An efficient implementation of LU decomposition in C

ALAN MEYER

Computer Science Department, Washington State University, Pullman, Wa. 99164–1210, USA.

An implementation of the well-known LU decomposition method for solving systems of linear equations is presented. Operation counts typically stated for this method are derived from a theoretical analysis of the algorithm and ignore the practical aspects of the implementation. The overhead associated with assessing elements of the two-dimensional coefficient array are explored herein, and seen to be substantial. This overhead is incorporated into the analysis of LU decomposition and new operation counts are developed. By exploiting the addressing capabilities of C the cost of array access is significantly reduced, resulting in an efficient implementation. The techniques employed here can be applied to a wide variety of C programs which utilize multi-dimensional arrays.

Keywords: LU decomposition, array access, compiler, C, pointers

1. INTRODUCTION

LU decomposition, a well-known method for solving systems of linear equations, is widely used since it is easily understood and well suited for computer implementation. This paper presents an implementation of the LU method with partial pivoting.[1] The implementation of this particular method was used to explore the overhead of using two-dimensional arrays and to study how this expense might be reduced. The technique described here can be readily applied to the implementation of other algorithms.

The LU method is composed of two parts: the decomposition step and the solve step. A straightforward analysis of the algorithm can provide a count of the number of arithmetic operations required for each of these two steps. By considering the expense incurred from accessing elements of the coefficient array one sees that, in practice, the study of the algorithm alone does not provide an accurate count. For example, the decomposition step theoretically requires

$$\frac{n^3}{3}-\frac{n^2}{2}+\frac{n}{6}$$

multiplies, but when one considers the two-dimensional array overhead, this count is seen to be n^3+5n^2-6n. While these counts are both $O(n^3)$, the coefficients differ substantially. Aware of the expense of array access, the programmer can include its reduction as one of the implementation goals and the C programming language provides features whereby this cost may be effectively reduced.

The second section of this paper discusses two-dimensional array overhead and how it may be reduced in C. The third section presents an implementation of LU decomposition which utilizes this reduction technique. A listing of this implementation is given in the Appendix.

The details of array overhead are based on the C compiler running under Unix* bsd 4.3 on a VAX 11/750. The results are independent of the optimizer. Although optimization did improve some portions of the code, the cost of accessing elements of two-dimensional arrays was not affected. While other compilers and optimizers may perform differently, these results are believed to be typical.

2. TWO-DIMENSIONAL ARRAYS

Most high level languages support multi-dimensional arrays as a standard data type. By relying on the compiler, the programmer need not dwell on the lower level details of array access. However, it is sometimes worthwhile to consider these details in order to quantify the overhead involved.

In C, arrays are stored internally in row major order.[2,3] As an example of array access, consider the following array:

```
double data[10][10];
```

To access a particular element, data[*i*][*j*], the address of the element is located with this access equation:

$$BASE+(i*10+j)*ELSIZE \tag{1}$$

where *BASE* is the base address of the array and *ELSIZE* is the size of each array element. For the machine used, the size of each **double** was 8 bytes.

In generating code for this equation, the compiler distributed the manipulation of *ELSIZE* to produce:

$$BASE+i*80+j*8$$

Accepted September 1987. Discussion closes September 1988.

* Unix is a trademark of AT&T Bell Laboratories.

The assembly code generated for this equation involves one multiplication, one shift and two additions. This is the overhead incurred every time an element of a two-dimensional array is accessed. It should be noted that this access equation is relatively simple in C since all arrays have a lower bound of zero, whereas languages which allow user specified lower bounds, such as Pascal and Fortran, require a more expensive array access.

To access a single element of an array, the evaluation of an equation like (1) cannot be avoided. The key to reducing the array overhead is in recognizing situations where multiple array elements are to be accessed in certain orders. For example, consider the situation in which one needs to access all elements of a particular row of the array. The first element of the row can be located using the access equation, but the expense is not necessary for the other elements. The second element of the row can be found by adding *ELSIZE* to the address of the first element, and so on. Unfortunately, most compilers are not smart enough to detect this situation and they will instead generate code to evaluate the access equation for each element. By using C pointers, the two-dimensional array can be viewed linearly, allowing the programmer to provide his own equation for accessing elements of the array.

Consider, for example, a function for computing the determinant of a triangular matrix, done by multiplying together the diagonal elements. Figure 1(a) shows a version of this function using the two-dimensional array features of C. The loop is entered *SIZE* times, each iteration requiring one multiplication plus the overhead in evaluating the array access equation to locate data[*i*] [*i*]. This results in 2 * *SIZE* multiplications, *SIZE* shifts and 2 * *SIZE* additions to process the loop.

A version of this function using pointers is shown in Figure 1(b). To access each successive diagonal element one needs only to add (*SIZE* + 1) * *ELSIZE* to the address of the previous element. Since both *SIZE* and *ELSIZE* are known at compile time, the compiler generates only one addition to locate each diagonal element. This results in only *SIZE* multiplications and *SIZE* additions to process the loop in this version. Thus a simple application of pointers provides a function less than half as expensive as a similar function using two-dimensional arrays.

The use of pointers in this manner is well known to the assembly programmer. Since assembly languages do not support two-dimensional arrays, the programmer has no option but to implement his own array access code. The advantage of employing this technique in a high level language like C is that the code is more portable. C has proven to be very portable and C compilers are available for a wide range of systems.

It can easily be argued that using pointers to circumvent the overhead of array access is contrary to the philosophy of a high level language. After all, the whole point of providing two-dimensional arrays as a language feature is so the programmer can ignore the details of array access. While this argument may be true for some high level languages, it is not as clear-cut for C. An experienced C programmer is well acquainted with using pointers to increase code efficiency, and it is for precisely this type of use that the pointer features of C were provided.

It should be noted that it is possible to use this method in other high level languages such as Pascal or Fortran. In Pascal, free-union variant records could be employed to trick the compiler into allowing a program-

```
#define SIZE 17

double Determinant(data)

     double data[SIZE][SIZE] ;

{
     double det = 1.0 ;
     int i ;

     for (i=0;  i<SIZE;  i++)
          det * = data[i ] [i ] ;

     return(det) ;
}
```

(a)

```
#define SIZE 17

double Determinant(data)

     double *data ;

{
     double det = 1.0 ;
     int i ;

     for (i=0;  i<SIZE;  i++)
     {
          det * = *data ;
          data + = SIZE + 1 ;
     }

     return(det) ;
}
```

(b)

Figure 1. Computing the Determinant of a Triangular Matrix

mer to access an array with his own access equations. Although this trick is illegal in ANSI Pascal[4] few compilers enforce it and it is a well known technique for getting around Pascal's strong typing. In Fortran, EQUIVALENCE statements could be used in a similar manner. Although employing these tricks may certainly be possible in Pascal and Fortran, C is preferable for this type of pointer manipulation. In C, this method can be implemented with standard features in a manner consistent with the philosophy of the language.

3. ARRAY OVERHEAD IN LU DECOMPOSITION

In a straightforward implementation of LU decomposition using the two-dimensional array features of C, array overhead added substantially to the total expense. In the development of a second version, using the pointer features of C, the pattern in which elements of the coefficient array were accessed during the different steps of the solution was analyzed. Based on this analysis, code to efficiently access the coefficient matrix was developed. The particular access methods used may be found by studying the LU code in the Appendix.

The decomposition step and the solve step were considered separately. For each of these steps, the assembly code generated by the compiler was studied to determine the array access overhead, which was included with the theoretical results to establish new operation counts. For the decomposition step the pivot search, the row interchange, the forming and storing of the multipliers,

	Theoretical Results	*Array Implementation*	*Pointer Implementation*
Decomposition			
Multipications	$\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$	$n^3 + 5n^2 - 6n$	$\frac{1}{3}n^3 + \frac{3}{2}n^2 + \frac{7}{6}n - 3$
Additions	0	$\frac{4}{3}n^3 + 11n^2 - \frac{37}{3}n$	$\frac{2}{3}n^3 + \frac{13}{2}n^2 - \frac{19}{6}n - 4$
Shifts	0	$\frac{2}{3}n^3 + \frac{11}{2}n^2 - \frac{37}{6}n$	$\frac{9}{2}n^2 - \frac{3}{2}n - 3$
Subtractions	$\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$	$\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$	$\frac{1}{3}n^3 - \frac{1}{2}n^2 + \frac{1}{6}n$
Divides	$\frac{1}{2}n^2 - \frac{1}{2}n$	$\frac{1}{2}n^2 - \frac{1}{2}n$	$\frac{1}{2}n^2 - \frac{1}{2}n$
Solve			
Multipications	$n^2 - n$	$2n^2 - n$	$n^2 + n$
Additions	0	$2n^2$	$\frac{1}{2}n^2 + \frac{5}{2}n$
Shifts	0	n^2	$3n$
Subtractions	$n^2 - n$	$n^2 - n$	$\frac{3}{2}n^2 - \frac{1}{2}n$
Divides	n	n	n

Figure 2. Operation Counts

n	*Array Implementation*	*Pointer Implementation*
20	.2000	.1000
50	2.444	1.305
100	17.78	9.095
500	2118.	1116.

Figure 3. Average Execution Times In Seconds for Decomposition

and the elimination were included. For the solve step the elimination and the back substitution were included. The results depict the case in which a row interchange is required at each iteration of the decomposition.

The operation counts are shown in Figure 2. By considering the cost of array access a more accurate view of the operation counts is obtained; the counts for the array implementation are substantially higher than the theoretical counts. The pointer implementation reduces the array overhead resulting in significantly improved counts.

After developing operation counts for both implementations, timing tests were run on the decomposition step. The timing results are shown in Figure 3. Each test run consisted of loading an $n \times n$ array with random values and then called LUDecompose. This timing was performed for $n = 20, 50, 100$, and 500. The time required for loading the coefficient values was measured and subtracted from the total time in order to isolate the decomposition cost. For each value of n, twenty-three test runs were performed for both versions of LUDecompose on a VAX 11/750 with Floating Point Accelerator. The times were measured using the C shell *time* function and only the amount of user time was considered. This timing utility is only accurate to one tenth of a second and so the results for $n = 20$, in particular, are not as precise as desired. However, the times show a general trend in which the pointer version executes significantly faster than the array version. These times reflect the speedup anticipated from the operation counts in Figure 2.

4. CONCLUSION

The effect of two-dimensional array overhead in the implementation of LU decomposition in C was studied to achieve a more accurate picture of the processing required by incorporating the array overhead into the analysis of the algorithm. This analysis was performed for two implementations in C, one using the two-dimensional array features of the language for the coefficient matrix, and the other using pointers. It was found that the cost associated with array access was substantial. The use of pointers allowed a significant reduction of this cost.

REFERENCES

1 Johnston, R. L., *Numerical Methods – A Software Approach*. Wiley, 1982, pp. 28–44

2 Horowitz, E. and Sahni, S., *Fundamentals of Data Structures*. Computer Science Press, 1976, pp. 62–66

3 Kernighan, B. and Ritchie D., *The C Programming Language*. Prentice-Hall 1978, p. 104

4 Cooper, D., *Standard Pascal – User Reference Manual*. Norton, 1983, pp. 107–112

APPENDIX

```
#include <math.h>
#include <stdio.h>


/*
**  This file contains routines for solving an nxn system of linear
**  equations using LU decomposition.  The implementation is based on
**  material in "Numerical Methods - A Software Approach" by Johnston.
**
**  Function LUDecompose performs the decomposition step, LUSolve
**  solves a system given a particular right hand side vector b, and
**  LUDeterminant will return the determinant of a matrix which
**  has been decomposed to LU form.
*/


static int *pivot = NULL ;    /* Pivot vector. */


#define    SMALLEST_PIVOT  1.0e-5  /* Smallest allowed non-singular pivot. */
```

```
int LUDecompose(amat, n, numcols)

  double *amat ;

  int n, numcols ;

/*
**  This function performs the LU decomposition step.  The coefficient
**  matrix 'amat' will be overwritten with the LU matrices in the process.
**
**  Parameters :
**
**        amat - This is the coefficient matrix, where the coefficients
**               are doubles.  This function performs the row-major mapping
**               itself, declaring 'amat' to be a pointer rather than an
**               array.  The calling routine can define the actual parameter
**               to be either.
**
**        n - This parameter indicates the size of the current system (nxn).
**
**        numcols - This parameter indicates the actual defined column dimension
**                  of the coefficient matrix. A bit reminiscent of FORTRAN.
*/


{
  int i, j, k, n_minus_1 = n - 1 ;

  double dtmp1, *dptr1, *dptr2 ;


  /* Allocate storage for the pivot vector. */

  if (pivot != NULL)

  free(pivot) ;




if ((pivot = (int *) malloc(n * sizeof(int))) == NULL)
{
  fprintf(stderr, "Error in LUDecompose - malloc \n") ;
  return(-2) ;
}


/* Initialize pivot vector. */

for (i = 0 ;  i < n_minus_1 ;  i++)
  *(pivot + i) = i ;

*(pivot + n_minus_1) = 1 ;


/* Main loop - perform LU decomposition row by row. */

for (i = 0 ;  i < n_minus_1 ;  i++)
{

  /* search for largest pivot */

  k = i ;                                  /* k will be pivot row. */

  dptr1 = dptr2 = amat + i * (numcols + 1) ;  /* These point to the */
                                              /* pivot element.     */

  for (j = i+1 ;  j < n ;  j++)
  {
    dptr2 += numcols ;
```

```
      if (fabs(*dptr2) > fabs(*dptr1))  /* Current row is new pivot row. */
      {
        dptr1 = dptr2 ;
        k = j ;
      }
    }

    /* k now indicates row containing largest pivot  */
    /* and dptr1 points to the pivot element.        */

    if (fabs(*dptr1) < SMALLEST_PIVOT)   /* Matrix is singular. */
    {
      fprintf(stderr, "Error in LUDecompose - matrix is singular \n") ;
      return(-1) ;
    }

    if (k != i)        /* Interchange rows i and k, updating pivot vector. */
    {
      *(pivot + i) = k ;
      *(pivot + n_minus_1) = - *(pivot + n_minus_1) ;
      dptr1 = amat + i * numcols ;
      dptr2 = amat + k * numcols ;

      for (j = 0 ;  j < n ;  j++, dptr1++, dptr2++)
      {
        dtmp1 = *dptr1 ;
        *dptr1 = *dptr2 ;
        *dptr2 = dtmp1 ;
      }
    }

    /* At this point, the largest pivot has been found and row i */
    /* contains this pivot value.  This next loop performs the   */
    /* elimination step on rows (i+1) through (n-1)              */

    for (j = i+1 ;  j < n ;  j++)
    {
      dtmp1 = *(amat + j * numcols + i) / *(amat + i * numcols + i) ;
      dptr1 = amat + j * numcols + i + 1 ;
      dptr2 = amat + i * numcols + i + 1 ;

      for (k = i+1; k < n ;  k++, dptr1++, dptr2++)
        *dptr1 -= dtmp1 * *dptr2 ;

      *(amat + j * numcols + i) = dtmp1 ;
    }
  }

  return(1) ;
}


void LUSolve(amat, b, n, numcols)

  double *amat, *b ;

  int n, numcols ;

/*
**  This function solves a system of equations given a particular right
**  hand side vector 'b'.  The coefficient matrix 'amat' must be decomposed
**  by the function LUDecompose prior to calling LUSolve.
**
**  Parameters :
**
**         amat - This is the LU decomposition of the coefficient matrix.
**                The original coefficient matrix must first be passed to
**                the function LUDecompose before calling this routine.
**
```

```
**         b - This parameter is the right hand side vector b.  This routine
**             will compute the solution vector and return this result in the
**             parameter 'b'.  NOTE - b WILL BE OVERWRITTEN.
**
**         n - This parameter indicates the size of the current system (nxn).
**
**         numcols - This parameter indicates the actual defined column dimension
**                   of the coefficient matrix. A bit reminiscent of FORTRAN.
*/
{
  double dtmpl, *dptrl ;

  int i, j, n_minus_1 = n-1 ;


  /* Perform the row interchanges recorded in 'pivot' to vector b. */

  for (i = 0 ;  i < n_minus_1 ;  i++)
  {
    j = *(pivot + i) ;

    if (j != i)                /* Interchange element i and j. */
    {
      dtmpl = *(b + i) ;
      *(b + i) = *(b + j) ;
      *(b + j) = dtmpl ;
    }
  }


  /* Solve the system Ld = Pb for d.  Vector d will overwrite b. */

  for (i = 0 ;  i < n ;  i++)
  {
    dtmpl = *(b + i) ;

    for (j = 0, dptrl = amat + i*numcols ;  j < i ;  j++, dptrl++)
      dtmpl -= *dptrl * *(b + j) ;

    *(b + i) = dtmpl ;
  }


  /* Solve the system Ux = d for x.  Vector x will overwrite b (and d). */

  for (i = n_minus_1 ;  i >= 0 ;  i--)
  {
    dtmpl = *(b + i) ;
    dptrl = amat + i*numcols + n - 1 ;

    for (j = n_minus_1 ;  j > i ;  j--)
    {
      dtmpl -= *dptrl * *(b + j) ;
      dptrl-- ;
    }

    *(b + i) = dtmpl / *dptrl ;
  }
}



double LUDeterminant(amat, n, numcols)

  double *amat ;

  int n, numcols ;
```

```
/*
**  This function will compute the determinant of the matrix 'amat'.  'amat'
**  must have already been decomposed to LU form (use LUDecompose).
*/

{
  double det, *fptr ;

  int i ;

  det = 1.0 ;

  for (fptr = amat, i = 0 ;  i < n ;  i++,  fptr += (numcols + 1))
    det *= *fptr ;

  return(det * *(pivot + n - 1)) ;
}
```