DOMAIN DECOMPOSITION AND ITERATIVE SOLVERS

LAB-2

LU and Cholesky Factorisation, dense storage and symmetric band

By

Sreekanth Reddy

BAKKA CHENNAIAH GARI

Under the guidance of

Prof. Nicolas Chevaugeon

Faculty of Engineering

Ecole Centrale de Nantes

October, 2023

ABSTRACT

Solving a system of linear equations involving matrices is crucial in engineering. The matrix can be either dense or sparse, and its properties, such as symmetry, positive definiteness, band structure, orthogonality, etc., can significantly simplify the analysis, memory storage, and computational operations, making the process more cost-efficient.

In physics, a common problem is solving $Ax = B$, where A often represents the stiffness matrix, which may possess certain properties like positive definiteness, band structure, or symmetry. In a simple 2D problem with 100 nodes in the x-direction and 100 nodes in the y-direction, the stiffness matrix A would be of size 100×100 with 100 degrees of freedom. However, in practice, directly inverting the matrix, especially for dense matrices, is seldom done due to its computational time and memory intensity.

Instead, methods like Gaussian elimination are employed. Gaussian elimination involves triangularizing the matrix and then using backward substitution to obtain the solution x. In fact, the Gaussian elimination process can be rewritten as a factorization of matrix A into two terms: $A = LU$, where L represents the lower triangular matrix and U represents the upper triangular matrix.

The equation $Ax = b$ can be transformed into $LUx = b$, and when we further consider $Ly = b$, it simplifies to $Ux = y$.

In general, the lower triangular matrix L has diagonal elements set to 1. The product of the diagonal elements of the upper triangular matrix U gives us the determinant of matrix A. If the determinant is close to zero, the matrix is non-invertible. When the determinant of a matrix is close to zero, it indicates that the matrix is nearly singular or ill-conditioned. In such cases, the condition number becomes large because the inverse of the matrix $\mathbf{A}^{-1}$ becomes large in magnitude, and the ratio $\|A\|/\|A^{-1}\|$ becomes significant.

Part-1 demonstrates LU factorization using different algorithms and BLAS levels for a dense matrix. Part-2 demonstrates LU factorization on a symmetric, banded, positive

definite matrix.

The need for pivoting during LU decomposition depends on the diagonal values of the matrix. If they are too small, we may need to interchange rows or columns to achieve diagonal dominance. However, if the matrix has the positive definite property, pivoting may not be necessary. Many engineering problems exhibit these properties, making the analysis more efficient.

In Lab-1, we explored how dense matrices are stored and the factors influencing the cost of analysis. We also examined BLAS libraries, which are efficient tools that make the best use of available resources. In this lab, we demonstrate LU and Cholesky factorization, dense storage, and symmetric band matrices.

The basic idea is that for the factorization to exist, no null pivot should appear. However, in a computer, to check if the pivot is close to zero, we typically use a threshold value like 10^{-6}. If the pivot is smaller than this threshold, an exception is raised, indicating that LU decomposition does not exist.

Computer Specifications of the system used for this lab: Intel(R) Core(TM) i7-10700 CPU @ 2.90GHz 2.90 GHz; 16 GB RAm; 64-bit operating system, x64-based processor. (ECN computer)

CONTENTS

THEORY

The condition number of a matrix A, denoted as $\kappa(A)$, is defined as follows:

$$\kappa(A) = \|A\| \cdot \|A^{-1}\|$$

Where: - $\|A\|$ represents the norm (a measure of the size) of matrix A. The choice of norm (e.g., the 2-norm or the Frobenius norm) can affect the condition number. - $\|A^{-1}\|$ represents the norm of the inverse of matrix A.

Key points about the condition number: 1. A condition number of 1 indicates that the matrix is perfectly conditioned, meaning that small changes in the input data will result in small changes in the output. 2. A condition number significantly larger than 1 indicates that the matrix is ill-conditioned, and small changes in the input data can result in large changes in the output. 3. When solving linear equations using a matrix with a high condition number, the numerical results may be less accurate due to the amplification of errors. 4. The condition number depends on the choice of norm, and different norms may yield different condition numbers for the same matrix. 5. It is important to check the condition number of a matrix when performing numerical computations, as it can help assess the reliability of the results.

PART 1 : DENSE MATRIX FACTORIZATION.

The basic implementation is by using the Gauss elimination, the dense matrix is converted to Upper triangular matrix and lower triangular matrix.
ALGORITHM: input: The matrix A, a n*n matrix and the n component vector b the right hand side
output: The upper triangular matrix U, Lower triangular matrix L and the modified right hand side.
Note that the factorisation is done in place. The memory used to store A and b is used to store U, L

Original matrix A and its LU decomposition:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = LU$$

Now we apply Gaussian elimination to make the first column below the diagonal to zero.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21}/a_{11} & a_{22} & a_{23} \\ a_{31}/a_{11} & a_{32} & a_{33} \end{bmatrix}$$

The update that we made in the above matrix A is a rank-0 update. which is giving the values of lower triangle matrix, which is stored in the same memory location.

The rank-0 update of the matrix:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ L_{21} & a_{22} - (L_{21} \cdot a_{12}) & a_{23} - (L_{21} \cdot a_{13}) \\ L_{31} & a_{32} - (L_{31} \cdot a_{12}) & a_{33} - (L_{31} \cdot a_{13}) \end{bmatrix}$$

$$A = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ L_{21} & U_{22} & U_{23} \\ L_{31} & a'_{32} & a'_{33} \end{bmatrix}$$

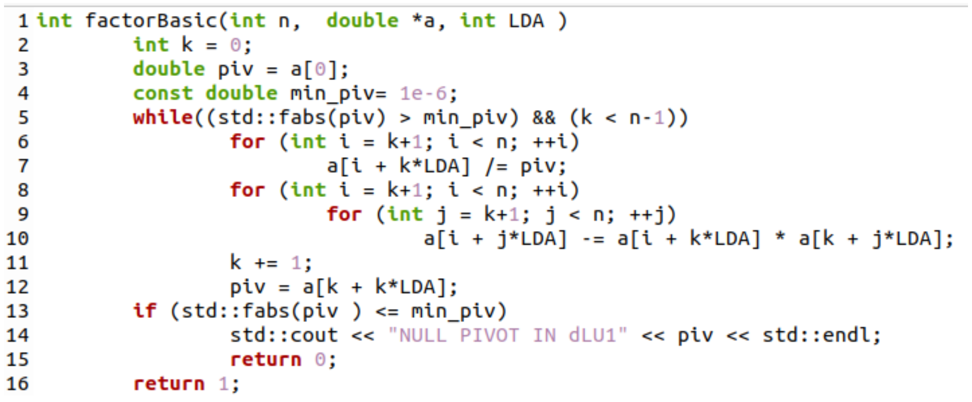
Similarly, we will make $a'_{32} = 0$ by rank-1 update and also update the value of a'_{33} to

U_{33}.

The updated LU decomposition of A can be computed as:

$$A = \begin{bmatrix} U_{11} & U_{12} & U_{13} \\ L_{21} & U_{22} & U_{23} \\ L_{31} & L_{32} & U_{33} \end{bmatrix} = LU$$

The implementation of the above algorithm is demonstrated below in fugue-1.1,



```
int factorBasic(int n,  double *a, int LDA )
        int k = 0;
        double piv = a[0];
        const double min_piv= 1e-6;
        while((std::fabs(piv) > min_piv) && (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 (std::fabs(piv ) <= min_piv)
                std::cout << "NULL PIVOT IN dLU1" << piv << std::endl;
                return 0;
        return 1;
```

Figure 1: Basic version of LU

LEVEL-2 IMPLEMENTATION OF LU DECOMPOSITION

Let's generalize: how to zero the terms from k+1 to n of a vector, Let r $= (0...0..r_{k+1}r_{k+2}...r_n)$ $where, r_i = \frac{x_i}{x_k}$

Now we define a matrix $B_k = I - r \cdot e_k^T$ In the matrix form,

$$B_k \cdot x = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -r_{k+1} & 1 & 0 & 0 & 0 \\ 0 & -r_{k+2} & 0 & 1 & 0 & 0 \\ 0 & | & 0 & 0 & 1 & 0 \\ 0 & -r_n & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} | \\ x_k \\ x_{k+1} \\ | \\ | \\ x_n \end{bmatrix} = \begin{bmatrix} | \\ x_k \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

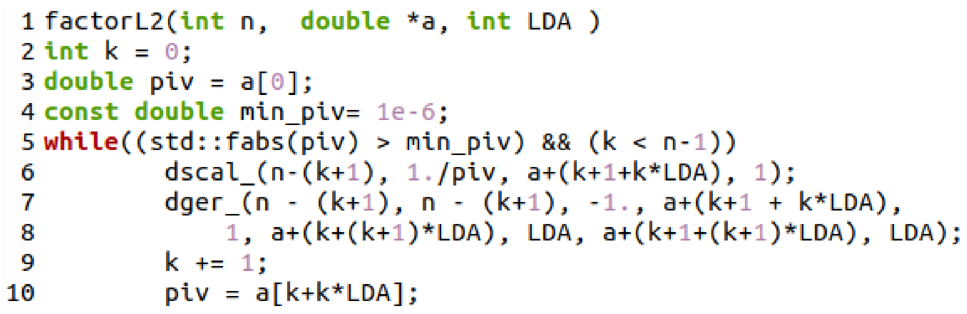
The matrix B_k is called Gauss transformation.

Now, left multiplication of B_k, can be done much faster than the general multiplication $O(n^3)$.

B_k A = (I - $r \cdot e_k^T$) A = A - $r(e_k^T \cdot A)$ = A - r A(k,:)

This operation is a rank-1 update. The number of operations is O((n-k)*n). It can be computed using the BLAS routine dger, A = A + $\alpha x y^T$.

Now that we perform the above operation we have the upper triangle, $M_n - 1..M_2 M_1$ A = U L = $M_1^{-1} M_2^{-1} .... M_n - 1^{-1}$ We know $M_n = (I - r e_k^T)$, $M_n^{-1} = (I + r e_k^T)$



```
factorL2(int n,  double *a, int LDA )
int k = 0;
double piv = a[0];
const double min_piv= 1e-6;
while((std::fabs(piv) > min_piv) && (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];
```

Figure 2: Level-2 implementation

The ‘dscal‘ operation is a level-0 operation. In the iterative loop from row 1 to n, the following steps are performed:

‘dscal‘ divides the entire column-1 by the pivot value. For the 1st iteration ($k = 0$), where n is the size of the matrix, ‘1./piv‘ represents the operation applied to the vector, which is dividing each element by the pivot. ‘a+(k+1+k*LDA)‘ represents the starting memory address of the vector. ‘LDA‘ represents the stride to reach the next value in memory. In our case, it’s one, as we are dividing the column by the pivot, and the matrix is stored in column-major format.

The ‘dger‘ operation performs a rank-1 operation: $A := \alpha \cdot x \cdot y^T + A$.

- The first and second arguments are the size of matrix A, which is $n - 1 \times n - 1$ for the first iteration. - The third argument is the scalar α, which is -1. - The fourth argument is the initial memory address of the sub-matrix. - The fifth argument is the stride to reach the next value in memory. - The sixth argument is the vector y, which is the kth row without the pivot (as explained earlier). - The last argument is the stride to reach the next value in memory.

The operation $\alpha \cdot x \cdot y^T$ computes the outer product by the rank-1 operation, and we reduce the matrix in the first column. Through iterations from the first column to the $n-1$th column, we reduce the matrix to its LU decomposition.

The memory occupied by matrix A is utilized to perform these operations, and the LU factorized values replace the previous values. The strictly lower part of A represents the lower triangular matrix with diagonals as 1, while the upper part of A represents the upper triangle of the matrix decomposition.

In our previous lab experience (Lab-1), it was observed that level-2 operations are more time-efficient compared to level-1 operations. To take advantage of the available level-2 subroutines, a new algorithm is proposed.

LEVEL-3 IMPLEMENTATION OF LU DECOMPOSITION

By dividing the Big matrix into blocks,

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$$

$$A = \begin{bmatrix} L_{11} & 0 \\ L_{21} & I \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & A_{22} - L21 \cdot U_{12} \end{bmatrix} \begin{bmatrix} U_{11} & U_{12} \\ 0 & I \end{bmatrix}$$

$$= \begin{bmatrix} L_{11}U_{11} & L_{11}U_{12} \\ L_{21}U_{11} & A_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = A$$

Step-1: A rank-1 update is used to obtain the LU decomposition of A_{11}

Step-2: $L_{12} \cdot U_{12} = A_{12}$

The `dtrsm()` function is part of the BLAS (Basic Linear Algebra Subprograms) library and is used for solving triangular systems of linear equations. It is typically used for solving equations of the form $A \cdot X = \alpha B$, where A is a triangular matrix, and X and B are rectangular matrices. The sub-routine `dtrsm()` takes several arguments:

1. The first argument specifies whether the triangular matrix A is on the left or right

side of the equation. Use "L" for the left side or "R" for the right side.

2. The second argument specifies whether the triangular matrix A is upper or lower triangular. Use "U" for upper or "L" for lower.

3. The third argument specifies whether to use the transpose of the triangular matrix A. Use "N" for no transpose or "T" for transpose.

4. The fourth argument specifies whether the diagonal of the triangular matrix A is unit or non-unit. Use "U" for unit or "N" for non-unit.

5. The fifth and sixth arguments represent the size of the matrix A.

6. The seventh argument, α, represents the scalar used to scale the matrix B before solving the equation.

7. The eighth and ninth arguments are the starting addresses of matrix A and the stride required to fetch the next value.

8. The tenth and last arguments are the starting addresses of matrix X and the stride required to fetch the next value.

During the execution of the subroutine, matrix multiplication is performed, and the results are stored in matrix B, effectively replacing the values of X in memory.

Similarly, $L_{21} \cdot U_{11} = A_{21}$, L_{21} is computed using dtrsm. Once the L_{21} and U_{12} are computed, we need to update the A_{22}. $A_{22} = A_{22} - L_{21} \cdot U_{12}$, which can be computed using the sub-routine dgemm C := alpha*op(A)*op(B) + beta*C,

The blas sub-routine:

void dgemm(const char* transa, const char* transb, const int* m, const int* n, const int* k, const double* alpha, const double* a, const int* lda, const double* b, const int* ldb, const double* beta, double* c, const int* ldc);

transa: Specifies whether to transpose matrix A. Use "N" for no transpose, "T" for transpose.

transb: Specifies whether to transpose matrix B. Use "N" for no transpose, "T" for transpose.

m : The number of rows of matrix C.

n : The number of columns of matrix C.

k : The number of columns of matrix A (if not transposed) or the number of rows of matrix A

alpha: A scalar used to scale the product of matrices A and B.

a : The matrix A.lda:

b : The matrix B.

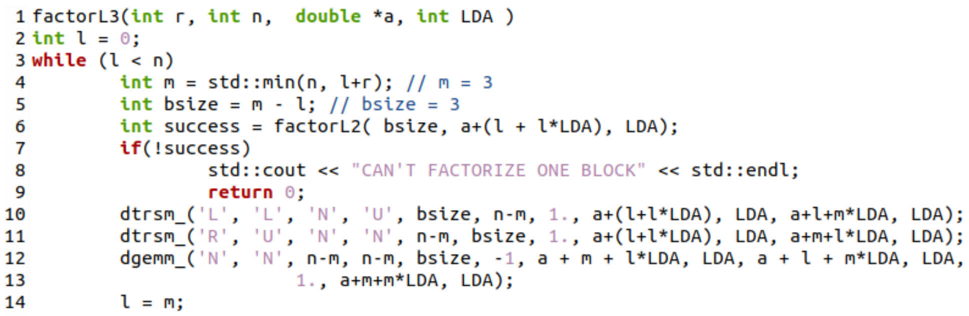
ldb: The leading dimension of matrix B. For column-major storage, this should be at least max(1,

beta: A scalar used to scale matrix C before adding the product of matrices A and B.

c : The matrix C, where the result of the multiplication is stored.

ldc: The leading dimension of matrix C. For column-major storage, this should be at least max(1,

Now the same procedure is used iteratively to compute the LU decomposition of the A_{22} until the last element is reached.



```
factorL3(int r, int n,  double *a, int LDA )
int l = 0;
while (l < n)
        int m = std::min(n, l+r); // m = 3
        int bsize = m - l; // bsize = 3
        int success = factorL2( bsize, a+(l + l*LDA), LDA);
        if(!success)
                std::cout << "CAN'T FACTORIZE 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', 'N', 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;
```

Figure 3: Level-3 implementation

The algorithm involves $2n^3/3$ flops, the same amount as the regular factorization. For the sake of the analysis, let's say that $n = rN$. The algorithm ends after N iterations. At each iteration, it performs:

- 1 factorization of a $r \times r$ matrix, dominated by Level 2 speed: $2r^3/3$ flops.

- 2 triangular solve with a matrix on the right-hand side at Level 3 speed.
- 1 rank r update at Level 3 speed.

So the only significant part of the algorithm which is stuck at Level 2 is the factorization of the $r \times r$ diagonal block.

Let's compute the fraction of the operations that are performed at Level 3 speed:

$$1 - \frac{N(2r^3/3)}{2n^3/3} = 1 - \frac{1}{N^2}$$

So for large N, almost all the arithmetic is performed using Level 3 BLAS. If r is sufficiently large, we can expect the same efficiency here as in a BLAS matrix-matrix operation.

RESULTS AND DISCUSSION: PART-1

`dmatrix_denseCM M(m, m);`:

This line declares a matrix M of type `dmatrix_denseCM` with dimensions m rows and m columns.

`std::generate(M.begin(), M.end(), std::bind(dis, gen));`:

This generates a matrix using the random library with values in the range [0.1, 1]. `M.begin()` is a pointer to the initial address of the matrix, and `M.end()` is a pointer to the end of the matrix. `std::bind(dis, gen)` generates the random values, where `dis` specifies the probability distribution to use in generating the values, and `gen` is the random value generator.

To ensure the matrix is symmetric positive definite, a matrix is created by $M \cdot M^T$.

`dLU_denseCM LU(A, factorpolicy(factorpolicy::L2));`:

Calling this function will perform the decomposition using the implemented functions.

`dmatrix_denseCM B(m, 1);`: This line declares a matrix B of type `dmatrix_denseCM` with m rows and 1 column.

`std::generate(B.begin(), B.end(), std::bind(dis, gen));`: This generates random values for the right-hand side vector B. `B.begin()` is a pointer to the initial address

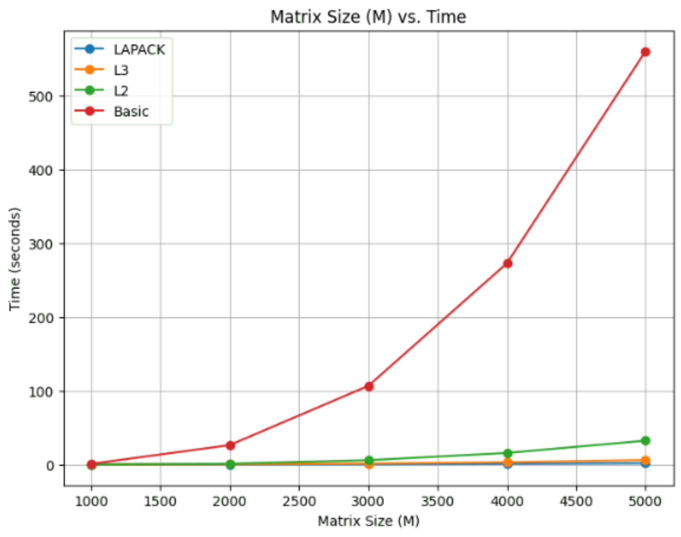


Figure 4: Comparison of basic, level-2, 3 and Lapack

of the vector, and `B.end()` is a pointer to the end of the vector.

`dmatrix_denseCM X = LU.solve(B);`: This line solves the linear system of equations $Ax = B$ to compute X, where A is the decomposed matrix obtained from the LU decomposition.

The error in the solution is computed as $\|Ax - B\|$, where $\|\cdot\|$ represents the norm.

The time taken to Solve a linear system of equations of different sizes and different versions are compared in the below figure 1.4.

`The time taken to solve Ax = B, with A of size 5000*5000`

1. Lapack version: 1.98646 secs
2. Level-3 block version: 6.17911 secs
3. Level-2 version: 32.4227 secs
4. Basic version: 560.9 secs

To visualize more clearly only L2, L3, and Lapack versions are shown in the below figure 1.5

The variation in time between the level-2, level-3 implementation can be visualized from figure.5, which is very significant as we increase the size of the matrix. where as, the lapack version makes use of the available computer architecture effectively. They are optimized for various hardware architectures. They take advantage of memory hierarchy

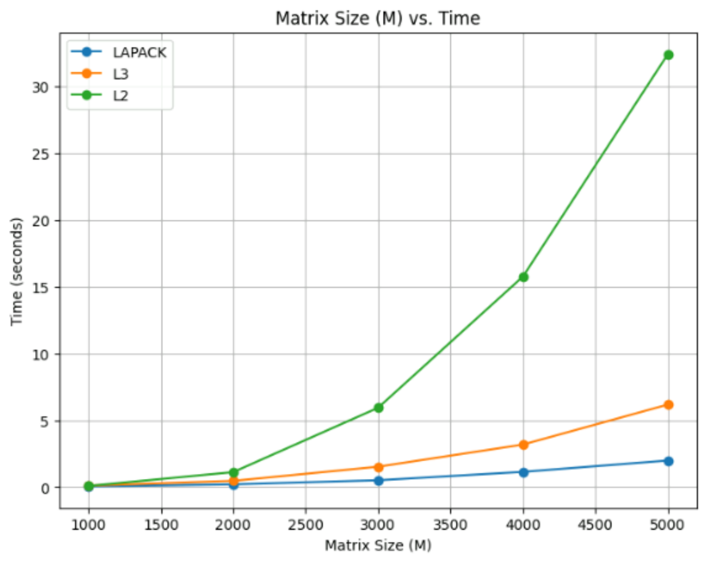


Figure 5: Comparison of level-2, 3 and Lapack

to maximize the performance. They also take advantage of the multiple CPU cores or threads for performance gains and is very efficient compared to the other implementations.

PART 2 : SYMMETRIC POSITIVE DEFINITE BAND MATRIX.

DECOMPOSITION OF BANDED SYMMETRIC MATRIX USING LU

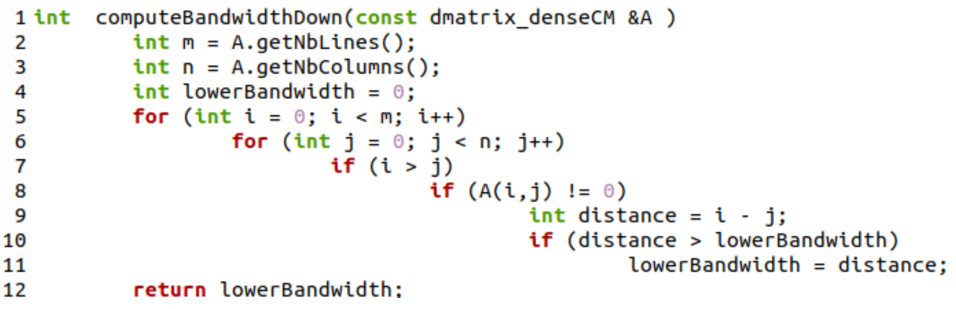
The algorithm for the computation of the upper and lower bandwidth is presented in Figure 2.1 and Figure 2.2.

for Upper bandwidth, iteratively check for j > i and a(i,j) != 0, if the above conditions are satisfied, find the distance of the element to the diagonal, overwrite the value of bandwidth if it exceeds the previously computed bandwidth.

Similarly for the lower bandwidth, for (i > j) and a(i,j)!= 0, if the above conditions are satisfied, find the distance of the element to the diagonal, overwrite the value of bandwidth if it exceeds the previously computed bandwidth.

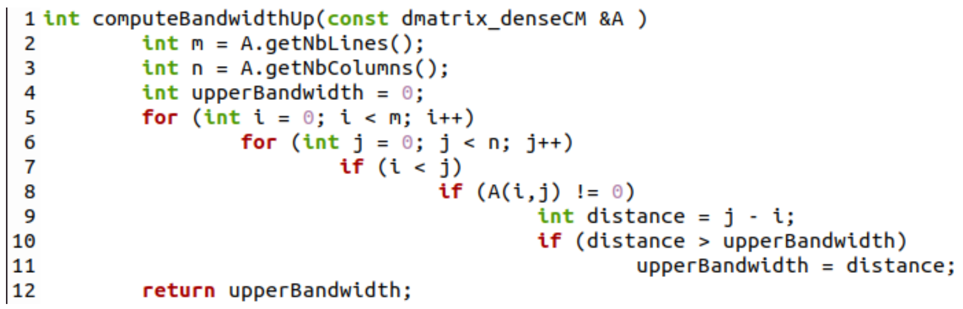
```
dmatrix_denseCM A;
std::string filename("bcsstk15.mtx");
std::ifstream in(filename.c_str());
in >> A;
dLU_denseCM LU(A, factorpolicy(factorpolicy::Lapack));


dmatrix_denseCM B(A.getNbLines(), 1);
std::generate(B.begin(), B.end(), std::bind(dis, gen));
dmatrix_denseCM X = LU.solve(B);
```

The object A from the class dmatrix_denseCM is created, bcsstk15.mtx contains a band matrix of size , The file is opened using the fstream library. The operator "»" is



```
int  computeBandwidthDown(const dmatrix_denseCM &A )
        int m = A.getNbLines();
        int n = A.getNbColumns();
        int lowerBandwidth = 0;
        for (int i = 0; i < m; i++)
                for (int j = 0; j < n; j++)
                        if (i > j)
                                if (A(i,j) != 0)
                                        int distance = i - j;
                                        if (distance > lowerBandwidth)
                                                lowerBandwidth = distance;
        return lowerBandwidth;
```

Figure 6: Computation of lower bandwidth



```
int computeBandwidthUp(const dmatrix_denseCM &A )
        int m = A.getNbLines();
        int n = A.getNbColumns();
        int upperBandwidth = 0;
        for (int i = 0; i < m; i++)
                for (int j = 0; j < n; j++)
                        if (i < j)
                                if (A(i,j) != 0)
                                        int distance = j - i;
                                        if (distance > upperBandwidth)
                                                upperBandwidth = distance;
        return upperBandwidth;
```

Figure 7: Computation of upper bandwidth

the function in the class which enables to read the elements from a file in to A.

The matrix A from the file bcsstk15.mtx is decomposed into LU using the Lapack library.

The matrix B of size (rows of A, 1) are randomly generated using the random library.

The Ax=B, the linear system of equations are solved and the computed x is used to see how accurate the Cholesky decomposition produces the result.

Column-Major Storage:

Matrix A:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$$

Memory Address	Matrix Element
0	a_{11}
1	a_{21}
2	a_{31}
3	a_{12}
4	a_{22}
5	a_{32}
6	a_{13}
7	a_{23}
8	a_{33}

When stored in column-major order, the matrix A is laid out in memory as follows: Each column is stored sequentially in memory, starting from the first column and moving to the second, and so on.

CHOLESKY FACTORIZATION

Cholesky factorization, is a numerical method for decomposing a symmetric, positive-definite matrix into the product of a lower triangular matrix and its transpose. This factorization has several important applications, particularly in numerical linear algebra and solving systems of linear equations.

MATRIX REQUIREMENTS

- Cholesky factorization can only be applied to symmetric matrices.
- The matrix must be positive-definite, which means all its eigenvalues are positive.

FACTORIZATION

Given a symmetric positive-definite matrix A, Cholesky factorization finds a lower triangular matrix L such that $A = LL^T$, where L^T is the transpose of L. The lower triangular matrix L is a unique factorization of A.

PROCEDURE

1. Start with an $n \times n$ matrix A, where n is the dimension of the matrix.
2. For each element $L(i, j)$ of L and corresponding element $A(i, j)$ of A:
    - If $i = j$ (the diagonal element), compute $L(i, j)$ as the square root of $A(i, i)$ minus the sum of squares of all previously computed $L(k, j)$ terms for k from 1 to $i - 1$.
    - If $i \neq j$ (off-diagonal element), compute $L(i, j)$ as ($A(i, j)$ minus the sum of products of $L(k, i)$ and $L(k, j)$ terms for k from 1 to $i - 1$) divided by $L(i, i)$.
3. Repeat this process for all elements of L and A.
4. The resulting matrix L will be the lower triangular Cholesky factor of A.

PROPERTIES AND ADVANTAGES

- Cholesky factorization is computationally efficient and numerically stable when applied to positive-definite matrices. It is often faster and more accurate than other matrix factorization methods.

- Once you have the Cholesky factor L, solving linear systems of equations of the form $Ax = b$ becomes simpler and more efficient, as you can solve for x using forward and backward substitution.

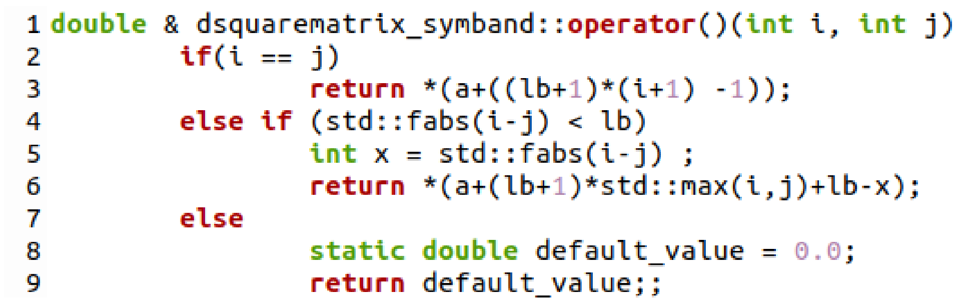
```
lbu = Compute_Bandwidth_Up(A)
dsquarematrix_symband Ab(A, lbu)
dCholesky_band cholAb(Ab);
dmatrix_denseCM X2 =  cholAb.solve(B);
```

The Compute_Bandwidth_Up(A) computes the bandwidth of the matrix A using the fuction implemented as show in the figure 2.1 .

dsquarematrix_symband Ab(A, lbu) : The constructor of the class dsquarematrix_symband allocates the storage based on the bandwidth.

COLUMN-MAJOR STORAGE OF BANDED SYMMETRIC MATRIX

In column-major storage, a banded symmetric matrix is stored by placing its columns sequentially in memory, including zero elements. Let's consider a banded symmetric matrix A of size $m \times m$ with a lower bandwidth (k_l) and an upper bandwidth (k_u), which is equal for a symmetric matrix:



```
double & dsquarematrix_symband::operator()(int i, int j)
        if(i == j)
                return *(a+((lb+1)*(i+1) -1));
        else if (std::fabs(i-j) < lb)
                int x = std::fabs(i-j) ;
                return *(a+(lb+1)*std::max(i,j)+lb-x);
        else
                static double default_value = 0.0;
                return default_value;;
```

Figure 8: Implementation of () operator for fetching band-storage data

$$A = \begin{bmatrix} a_{11} & a_{21} & a_{31} & 0 & 0 & \dots & 0 \\ a_{21} & a_{22} & a_{32} & a_{42} & 0 & \dots & 0 \\ a_{31} & a_{32} & a_{33} & a_{43} & a_{53} & \dots & 0 \\ 0 & a_{42} & a_{43} & a_{44} & a_{54} & \dots & 0 \\ 0 & 0 & a_{53} & a_{54} & a_{55} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & a_{mm} \end{bmatrix}$$

The upper or lower part is only stored due to symmetry, the banded symmetric matrix A is laid out in memory as follows:

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} & 0 & 0 & 0 & 0 \\ & a_{22} & a_{23} & a_{24} & 0 & 0 & 0 \\ & & a_{33} & a_{34} & a_{35} & 0 & 0 \\ & & & a_{44} & a_{45} & a_{46} & 0 \\ & & & & a_{55} & a_{56} & a_{57} \\ & & & & & a_{66} & a_{67} \\ & & & & & & a_{77} \end{bmatrix}$$

The above matrix can be stored in memory using row or column major. Let's say the matrix is stored in column major,

Memory Layout (Column-Major Order):

| 0 | 0 | a11 | 0 | a12 | a22 | a13 | a23 | a33 | a24 | a34 | a44 | a35 | a45 | a55 | a46 | a56 | a66 | a57 | a67 | a77 |

To fetch the data from the storage, a () operator is implemented for a banded-symmetric matrix. The pseudo-code of the implementation is demonstrated in Figure 2.3 .

```
dsquarematrix_symband::dsquarematrix_symband( const dmatrix_denseCM & A, int lb):
    set_size(A.getNbLines(), lb);
    std::fill(a,a+m*(lb+1),0.);
    for (int idiag = 0; idiag <=lb; ++idiag)
        for (int j = idiag; j< m; ++j)
            int i = j-idiag;
            (*this)(i,j) = A(i,j);
```

The above constructor copy the symmetric band matrix A which is stored in a dense format, to a banded storage matrix. The argument lb is the bandwidth of the matrix A.

```
dCholesky_band::dCholesky_band(const dsquarematrix_symband &in ):m(in.getNbLines()), 
  std::copy(in.begin(), in.end(), a);
  int info;
  dpbtrf_('U', m, lb, a, lb+1, info );
```

std::copy(in.begin(), in.end(), a); copies the data from 'in' to 'a'.

dpbtrf (Double Precision Band TRiangular Factorization)

Computes the Cholesky factorization of a symmetric positive definite band matrix.

Input:

- *U* Specifies whether the upper or lower triangular part of the matrix is stored.
- *m* The order of the matrix.
- *lb* The number of super-diagonals or sub-diagonals of the matrix.
- *a* The packed storage representation of the matrix (input). It should contain the matrix data.
- *lb+1* The leading dimension of the array AB.

- *INFO:* An integer output variable. If INFO is 0, the factorization was successful. If INFO is a positive integer, it indicates the index of the first zero diagonal element in the factorization, and the matrix is not positive definite.

```
dmatrix_denseCM dCholesky_band::solve(const dmatrix_denseCM &B) const
  int nrhs = B.getNbColumns();
  int nequ = B.getNbLines();
  assert(nequ == m);
  dmatrix_denseCM X(B);
  int info;
  dpbtrs_('U', m, lb, nrhs, a, lb+1, X.data(), nequ, info);
```

dpbtrs (Double Precision Band TRiangular Solve) Solves a system of linear equations with a symmetric positive definite band matrix.

Input:

- *U* Specifies whether the upper or lower triangular part of the matrix is stored.
- *m* The order of the matrix.
- *lb* The number of super-diagonals or sub-diagonals of the matrix.
- *NRHS* The number of right-hand sides in the system of equations.
- *a* The packed storage representation of the factorized matrix (output from dpbtrf).
- *lb+1* The leading dimension of the array AB. Typically, LDAB $\geq$ 2*KD + 1.
- *X.data()* The right-hand side matrix.
- *nequ* The leading dimension of the array B.
- *INFO:* An integer output variable. If INFO is 0, the system was successfully solved. If INFO is a positive integer, it indicates a problem, and the solution may not be reliable.

RESULTS AND DISCUSSION:

The solution x, solving the Ax = b using cholesky factorisation stored in a banded format is compared with the solution from the LU decomposition of matrix stored in dense format. The Lapack library is used in both cases to decompose the system of equations into LU.

Time Comparison:

Table 1: Execution Times for Different Bandwidths

Bandwidth	**Exec. Time in secs**					
	56	140	161	437	521	1243
Dense Storage	0.0226	0.4657	0.0419	0.2766	4.503	5.5329
Banded Storage	0.0018	0.0439	0.0062	0.0236	0.0835	0.2538

Storage Comparison: Dense Square Matrix:

- In a square matrix of size $n \times n$, you have n^2 variables.
- Each element of the matrix is stored, regardless of whether it is zero or non-zero.
- Total storage required: n^2

Symmetric Banded Matrix:

- In a symmetric banded matrix of size $n \times n$ with bandwidth k:
- Only the non-zero elements along the main diagonal and k adjacent diagonals are stored.
- For each row, you need to store $(2k + 1)$ elements (k elements to the left, the diagonal element, and k elements to the right).
- However, since the matrix is symmetric, you only need to store half of the matrix (either the upper or lower triangular part).
- Total storage required: $(k + 1) \times n$

Error in the Factorization:

Bandwidth of a Matrix	**Error in the Factorization**
56	0.00178607
140	0.0438962
161	6.49E-07
437	2.56775E-05
521	0.000116369
1243	0.0014372

The table demonstrates the error in factorization w.r.t bandwidth size, computed using Cholesky factorization in banded storage format compared to LU factorization in dense storage format, is influenced by various matrix properties beyond just the bandwidth. Several matrix properties can significantly impact the accuracy of the factorization process. These properties include:

- **Condition Number:** The condition number of a matrix measures its sensitivity to small perturbations. A high condition number indicates that the matrix is ill-conditioned and can lead to numerical instability in factorization algorithms.

- **Diagonal Dominance:** A matrix is said to be diagonally dominant if the magnitude of each diagonal element is greater than or equal to the sum of the magnitudes of the off-diagonal elements in the same row. Diagonally dominant matrices tend to have better numerical stability in factorization methods.

- **Skyline Format:** Skyline storage is a format that efficiently stores symmetric matrices with a specific structure where entries outside the lower or upper triangle are zero. Banded storage may not be as efficient for skyline matrices as it is for general banded matrices due to the specific structure of skyline matrices. In the matrices that we considered, if the matrix is skyline, we are computing the bandwidth on the whole matrix depending on the distance between the non-zero element and the diagonal element. The bandwidth of the skyline matrix can be high, and sparsity of the matrix can also be high, where solving this skyline matrix using banded storage is not efficient and thereby needs a different algorithm to compute the decomposition of a skyline matrix.

CONCLUSION

In summary, matrix properties are critical factors that impact the accuracy and efficiency of matrix factorization techniques. Sparse matrices, defined as the ratio of the number of zero-valued elements to the total number of elements, can be less efficient to factorize using dense storage methods for sparse matrices. Banded storage formats are often more suitable for sparse matrices because they exploit sparsity to reduce storage and computation requirements. Skyline matrices exhibit a unique structure where entries outside the lower or upper triangle are zero. The skyline matrix solved using the banded storage algorithm is not efficient when a banded algorithm is implemented due to the high sparsity of the matrix. The Lapack library offers a valuable resource for accessing algorithms and subroutines tailored to specific matrix properties. These algorithms are optimized to make the most efficient use of computational resources. In conclusion, an understanding of matrix properties is essential for making decisions regarding the choice of factorization methods and storage formats. These factors are instrumental in achieving accurate and efficient matrix factorization.

BIBLIOGRAPHY

- The LAPACK library documentation: https://netlib.org/lapack/explore-html/index.html
- The Matrix Market: https://math.nist.gov/MatrixMarket/
- Oracle's documentation on plug matrices: https://docs.oracle.com/cd/E19422-01/819-3692/plug$_matrices.html$
- LU Decomposition of band matrix : https://www.math.fsu.edu/ gallivan/courses/NumOptim/LUFactorization.pdf
- Lecture notes from the DDIS course