Rapport de Travaux Pratiques

LU and Cholesky Factorisation, dense storage and symmetric band

Mention : dense matrix product and BLAS
Filière : Lab 2
Computaional Mechanics

28/09/2023

Tuteurs école :
Nicolas Chevaugeon
Nicolas.Chevaugeon@ec-nantes.fr
`https://cv.hal.science/nicolas-chevaugeon`

Etudiant :
Li Zichen
Zichen.Li@eleves.ec-nantes.fr

Résumé

Many engineering problems can be reduced to the task of solving systems of linear equations, where we typically employ direct and iterative methods during the solving process. In this experiment report, we focus particularly on direct methods, specifically transitioning from Gaussian elimination to LU decomposition and, subsequently, to Cholesky decomposition. In the process of implementing the code, we utilized functions from the LAPACK library and compared the speed with regular decomposition methods.

The experiments below were conducted on a laptop equipped with an Intel 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz 2.30 GHz CPU and 16.0 GB of RAM. The operating system is Ubuntu 22.04 installed on Virtual Box 7 based on Windows 10 Professional, with 8192MB of memory and 8 processors allocated, and 64MB of video memory.

Table des matières

1 Introduction and set-up

Solving systems of linear equations $Ax = b$ is a central problem in scientific computing. We first explore the Gaussian elimination method, which is the preferred algorithm when dealing with A that is square, dense, and unstructured.[1]

Traditional decomposition methods for systems of linear equations involve transforming a given square system of linear equations into an equivalent triangular system with the same solutions. We will start our discussion with the solution of triangular systems here.

1.1 Forward Elimination

Here is an example for 2x2 matrix.

$$\begin{bmatrix} l_{11} & 0 \\ l_{21} & l_{22} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

The general form of the solution for this algorithm is as follows :

$$x_i = \left(b_i - \sum_{j=1}^{i-1} l_{ij} x_j \right) / l_{ii}$$

Since b_1 is only used in the formula for computing x_i, it can be overwritten by b_i.

This algorithm requires n^2 flops. Below is the pseudocode for the algorithm :

```
$b(1) = b(1)/L(1,1)$
for $i = 2 : n$
    $b(i) = (b(i) - L(i, 1 : i-1)b(1 : i-1))/L(i,i)$
end
```

Where L is a nxn lower triangular matrix in $\mathbb{R}^{n \times n}$, and L is a non-singular matrix.

By swapping the loop order, we can obtain a column-based form of the above algorithm. To understand it from an algebraic perspective, we consider the forward elimination method. Once x_1 is solved, this variable can be eliminated from the 2 nd to the nth equations. We can then only consider the reduced system of equations $L(2 : n, 2 : n)x(2 : n) = b(2 : n) - x(1)L(2 : n, 1)$. Next, we calculate x_2, and eliminate x_2 from the 3 rd to the nth equations, and so on.

Let $L \in \mathbb{R}^{n \times n}$ be a lower triangular matrix, and $\boldsymbol{b} \in \mathbb{R}^n$. This algorithm overwrites b with the solution to $Lx = b$. Assume that L is non-singular.

```
for $j = 1 : n - 1$
    $b(j) = b(j)/L(j, j)$
    $b(j + 1 : n) = b(j + 1 : n) - b(j)L(j + 1 : n, j)$
end
$b(n) = b(n)/L(n, n)$
```

This algorithm primarily involves SAXPY operations, the details of which can be found in Appendix A.

1.2 Backward Elimination

The general form of the solution for backward elimination algorithm is as follows :

$$x_i = \left(b_i - \sum_{j=i+1}^{n} u_{ij} x_j \right) / u_{ii}$$

This algorithm requires n^2 flops. Below is the pseudocode for the algorithm :

```
$b(n) = b(n)/U(n, n)$
for $i = n - 1 : -1 : 1$
    $b(i) = (b(i) - U(i, i + 1 : n)b(i + 1 : n))/U(i, i)$
end
```

Where U is a nxn lower triangular matrix in $\mathbb{R}^{n \times n}$, and U is a non-singular matrix.

Similarly, we also swap the loop order to obtain a column-based form.

Let $U \in \mathbb{R}^{n \times n}$ be a lower triangular matrix, and $\boldsymbol{b} \in \mathbb{R}^n$. This algorithm overwrites b with the solution to $Ux = b$. Assume that U is non-singular.

```
for $j = n : -1 : 2$
    $b(j) = b(j)/U(j, j)$
    $b(1 : j - 1) = b(1 : j - 1) - b(j)U(1 : j - 1, j)$
end
$b(1) = b(1)/U(1, 1)$
```

This algorithm primarily involves SAXPY operations, the details of which can be found in Appendix A.

1.3 LU Factorisation

LU decomposition is the process of decomposing a square matrix A into an upper triangular matrix U and a lower triangular matrix L :

$$A = LU$$

In fact, LU decomposition is essentially the product of the Gauss elimination process. The process of Gauss elimination is equivalent to :

$$\begin{aligned} A_1 &= E_1 A \\ A_2 &= E_2 A_1 \\ U = A_3 &= E_3 A_2 \end{aligned}$$

Cascading this series of transformation matrices gives :

$$\begin{aligned} U &= (E_3 E_2 E_1) A, \\ U &= EA \end{aligned}$$

This is equivalent to :

$$\begin{aligned} &(E_3 E_2 E_1)^{-1} U = A \\ &E^{-1} U = A \end{aligned}$$

Here, E^{-1} is precisely the lower triangular matrix L that we are looking for. Reviewing the process of Gauss decomposition, it is also the process of completing LU decomposition. Therefore, the applicable range of LU decomposition is the same as that of the Gauss elimination method.

Gauss elimination and LU decomposition are equivalent, so why do we need LU decomposition ? This is related to time complexity. Firstly, for the Gauss elimination method, suppose A has dimensions $n \times n$. The first elimination requires n^2 multiplication operations, the second elimination requires $(n-1)^2$ multiplication operations, and so on, with the final elimination requiring 1 multiplication operation. The time complexity for the entire Gauss elimination process is cubic :

$$n^2 + (n-1)^2 + \ldots + 1 \approx \frac{1}{3} n^3 = O\left(n^3\right)$$

However, for linear equations where the coefficient matrix is triangular, the solution process is more efficient. Taking the upper triangular matrix as an example, we can solve it by back-substitution, starting from the bottom row and moving upwards. The last row requires 1 multiplication operation, the second to last row requires 2 multiplication operations, ..., and the first row requires n multiplication operations. The entire solution process has quadratic complexity :

$$1 + 2 + \ldots + n = O\left(n^2\right)$$

Therefore, LU decomposition is more efficient for large-scale problems.

2 Part 1

In the first part of the report, we implemented three versions of the LU factorization, without pivoting, namely factorBasic, factorL2, and factorL3. They respectively refer to a direct implementation, a level 2 BLAS implementation, and a level 3 BLAS implementation using a block decomposition of the matrix. Subsequently, we increased the matrix size and compared the running speeds of the three LU decomposition implementations.

2.1 Implementation of factorBasic

The code to achieve the basic LU decomposition, no principal component selection, the given matrix operation. It initializes some variables and then enters the main loop, which computes the elements of the lower triangle and Triangular matrix. The loop continues until all principal diagonal elements are processed or a pivot element near zero is found. At the end of the loop, the code checks to see if the last pivot element is close to zero, and if it is, it prints an error message and returns 0, otherwise it returns 1 for successful LU decomposition.

```
int factorBasic(int n,  double *a, int LDA )
{
    int k = 0;
    double piv = a[0];
    const double minpiv = 1.e-6;
    while ((fabs(piv) > minpiv) && (k < n-1)){
      for (int i = k+1; i < n; ++i){
        a[i+k*LDA] /= piv;
      }
      for (int i = k + 1; i < n; ++i){
        for (int j = k + 1; j < n; ++j){
          a[i+j*LDA] -= a[i+k*LDA]*a[k+j*LDA];
        }
      }
      k += 1;
      piv = a[k+k*LDA];
    }
    if (fabs(piv) <= minpiv){
      std::cout << "Null pivot in factorBasic: "<<piv<<
         __FILE__ << ":" << __LINE__ << std::endl;
    return 0;
    }
    return 1;
};
```

2.2 Implementation of factorL2

The code optimizes the computation of LU decomposition by utilizing Level 2 BLAS functions. Within the main loop, the 'dscal_' function is used for scaling a portion of the matrix, assisting in the computation of the elements of the lower triangular matrix, while the 'dger_' function performs a rank-1 update, aiding in the computation of the elements of the upper triangular matrix. The calls to these BLAS functions aim to enhance computational efficiency, especially when dealing with large matrices. The code also includes a check to ensure that the pivot element is not near zero, to avoid division-by-zero errors during the computation.

```
int factorL2(int n,  double *a, int LDA )
{
    int k = 0;
    double piv = a[0];
    const double minpiv = 1.e-10;//Here we change minpiv
    while ((fabs(piv) > minpiv) && (k < n-1)){
      dscal_(n-(k+1), 1./piv, a+(k+1+k*LDA),1);

      dger_(n-(k+1), n-(k+1), -1., a+(k+1+k*LDA), 1, a+(k+(k
         +1)*LDA), LDA, a+(k+1+(k+1)*LDA), LDA);
      k += 1;
      piv = a[k+k*LDA];
    }
    if (fabs(piv) <= minpiv){
      std::cout << "Null pivot in factorL2: "<<piv<< __FILE__
         << ":" << __LINE__ << std::endl;
    return 0;
    }
    return 1;
};
```

2.3 Implementation of factorL3

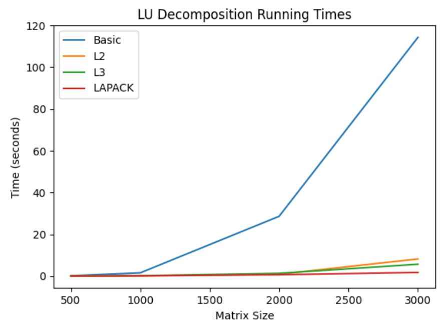
The factorL3 function is another implementation of LU decomposition, utilizing Level 3 BLAS functions to optimize computations. Within the main loop, it first calculates the block size, then calls the factorL2 function to process the current block. If successful, it invokes the BLAS function dtrsm_ twice and dgemm_ once. dtrsm_ is used for solving triangular linear systems, specifically for updating the upper and left parts of the remaining matrix. dgemm_ is employed for performing matrix-matrix multiplication, used for updating the lower right corner of the remaining matrix. These calls to BLAS functions aim to enhance computational efficiency. Finally, if all blocks are successfully processed, the function returns 1 indicating the successful completion of LU decomposition.

```cpp
int factorL3(int r, int n,  double *a, int LDA ){
    int l = 0;
    while (l < n){
      int m = std::min(n, l+r);
      int bsize = m-l;
      int succes = factorL2(bsize, a+(l+l*LDA), LDA);
      if(!succes){
        std::cout <<"Can not factorise one block"<< std::endl;
        return 0;
      }
    dtrsm_('L', 'L', 'N', 'U', bsize, n-m, 1., a+(l+l*LDA),
       LDA, a+(l+m*LDA), LDA);
    dtrsm_('R', 'U', 'N', 'U', n-m, bsize, 1., a+(l+l*LDA),
       LDA, a+(m+l*LDA), LDA);
    dgemm_('N', 'N', n-m, n-m, bsize, -1., a+(m+l*LDA), LDA, a
       +(l+m*LDA), LDA, 1., a+(m+m*LDA), LDA);
    l = m;
    }
    return 1;
}
```

The following are the results we obtained from calling different LU decomposition functions on various matrix sizes.

Matrix size	Basic	L2	L3	LAPACK
500 × 500	0.189608 s	0.0405041 s	0.055522 s	0.0355789 s
1000 × 1000	1.58063 s	0.145896 s	0.202325 s	0.140836 s
2000 × 2000	28.633 s	0.914466 s	1.34714 s	0.672639 s
3000 × 3000	114.293 s	8.22242 s	5.70103 s	1.76227 s

We can observe that calling functions from the BLAS library significantly enhances the efficiency of LU decomposition. By plotting the running time on the x-axis and matrix size on the y-axis, we can clearly see the efficiency of the four LU decomposition functions.



3 Part 2

3.1 Bandwidth

In the following two functions, we need to consider how to obtain the bandwidth of the band matrix from the upper and lower triangular matrices. We need to obtain the maximum bandwidth of the upper or lower triangular matrices as the overall bandwidth of our band matrix.

The 'computeBandwidthUp' function calculates the bandwidth of the upper triangular part of a given matrix, traversing through the matrix's upper triangular part with a nested loop. Upon encountering a non-zero element, it updates the bandwidth to the larger value between the current bandwidth and the difference between the column and row indices. Eventually, the function returns the calculated maximum bandwidth, representing the column difference between the diagonal element and the furthest non-zero element in that row.

```cpp
int computeBandwidthUp(const dmatrix_denseCM &A ){
    int bandwidth = 0;
    for(int i = 0; i < A.getNbLines(); ++i){
        for(int j = i; j < A.getNbColumns(); ++j){
            if(A(i, j) != 0)
                bandwidth = std::max(bandwidth, j - i);
        }
    }
    return bandwidth;
};
```

'computeBandwidthDown' employs the same algorithm.

```cpp
int  computeBandwidthDown(const dmatrix_denseCM &A ){
    int bandwidth = 0;
    for(int i = 0; i < A.getNbLines(); ++i){
        for(int j = 0; j <= i; ++j){
            if(A(i, j) != 0)
                bandwidth = std::max(bandwidth, i - j);
        }
    }
    return bandwidth;
};
```

3.2 Overload parenthesis operator

The operator() function in the dsquarematrix_symband class provides array-like access to elements of a symmetric band matrix. It checks if the requested element is within the bandwidth of the matrix ; if not, it returns zero. If within the bandwidth, it calculates a one-dimensional index based on the row and column indices, and the bandwidth, then returns a reference to the element in the matrix's underlying array.

```cpp
double &dsquarematrix_symband::operator()(int i, int j) {

  if (abs(i - j) > lb) {  // Checking if the element is within
     the band
        static double zero = 0.0;  // static to ensure its
           address remains valid
        return zero;  // Return 0 if outside the band
     }

  int index = (lb  + i - j) + j * (lb + 1);
  std::cout << i << " " << j << " " << index << " "<< m*(lb+1)
     << " " << m << std::endl;
  return a[index];
}
```

Through Cholesky decomposition, we can obtain the following computational results :

TABLE 1 – Factorization Times for Different Files

File Name	Factorization Time	lbu	lbd	Cholesky Band Factorization Time
data_band.mat	3.3781×10^{-5} s	2	2	3.985×10^{-6} s
bcsstk14.mtx	0.0379518 s	161	161	0.00283481 s
bcsstk15.mtx	0.215916 s	437	437	0.0122724 s

A SAXPY

The SAXPY operation is a basic operation in linear algebra, typically included in the BLAS (Basic Linear Algebra Subprograms) library. SAXPY stands for "Single-precision A.X Plus Y." In this operation, two vectors, x and y, and a scalar, a, are involved in the computation :

$$y := a \cdot x + y$$

Here, each element x_i and y_i are elements of vectors x and y, respectively.

Below is a simple C++ implementation of the SAXPY algorithm :

```
#include <iostream>
#include <vector>

void saxpy(float a, std::vector<float>& x, std::vector<float>&
    y) {
    if(x.size() != y.size()) {
        std::cerr << "Error: Size of x and y must be the same!
           " << std::endl;
        return;
    }

    for(size_t i = 0; i < x.size(); ++i) {
        y[i] = a * x[i] + y[i];
    }
}

int main() {
    std::vector<float> x = {1.0f, 2.0f, 3.0f};
    std::vector<float> y = {4.0f, 5.0f, 6.0f};
    float a = 2.0f;

    saxpy(a, x, y);

    std::cout << "Resultant y: ";
    for(const auto& val : y) {
        std::cout << val << ' ';
    }
    std::cout << std::endl;

    return 0;
}
```

Références

[1] Golub, G. H., Van Loan, C. F. (1996). Matrix Computations. JHU Press.

[2] Ghaboussi, Jamshid, and Xiping Steven Wu. Numerical Methods in Computational Mechanics. Boca Raton London New York : CRC Press, an imprint of Taylor et Francis Group, a Spon Press book, 2017.