X_{38} \\ X_{39} \\ X_{40} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0, \end{pmatrix} \quad (6)$$

because the pentadiagonal system, $n = 40, m = 2$, is less trivial than a tridiagonal system – double the operation count – and the condition number of the matrix, $5 \cdot 10^5$ in the spectral norm, moderate for double–precision, high single–precision.

Table I's program calculates the solution, given in Table III, in $0^S.01$. For comparison a program for general Cholesky decomposition needs $12^S.77$; if the program is modified to check for elements of the order of the machine epsilon and perform no operations on them it still takes $3^S.86$. The banded version of Cholesky decomposition for pentadiagonal systems is four hundred times faster than tailored code for general Cholesky decomposition.

To check the precision of the solution I compared Table III's with an exact solution, where "exact" means a solution calculated in FORTRAN with extended precision, thirty–three decimal digits. (VAX C does not implement extended precision, one of the few instances where it compares unfavorably with FORTRAN.) The RMS difference between the two was 8.10^{-11}, the maximum difference $1 \cdot 10^{-10}$, and the minimum $9 \cdot 10^{-12}$. Given the matrix's condition number these differences are reasonable. Employment of cascaded accumulators would undoubtedly decrease the magnitude of the differences, but hardly seems necessary in this example.

VI. Conclusions.

Banded Cholesky decomposition, particularly the square root–free version, is significantly faster than general Cholesky decomposition and involves significantly less storage. C, a powerful and flexible language, offers numerous advantages over FORTRAN for sparse matrix processing. Among there are: the conditional operator, notationally elegant and compact, to map double indices i, j of a matrix to a single index k for a vector representing the matrix; the shift operator to replace a division by two by a shift of one bit to, the right; dynamic allocation of a vector, containing no null elements, to represent the matrix; and use of preprocessor macros, substituting in–line code for subroutine calls, to implement functions.

References.

[1] Golub, G.H. and C.F. Van Loan, *Matrix Computations* (Johns Hopkins U. Press, Baltimore, 1983), Sec.5.3.

[2] Martin, R.S. and J.H. Wilkinson, Num. Math. **7**, 355. (1965).

[3] Press, W.H., B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling, *Numerical Recipes in C* (Cambridge U. Press, Cambridge, 1988), Sec. 1.2.

[4] Branham, R.L. Jr., *Computers in Physics* 3(3), 42 (1989).

```
/*                                                                    */
/*                                                                    */
/* This program forms and solves banded, symmetric, positive definite sys- */
/* tems by square root-free Cholesky decomposition:                   */
/*                                                                    */
/*                        A*X=B => A=GTRANS*D*G,                      */
/*                                                                    */
/*                     Y=D*G*X, GTRANS*Y=B, G*X=Y/D.                  */
/*                                                                    */
/* The nxn banded, symmetric, positive definite matrix A is stored in */
/* compressed mode, using an array of size (m+1)*n-m*(m+1)/2, where  m is */
/* the bandwidth. Two auxiliary vectors of size n are needed for the  vec- */
/* tors x and y. Elements in the array are indexed by the mapping: if */
/* j<=m+1 then k=j*(j-1)/2+i; else k=m*j+i-m*(m+1)/2. The Cholesky factor */
/* g replaces the matrix a. A vector hold is used to contain certain in- */
/* termediate elements so they need not be calculated twice. (For great- */
/* est efficiency divisions by 2 in the mapping function are replaced by */
/* a shift to the right by one bit, a technique that works on binary com- */
/* puters.)                                                           */
/*                                                                    */
/*                                                                    */
#include math
#include stdio
#define MAX(x,y) ((x)>(y)) ? (x) : (y)
#define MIN(x,y) ((x)<(y)) ? (x) : (y)
main()
{
   FILE *in;
   double *a,*x,*y,*hold,sqrt(),sum;
   char *gets(),filename[40];
   int i,j,k,kk,ii,ki,ik,kj,ij,ic,m,n,nn,ulim,llim,mpl1,kpr;
   printf("What is the name of the file?\n");
   gets(filename);
   in=fopen(filename,"r");
   printf("How many unknowns are there?\n");
   scanf("%d",&n);
   printf("What is the bandwidth?\n");
   scanf("%d",&m);
   ic=m*(m+1)>>1;
   mpl1=m+1;
   a=(double *)malloc(((mpl1)*n-ic)*sizeof(double))-1; /* Dynamic allocation  */
   x=(double *)malloc(n*sizeof(double))-1;             /* of vectors a, x, y, */
   y=(double *)malloc(n*sizeof(double))-1;             /* and hold. */
   hold=(double *)malloc(n*sizeof(double))-1;
   for (j=1;j<=n;++j)
   {
      for (i=MAX(1,j-m);i<=j;++i)
      {
         k= (j<=mpl1) ? (j*(j-1)>>1)+i : m*j+i-ic;
         fscanf(in,"%f",&a[k]);
      }
      x[j]=0.0;
   }
   for (i=1;i<=n;++i)
     fscanf(in,"%f",&y[i]);
```

Table 1. C Program for Solving Band, Symmetric, Positive Definite Systems

```
/*                                                                    */
/* Decompose the banded normal equations. Cholesky factor s replaces a.  */
/*                                                                    */
   for (j=2;j<=mpl1;++j)
      a[(j*(j-1)>>1)+1]/=a[1];
   for (i=2;i<=n;++i)
   {
      sum=0.0;
      for (kpr=MAX(1,i-m);kpr<=i-1;++kpr)
      {
         ki= (i<=mpl1) ? (i*(i-1)>>1)+kpr : m*i+kpr-ic;
         kk= (i<=mpl1) ? (kpr*(kpr-1)>>1)+kpr : m*kpr+kpr-ic;
        hold[kpr]=a[kk]*a[ki];

         sum+=hold[kpr]*a[ki];
      }
      ii= (i<=mpl1) ? (i*(i-1)>>1)+i : m*i+i-ic;
      a[ii]-=sum;
      ulim=MIN(i+m,n);
      for (j=i+1;j<=ulim;++j)
      {
         sum=0.0;
         for (kpr=MAX(1,j-m);kpr<=i-1;++kpr)
         {
            ki= (j<=mpl1) ? (i*(i-1)>>1)+kpr : m*i+kpr-ic;
            kj= (j<=mpl1) ? (j*(j-1)>>1)+kpr : m*j+kpr-ic;
            sum+=hold[kpr]*a[kj];
         }
         ij= (j<=mpl1) ? (j*(j-1)>>1)+i : m*j+i-ic;
         ii= (j<=mpl1) ? (i*(i-1)>>1)+i : m*i+i-ic;
         a[ij]=(a[ij]-sum)/a[ii];
      }
   }
/*                                                                    */
/* First stage of solution of banded normal equations. strans*y=d. Solve  */
/* for y.                                                             */
/*                                                                    */
   for (i=2;i<=n;++i)
   {
      sum=0.0;
      llim=MAX(1,i-m);
      for (kpr=llim;kpr<=i-1;++kpr)
      {
         ik= (i<=mpl1) ? (i*(i-1)>>1)+kpr : m*i+kpr-ic;
         sum+=a[ik]*y[kpr];
      }
      ii= (i<=mpl1) ? (i*(i-1)>>1)+i : m*i+i-ic;
      y[i]-=sum;
   }
```

Table I. Continued

```
/*                                                                          */
/* Second stage of solution of banded normal equations, s*x=y. Solve for x.*/
/*                                                                          */
   for (i=1;i<=n;++i)
   {
      ii= (i>=mpl1) ? m*i+i-ic : (i*(i-1)>>1)+i;
      y[i]/=a[ii];
   }
   x[n]=y[n];
   for (i=n-1;i>=1;--i)
   {
      sum=0.0;
      ulim=MIN(i+m,n);
      for (kpr=i+1;kpr<=ulim;++kpr)
      {
         ik= (kpr<=mpl1) ? (kpr*(kpr-1)>>1)+i : m*kpr+i-ic;
         sum+=a[ik]*x[kpr];
      }
      ii= (i<=mpl1) ? (i*(i-1)>>1)+i : m*i+i-ic;
      x[i]=y[i]-sum;
   }
/*                                                                     */
/*                          Print the solution.                        */
/*                                                                     */
   printf("The solution vector:\n");
   for (i=1;i<=n;++i)
      printf(" %22.15e\n",x[i]);
}
```

Table I. Continued

```
/*                                                                    */
/*                                                                    */
/* A C function for implementing cascaded accumulators. The code     */
/* is machine dependent. What is given here works for the VAX ser-   */
/* ies of computers. A double-precision floating point number y      */
/* (declared within a union so that y shares the same space as the   */
/* integer array i[2]) is divided into two parts such that y=a1+a2   */
/* and both a1 and a2 end in nonsignificant zeros. MASK1 is used     */
/* to isolate the exponents of a1 and a2.                            */
/*                                                                    */
/*                                                                    */
#define MASK1 32640L
double cascade(x,n)
double x[];
int n;
{
   double s,*accum,y,a1,a2;
   int nu=64;
   int i;
   unsigned int iexp1,iexp2;
   union                          /* Both y and i[2] share same space */
   {
      unsigned i[2];
      double y;
   } num;
   accum=(double *)malloc(sizeof(double))-1;  /* Dynamic allocation */
   for(i=1;i<=nu;++i)              /* Initialize accumulators to zero */
      accum[i]=0.0;
   for(i=1;i<=n;++i)
   {
      num.y=x[i];
      num.i[1]=0;         /* Set least significant bits of y to zero */
      a1=num.y;
      a2=x[i]-a1;           /* Set insignificant bits of a2 to zero */
      num.y=a1;
      iexp1=num.i[0]&MASK1;                          /* Isolate exponent */
      iexp1=iexp1>>9;        /* Shift exponent to right so that its */
      accum[iexp1]+=a1;      /* numerical value (0-255) corresponds */
      num.y=a2;                    /* to integer with same value. Sum in- */

      iexp2=num.i[0]&MASK1; /* to corresponding accumulator and do */
      iexp2=iexp2>>9;        /* same for exponenet of a2.            */
      accum[iexp2]+=a2;
   }
   s=0.0;                      /* Sum accumulators in decreasing order */
   for(i=nu;i>=1;--i)
      s+=accum[i];
   return(s);
}
```

Table II. C Function to Implement Cascaded Accumulators on VAX Computers.

1	1.317073170732582e+01	21	1.041463414635292e+02
2	2.536585365855403e+01	22	1.019512195123090e+02
3	3.660975609758700e+01	23	9.929268292694179e+01
4	4.692682926832713e+01	24	9.619512195132999e+01
5	5.634146341467680e+01	25	9.268292682937607e+01
6	6.487804878053841e+01	26	8.878048780498248e+01
7	7.256097560981434e+01	27	8.451219512205170e+01
8	7.941463414640699e+01	28	7.990243902448618e+01
9	8.546341463421878e+01	29	7.497560975618840e+01
10	9.073170731715209e+01	30	6.975609756106083e+01
11	9.524390243910936e+01	31	6.426829268300595e+01
12	9.902439024399299e+01	32	5.853658536592622e+01
13	1.020975609757054e+02	33	5.258536585372413e+01
14	1.044878048781489e+02	34	4.643902439030214e+01
15	1.062195121952261e+02	35	4.012195121956275e+01
16	1.073170731708392e+02	36	3.365853658540843e+01
17	1.078048780488908e+02	37	2.707317073174167e+01
18	1.077073170732831e+02	38	2.039024390246497e+01
19	1.070487804879187e+02	39	1.363414634148079e+01
20	1.058536585366999e+02	40	6.829268292691641e+00

Table III. Solution of Eq.(6).