Rapport de Travaux Pratiques

Factorization of sparse matrices

Mention : Sparse direct solver
Filière : Lab 3
Computational Mechanics

12/10/2023

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

Standard discretizations of Partial Differential Equations typically lead to large and sparse matrices. A sparse matrix is defined, somewhat vaguely, as a matrix which has very few nonzero elements. But, in fact, a matrix can be termed sparse whenever special techniques can be utilized to take advantage of the large number of zero elements and their locations. These sparse matrix techniques begin with the idea that the zero elements need not be stored. One of the key issues is to define data structures for these matrices that are well suited for efficient implementation of standard solution methods, whether direct or iterative.[4]

In this experiment report, I focus on describing the direct methods for solving sparse matrices.

In the references section of this report, we have provided some excellent textbooks and documents. Given that the focus of this experiment is on the direct solution of sparse matrices, we highly recommend reading the excellent textbook 'Direct Methods for Sparse Linear Systems, SIAM, Philadelphia, PA, 2006' by Tim Davis.[6] Also, we suggest checking out 'CSparse' on GitHub, written by Tim Davis and part of the SuiteSparse project.

Table des matières

1 Introduction and set-up

Solving a sparse system of equations, $Ax = b$, using a direct method starts with a factorization of A, followed by a solve phase that uses the factorization to solve the system. In all but a few methods, the factorization splits into two phases : a symbolic phase that typically depends only on the nonzero pattern of A, and a numerical phase that produces the factorization itself.[3] The solve phase uses the triangular solvers as discussed in Lab2.

The symbolic phase is asymptotically faster than the numeric factorization phase, and it allows the numerical phase to be more efficient in terms of time and memory. It also allows the numeric factorization to be repeated for a sequence of matrices with identical nonzero pattern, a situation that often arises when solving non-linear and/or differential equations.

The first step in the symbolic analysis is to find a good fill-reducing permutation. Sparse direct methods for solving $Ax = b$ do not need to factorize A, but can instead factorize a permuted matrix : PAQ if A is unsymmetric, or PAP^T if it is symmetric. Finding the optimal P and Q that minimizes memory usage or flop count to compute the factorization is an NP-hard problem (Yannakakis 1981), and thus heuristics are used. Fill-reducing orderings are a substantial topic in their own right. Understanding how they work and what they are trying to optimize requires an understanding of the symbolic and numeric factorizations.

Once the ordering is found, the symbolic analysis finds the elimination tree, and the nonzero pattern of the factorization or its key properties such as the number of nonzeros in each row and column of the factors.

Although the symbolic phase is a precursor for all kinds of factorizations, it is the Cholesky factorization of a sparse symmetric positive definite matrix $A = LL^T$ that is considered first. Much of this analysis applies to the other factorizations as well (QR can be understood via the Cholesky factorization of A^TA, for example).[3]

1.1 Cholesky Factorisation

If a matrix is symmetric and positive definite, It can be factorised as $A = LL^T$, Where L is Lower triangular matrix with positive diagonal entries. It is called the *Cholesky* factorisation.

For symmetric positive matrix :

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{12} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{1n} & a_{2n} & \cdots & a_{nn} \end{bmatrix}$$

We use the Gaussian elimination method, first performing row transformations, then column transformations, until matrix A is transformed into a diagonal matrix.

Let $L_i, i = 1, 2, \cdots, k$ represent the row transformations on A. Its transpose can then represent the column transformations. Thus, the transformation that eliminates elements from the first row and first column can be expressed as :

$$\mathbf{L}_1\mathbf{A}\mathbf{L}_1^T = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22}^{(1)} & \cdots & a_{2n}^{(1)} \\ \cdots & \cdots & \cdots & \cdots \\ 0 & a_{2n}^{(1)} & \cdots & a_{nn}^{(1)} \end{bmatrix}$$

By repeating such operations, matrix A can be transformed into a diagonal matrix :

$$\mathbf{L}_k\mathbf{L}_{k-1}\cdots\mathbf{L}_1\mathbf{A}\mathbf{L}_1^T\mathbf{L}_2^T\cdots\mathbf{L}_k^T = \mathbf{D}$$

While this method can transform the matrix into a diagonal matrix, it may not necessarily be the most rational sequence. There are two reasons for this : **Firstly,** if the value of a diagonal element is relatively small, using it for elimination may cause significant rounding errors. **Secondly,** eliminating elements in sequence may introduce more non-zero elements, causing the matrix to lose its sparsity.

Therefore, it is necessary to adjust the order of the matrix when needed. Let $\mathbf{P}_{ij}$ represent the swapping of the i th and j th rows of the identity matrix. Then $\mathbf{P}_{ij}^T$ represents the swapping of the i th and j th columns. For a symmetric matrix $A, \mathbf{P}_{ij}\mathbf{A}\mathbf{P}_{ij}^T$ remains symmetric, but these two transformations change the order of the diagonal elements.

Suppose the symmetric matrix A can be transformed into a diagonal matrix by a series of elementary matrices $\mathbf{L}_1, \mathbf{L}_2, \cdots, \mathbf{L}_k$ through row and column transformations. Let $\mathbf{L} = \mathbf{L}_k\mathbf{L}_{k-1}\cdots\mathbf{L}_1$, then we have :

$$\mathbf{L}\mathbf{A}\mathbf{L}^T = \mathbf{D}$$
$$\mathbf{A} = \mathbf{L}^{-1}\mathbf{D}\left(\mathbf{L}^T\right)^{-1}$$
$$\mathbf{A}^{-1} = \left(\mathbf{L}^{-1}\mathbf{D}\left(\mathbf{L}^T\right)^{-1}\right)^{-1} = \mathbf{L}^T\mathbf{D}^{-1}\mathbf{L}$$

So the solution of $Ax = b$:

$$x = \mathbf{A}^{-1}\mathbf{b} = \mathbf{L}^T\mathbf{D}^{-1}\mathbf{L}\mathbf{b}$$

This solution method is called the Cholesky decomposition method. In this method, choosing a reasonable order for elimination is the key to reducing computational load and improving solution accuracy.

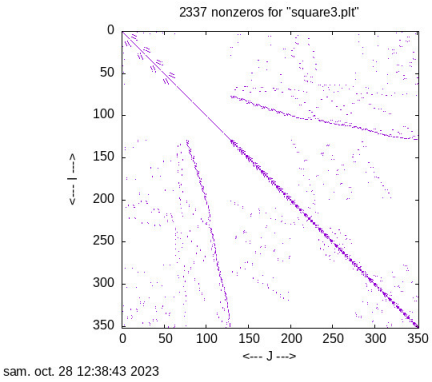
We define a class in dLLT_denseCM.h/cc that computes and stores a Cholesky factorization of a dense matrix in Column major format. Additionally, we also define a class in dLLT_CCS.h/cc that computes and stores a Cholesky factorization of a sparse matrix in CCS format.

1.2 sparse LU factorization

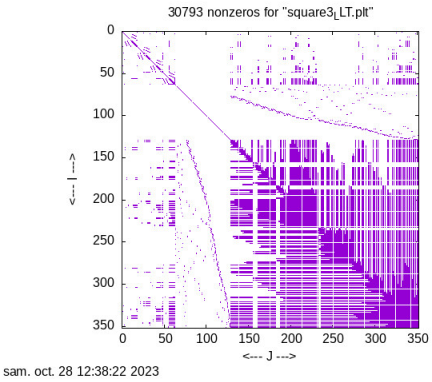
Most direct methods for sparse linear systems perform an LU factorization of the original matrix and try to reduce cost by minimizing fill-ins, i.e., nonzero elements introduced during the elimination process in positions which were initially zeros. The data structures employed are rather complicated. The early codes relied heavily on linked lists which are convenient for inserting new nonzero elements. Linkedlist data structures were dropped in favor of other more dynamic schemes that leave some initial elbow room in each row for the insertions, and then adjust the structure as more fill-ins are introduced.[4]

Gaussian elimination on the original matrix results in disastrous fill-ins. Specifically, the L and U parts of the LU factorization are now dense matrices after the first step of Gaussian elimination. With direct sparse matrix techniques, it is important to find permutations of the matrix that will have the effect of reducing fill-ins during the Gaussian elimination process.

Here we show two pictures using gnuplot to produce nice pictures of the non-zero term of the matrix. We describe in detail the generation method of these two pictures in Chapter 2, Section 3. The purple part of the picture represents non-zero elements. Here we can find that the sparse matrix of the first picture produces a large number of fill-ins after Cholesky decomposition (such as the second picture). This will greatly increase solution time and storage space.



(a) Sparse matrix by square3.msh



(b) After Cholesky decomposition

FIGURE 2 – Sparse matrix and fill-ins

1.3 Graph

Graph theory is an ideal tool for representing the structure of sparse matrices and for this reason it plays a major role in sparse matrix techniques. For example, graph theory is the key ingredient used in unraveling parallelism in sparse Gaussian elimination or in preconditioning techniques.[4]

A graph is defined by two sets, a set of vertices

$$V = \{v_1, v_2, \ldots, v_n\}$$

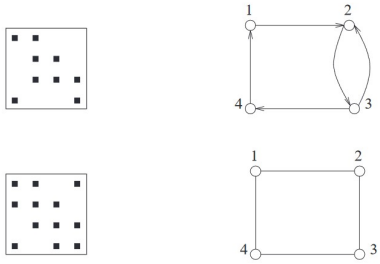
and a set of edges E which consists of pairs (v_i, v_j), where v_i, v_j are elements of V,

$$E \subseteq V \times V$$

This graph $G = (V, E)$ is often represented by a set of points in the plane linked by a directed line between the points that are connected by an edge. A graph is a way of representing a binary relation between objects of a set V.[4]

The *adjacency graph* of a sparse matrix is a graph $G = (V, E)$, whose n vertices in V represent the n unknowns. Its edges represent the binary relations established by the equations in the following manner : There is an edge from node i to node j when $a_{ij} \neq 0$. This edge will therefore represent the binary relation equation i involving unknown j. Note that the adjacency graph is an undirected graph when the matrix pattern is symmetric, i.e., when $a_{ij} \neq 0$ iff $a_{ji} \neq 0$ for all $1 \leq i, j \leq n$).

When a matrix has a symmetric nonzero pattern, i.e., when a_{ij} and a_{ji} are always nonzero at the same time, then the graph is undirected. Thus, for undirected graphs, every edge points in both directions. As a result, undirected graphs can be represented with nonoriented edges.



In FEM, for many linear elasticity mechanics problems, the stiffness matrix obtained using the standard finite element method is typically symmetric positive definite. For certain time-dependent problems, such as those related to time-dependent diffusion or transport, the matrix that results from using implicit time-stepping methods might also be symmetric positive definite. In FVM, for some steady-state, elliptical problems, the discretized equations derived using the central difference method might produce a symmetric positive definite coefficient matrix. In this experiment, our primary focus is on symmetric positive definite matrices and their undirected graphs.

For undirected and directed graphs, DFS and BFS work essentially the same, except that the order in which edges and vertices are accessed may vary. Note that when applying DFS and BFS to a digraph, the direction of the arrows (that is, the direction of the edges) is important because it determines which vertices can

be reached from one vertex. This will be discussed in Part 1, where we will explain BFS for an undirected graph.

1.4 Fill-reducing heuristics

1.4.1 π-permutation

First, we talk about the permutation of matrix.

Let A be a matrix and $\pi = \{i_1, i_2, \ldots, i_n\}$ a permutation of the set $\{1, 2, \ldots, n\}$. Then the matrices

$$A_{\pi,*} = \left\{a_{\pi(i),j}\right\}_{i=1,\ldots,n;j=1,\ldots,m},$$
$$A_{*,\pi} = \left\{a_{i,\pi(j)}\right\}_{i=1,\ldots,n;j=1,\ldots,m}$$

are called row π-permutation and column π-permutation of A, respectively.

Consider, for example, the linear system $Ax = b$ where

$$A = \begin{pmatrix} a_{11} & 0 & a_{13} & 0 \\ 0 & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & 0 \\ 0 & a_{42} & 0 & a_{44} \end{pmatrix}$$

and $\pi = \{1, 3, 2, 4\}$, Given a permutation matrix P corresponding to the permutation π, the symmetrically permuted matrix can be represented as :

$$A_{\pi,\pi} = P_\pi A P_\pi^T$$

In this equation :

- P^T is the transpose of the permutation matrix, which permutes the rows of A.
- P permutes the columns of A.

For the previous example, if the rows are permuted with the same permutation as the columns, the linear system obtained is

$$\begin{pmatrix} a_{11} & a_{13} & 0 & 0 \\ a_{31} & a_{33} & a_{32} & 0 \\ 0 & a_{23} & a_{22} & a_{24} \\ 0 & 0 & a_{42} & a_{44} \end{pmatrix} \begin{pmatrix} x_1 \\ x_3 \\ x_2 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_3 \\ b_2 \\ b_4 \end{pmatrix}$$

Observe that the diagonal elements are now diagonal elements from the original matrix, placed in a different order on the main diagonal.[4]

- **Global Approach :** The matrix is permuted into a matrix with a given pattern, so that the fill-in is restricted from occurring within that pattern(like *π-permutation*).
  - Cuthill-McKee

  - Reverse Cuthill-McKee
  - Nested dissection
- **Local approach :** using heuristic that select pivot during a symbolic factorization to reduce fill-in, like Approximated minimum degree algorithm.

Our implementation for permuttion is primarily in main_testCholesky_CCS.cc. We use the Cuthill-McKee ordering and reverse Cuthill-McKee ordering to achieve permutation.

1.4.2 Breadth First Search (BFS)

The nodes of a certain level can be visited in the natural order in which they are listed. The neighbors of each of these nodes are then inspected. Each time, a neighbor of a visited vertex that is not numbered is encountered, it is added to the list and labeled as the next element of the next level set. This simple strategy is called Breadth First Search (BFS) traversal in graph theory.

```
1.  Initialize S = {v}, seen = 1, π(seen) = v; Mark v;
2.  While seen < n Do
3.      S_new = ∅;
4.      For each node v in S do
5.          For each unmarked w in adj(v) do
6.              Add w to S_new;
7.              Mark w;
8.              π(++seen) = w;
9.          EndDo
10.         S := S_new
11.     EndDo
12. EndWhile
```

1. **Initialization :**
   - S is a set that initially contains the starting node v.
   - *seen* is a counter that tracks the number of nodes that have been observed (or visited). It's set to 1 initially since we've already observed the starting node v.
   - $\pi(seen) = v$ is an array that records the order in which nodes are visited.
2. **Main Loop :** The loop continues as long as the number of observed nodes *seen* is less than the total number of nodes n.
3. **Exploration at Each Level :**
   - S_{new} is a new set to collect all neighboring nodes at this level.
   - For each node v in the current set S, inspect each of its unmarked neighboring nodes w.
   - Add w to S_{new} and mark it.
   - Update the π array to record the order of visiting w.
   - After all nodes in S have been inspected, assign S_{new} to S and proceed to the next iteration.
4. **Termination :** The loop ends when all nodes have been visited.

1.4.3 Cuthill-McKee Algorithm

The Cuthill-McKee Algorithm is a variant of the breadth-first search algorithm. In the Cuthill-McKee ordering the nodes adjacent a a visited node are always traversed from lowest to highest degree.

```
0.   Find an intial node v for the traversal
1.   Initialize S = {v}, seen = 1, π(seen) = v; Mark v;
2.   While seen < n Do
3.       S_new = ∅;
4.       For each node v Do:
5.           π(++seen) = v;
6.           For each unmarked w in adj(v), going from lowest to highest degree D
7.               Add w to S_new;
8.               Mark w;
9.           EndDo
10.          S := S_new
11.      EndDo
12.  EndWhile
```

Here's a brief explanation of the given algorithm :

1. **Initialization :**
   - The algorithm starts by selecting an initial node v. This node could be chosen based on certain criteria, e.g., the node with the lowest degree.
   - S is initialized to contain this initial node.
   - **'seen'** is a counter to keep track of how many nodes have been processed.
   - π is an array to store the new ordering of the nodes.
   - The initial node v is marked to indicate that it has been processed.
2. **Main Loop :**
   - As long as not all nodes are seen ('**seen** $<$ **n**'), the algorithm continues.
   - For the current set S, the algorithm looks at all adjacent nodes (neighbors) that haven't been marked (or processed) yet.
   - These unmarked neighbors are sorted based on their degree (from lowest to highest).
   - These sorted nodes are added to the new set S_new new and marked.
   - After processing all nodes in S, S is replaced by S_new and the process continues.
3. **Outcome :**
   - The end result is the π array, which gives a new ordering of the nodes such that the matrix's bandwidth is (potentially) reduced.

The key idea of the Cuthill-McKee algorithm is that by starting from a node with a low degree and expanding outwards, one can reorder the matrix in a way that brings non-zero elements closer to the diagonal. This has benefits for storage and computational efficiency, especially for algorithms that operate on the matrix's non-zero elements.

The previous two algorithms were described with the explicit use of level sets.[4] A more common, and somewhat simpler, implementation relies on queues. The queue implementation is as follows.

```
0.  Find an intial node v for the traversal
1.  Initialize Q = {v}, Mark v;
2.  While |Q| < n Do
3.      head++;
4.      For each unmarked w in adj(h), going from lowest to highest degree Do:
5.          Append w to Q;
6.          Mark w;
7.      EndDo
8.  EndWhile
```

1. **Data Structure :**
   - The original algorithm uses a set S (and a new set S_{new} for each iteration) to keep track of the nodes being processed.
   - The improved version uses a queue Q with a head pointer to track the front of the queue.
2. **Loop Logic :**
   - In the original algorithm, the loop relies on the existence of unprocessed nodes in S. Once S is emptied, the nodes from S_{new} are moved to S to continue processing.
   - In the improved version, the loop continues until the queue's size equals the number of nodes. The head pointer is incremented to move to the next node in the queue, thus eliminating the need to shuffle nodes between two sets.
3. **Ordering :**
   - Both versions process adjacent nodes in increasing order based on their degree. This is a key feature of the CMK algorithm that helps reduce the matrix bandwidth.
4. **Simplicity :**
   - The queue-based implementation is simpler and more intuitive. Using a queue naturally aligns with the breadth-first search (BFS) nature of the CMK algorithm.
   - The queue-based version doesn't require swapping between two sets, which reduces overhead and potential for errors.
5. **Efficiency :**
   - The queue-based version is likely more efficient because it uses a linear data structure that is more cache-friendly and doesn't involve the overhead of constantly moving nodes between two sets.
   - Using a head pointer to traverse the queue is faster than repeatedly checking and updating two different sets.

In summary, the improved version of the CM algorithm using a queue is a more streamlined and efficient approach that captures the BFS essence of the algorithm in a more natural way. The use of a queue simplifies the logic and likely improves the runtime efficiency of the algorithm.

Our subsequent implementation in Chapter 2, Section 4 also relies on queue implementation.

1.4.4 Reverse Cuthill-McKee Algorithm

The Reverse Cuthill-McKee (RCMK) algorithm is a direct extension of the Cuthill-McKee (CMK) algorithm. The main goal of both algorithms is to reduce the bandwidth of a sparse matrix, which is beneficial for storage and computation, especially in solving sparse linear systems.

The primary difference is in the final step. After obtaining the ordering from the Cuthill-McKee algorithm, the Reverse Cuthill-McKee algorithm simply reverses that order. Reversing the order has been found to often result in even smaller bandwidth than the original CMK ordering.

Pseudocode for RCMK :

The RCMK pseudocode can be built directly upon the CMK pseudocode :

```
function CuthillMcKee(graph G)
    Initialize an empty queue Q
    Select the initial vertex v with the lowest degree in G
    Push v into Q
    Mark v

    while Q is not empty do
        Pop vertex u from the front of Q
        for each unmarked neighbor w of u, in increasing order
            of degree do
            Mark w
            Push w into Q
        end for
    end while

    return the order of vertices as they were dequeued from Q
end function

function ReverseCuthillMcKee(graph G)
    order = CuthillMcKee(G)  // Obtain the order using the CM
       algorithm
    reverse(order)           // Reverse the order
    return order
end function
```

After obtaining the order from the CMK function, the RCMK algorithm simply reverses it to get the final order. This reversed order is the one that's typically used in applications because it generally provides better bandwidth reduction.

2 Part 1 : Sparse Cholesky and the importance of numbering

2.1 Analysis of main_testCholesky_CCS.cc

2.1.1 generateFromCommandLineArguments

- Set a default input file.
- Parses the command line arguments to obtain the names of the input files and determine whether to print the matrix.
- Determines how input files are processed based on their file extension ('.msh' or '.mtx').
  - '.msh' represents a mesh file. It will read this mesh and generate a finite element mass matrix.
  - '.mtx' is a matrix file. It will read this file directly to get the matrix data.

2.1.2 main

1. Use 'generateFromCommandLineArguments' function to generate matrix A.
2. Initialize variables :
   - 'B' is a randomly generated right-hand matrix
   - 'Xref' will store the solution of the original system
3. Original system solution : Use Cholesky decomposition to solve 'A * X = B', where 'A' is a sparse matrix and 'X' is the solution we want to find.
   - Measure and print the time required for Cholesky decomposition.
   - Print the number of nonzero items in 'L'.
   - Check the correctness of the solution.
4. Using Cuthill-McKee sorting : The goal of this part is to see if reordering improves the speed of solving the problem
   - Use the cuthillmckee function to calculate the new ordering
   - Reorder the system and fix it
   - Measure and print the time required for the newly sorted Cholesky decomposition
   - Check the correctness of the solution
5. Use reverse Cuthill-McKee sorting

2.2 Optimization of main_testCholesky_CCS.cc for plotting

Here we provide a function to write the matrix renumbered into the '.mtx' format. Afterwards, we can use the **examplePlot.cc** and the **gnuplot** tool to obtain a visualization of the matrix's non-zero element distribution.

```cpp
/* main_testCholesky_CCS.cc */
void writeMatrixToMtx(const dmatrix_CCS &matrix, const std::string &filename) {
    std::ofstream outfile(filename);

    if (!outfile.is_open()) {
        std::cerr << "Error opening file for writing: " << filename << std::endl;
        return;
    }

    // Write the Matrix Market banner
    outfile << "%%MatrixMarket matrix coordinate real general\n";

    // Write matrix dimensions and non-zero count
    outfile << matrix.m << " " << matrix.n << " " << matrix.nnz << "\n";

    // Write the matrix data
    for (int col = 0; col < matrix.n; ++col) {
        for (int idx = matrix.columnptr[col]; idx < matrix.columnptr[col + 1]; ++idx) {
            outfile << matrix.lineindex[idx] + 1 << " " << col + 1 << " " << matrix.a[idx] << "\n";  // +1 because MTX is 1-indexed
        }
    }
    outfile.close();
}

int main(int argc, char *argv[]){
    //...
   std::vector< int > neworder = cuthillmckee( buildSymGraph(A) );
  { //...
// Usage:
    writeMatrixToMtx(pAp, "data/permuted_matrix_square_cuthillmckee.mtx");
    //...
}
    //...
}
```

2.3 Breadth First Search (BFS)

The code is a standard BFS implementation on a graph. The use of a queue **(to_treat)** ensures that nodes are processed in a breadth-first manner. The **visited** vector ensures that each node is processed only once, avoiding infinite loops in the case of cycles in the graph. And the **order** vector captures the sequence of node visits.

```
std::vector<int > breathFirstSearch(const graph & g, int
    start_node){
  assert (start_node < g.nbNodes());
  std::vector<int > order;
  std::vector<int>  visited( g.nbNodes(), 0);
  std::queue<int > to_treat;
  to_treat.push(start_node);
  while(!to_treat.empty()){
    int i = to_treat.front();
    to_treat.pop();
    if (!visited[i]){
      visited[i] = 1;
      order.push_back(i);
      for(auto j : g.getNeighbour(i))  if(!visited[j])
         to_treat.push(j);
    }
  }
  return order;
}
```

Let's analysis the implementation of BFS.

1. **Initialization :**
   - **order** is a vector to store the order of the nodes as they are visited.
   - **visited** is a vector that marks whether a node has been visited or not.
   - **to_treat** is a queue to handle nodes that need to be processed, similar to set S in the pseudocode.
2. **Starting Node :** Push the **start_node** onto the **to_treat** queue, indicating that it's the first node to be processed.
3. **Main Loop :**
   - As long as there are nodes left in the **to_treat** queue, continue to process them.
   - For each node, check if it has been visited. If not :
     - Mark it as visited.
     - Add it to the **order** vector.
     - Push all its unvisited neighbors onto the **to_treat** queue.
4. **Return :** After processing all nodes, the function returns the **order** vector, which contains nodes in the order they were visited using BFS.

2.4 Implementation of Cuthill-McKee Algorithm

We test our Cuthill-McKee Algorithm in a small matrix first, our program runs well. Then we test our program in a large matrix, we got a problem with non-connected graphs.

A simple way to deal with non-connected graphs is to add a loop to the implementation of the Cuthill-McKee algorithm, so that when we have processed a connected component, we can choose another node that has not been visited yet and continue the algorithm. This way, we can generate a separate ordering for each connected component in the graph and eventually combine them into a single ordering.

We iterate through all nodes in the outer loop. If a node has already been visited, we skip it. Otherwise, we select the node with the smallest degree among the current connected components as the starting node and perform the Cuthill-McKee algorithm. This way, the algorithm is executed independently on each connected component and then the results are merged into a single sequence.

```
/* graph.cc */
std::vector<int> cuthillmckee(const graph &g) {
    std::vector<int> order;   // This will store the result
    std::vector<int> visited(g.nbNodes(), 0);  // Initialization with 0 meaning not visited
    std::queue<int> to_treat;

    for (int start_node = 0; start_node < g.nbNodes(); ++start_node) {
        if (visited[start_node]) continue;  // Skip if this node has already been visited

        // Selecting the node with the least degree in the current connected component
        int min_degree = g.nbNodes();
        for (int i = start_node; i < g.nbNodes(); ++i) {
            if (g.getNeighbour(i).size() < min_degree && !visited[i]) {
                start_node = i;
                min_degree = g.getNeighbour(i).size();
            }
        }

        to_treat.push(start_node);

        while (!to_treat.empty()) {
            int i = to_treat.front();
            to_treat.pop();
```

```
25              if (!visited[i]) {
                    visited[i] = 1;
27                  order.push_back(i);

29                  // Get the neighbors of the current node and
                       sort them based on their degree
                    std::vector<int> neighbours(g.beginNeighbour(i
                       ), g.endNeighbour(i));
31                  std::sort(neighbours.begin(), neighbours.end()
                       , [&g](int a, int b) {
                        return g.getNeighbour(a).size() < g.
                           getNeighbour(b).size();
33                  });

35                  // Push the sorted neighbors to the queue
                    for (int j : neighbours) {
37                      if (!visited[j]) {
                            to_treat.push(j);
39                      }
                    }
41              }
            }
43      }

45      if (order.size() != g.nbNodes()) {
            std::cout << "Error: Not all nodes were visited!" <<
               std::endl;
47          throw;
        }
49

        return order;
51  }
```

Method	Cholesky Time(s)	NNZ in L	LB of L	Solution Error
Original System	14.112	190791	161	3.17742×10^{-11}
Cuthill-McKee Ordering	18.2671	207811	199	1.43623×10^{-16}
Reversed Cuthill-McKee	13.896	179583	199	9.87644×10^{-17}

TABLE 1 – Performance Comparison of Different Methods for bcsstk14.mtx

Here we can find that Cuthill-McKee and even Reversed Cuthill-McKee have no obvious efficiency improvement. At this time, we need to use the gnuplot tool provided previously to observe the structure of the matrix.

2.4.1 Plotting to analyse for bcsstk14.mtx

In Chapter 1, Section 2, the images we presented were drawn using files such as mat2gnuplot.h, example_plot.cc, mesh.h/cc, and others. Specifically, mat2gnuplot.h contains a function to transform a matrix into a format usable with gnuplot to produce a visually pleasing representation of the matrix's non-zero terms. Examples of how to use this are provided in example_plot.cc. Additionally, mesh.h/cc defines a 'mesh' class to store data about meshes, capable of reading a mesh from a file in the gmsh format ([link]). It also defines the 'fem' class, which can assemble the mass matrix of a finite element problem, allowing for the construction of large sparse symmetric positive definite matrices.

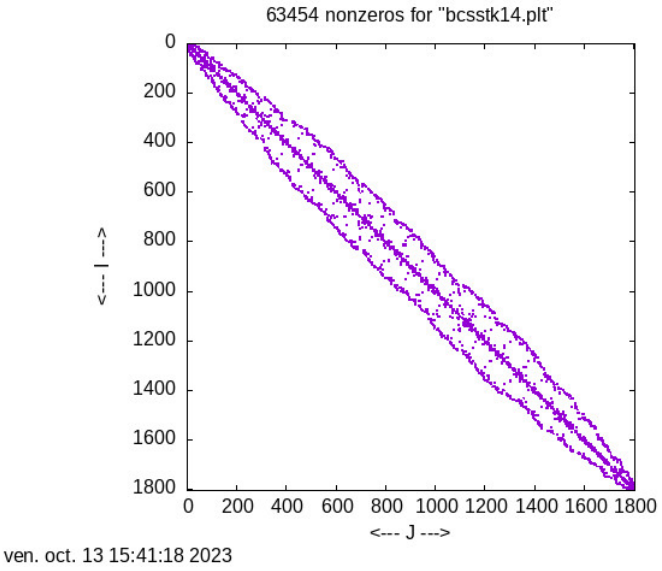
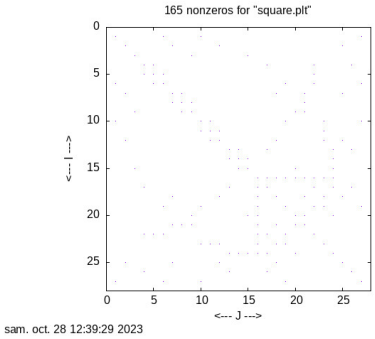


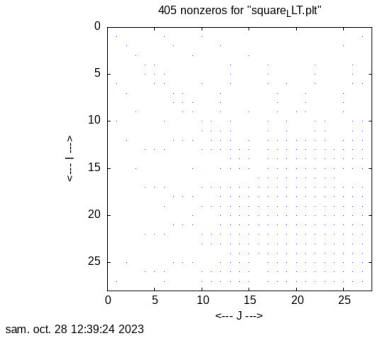
FIGURE 3 – Plot matrix bcsstk14

So we find that for small-bandwidth, well-banded band matrices, our Cuthill-McKee and even Reversed Cuthill-McKee improvements are limited. Now let us turn our attention to large mass matrices.

2.4.2 Cuthill-McKee and Reversed Cuthill-McKee for square.msh



(a) Sparse matrix by square.msh



(b) After Cholesky decomposition (without renumbering)

FIGURE 4 – Sparse matrix and fill-ins

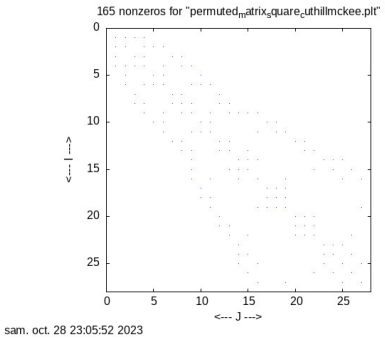
We can find that compared to the bcsstk14 matrix, this mass matrix generated by the square mesh is not "well-banded", so we think Cuthill-McKee and Reversed Cuthill-McKee will provide better results.

We conclude our outputs as following :

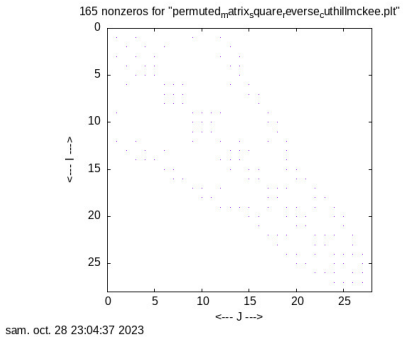
Method	**Cholesky Time(s)**	**NNZ in L**	**LB of L**	**Solution Error**
Original System	0.000539509	217	26	1.07418×10^{-15}
Cuthill-McKee Ordering	0.000406525	203	11	3.31243×10^{-13}
Reversed Cuthill-McKee	0.000308337	165	11	2.60011×10^{-13}

TABLE 2 – Performance Comparison of Different Methods for square.msh

Even we just apply a small sparse matrix, we can observe significant progress by cuthillmckee ordering and reverse cuthillmckee ordering. Our symmetric band matrix breadth is reduced to 42.3%. Then we try to visualize these orderings.



(a) Renumbering by cuthillmckee



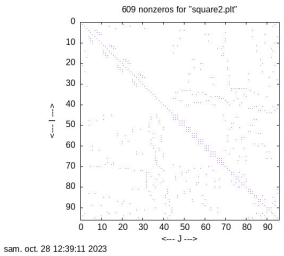
(b) Renumbering by reverse cuthillmckee

FIGURE 5 – Fill-reducing permutation by cuthillmckee and reverse cuthillmckee

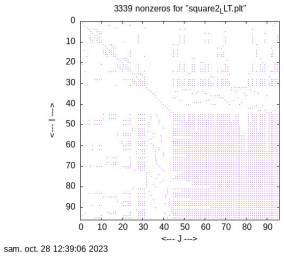
Our optimized code helps us observe the results of renumbering. Generally, the bandwidth of the band matrix obtained by Cuthill-McKee and Reversed Cuthill-McKee in a symmetric sparse matrix (symmetric because it is a mass matrix) is consistent. But Reversed Cuthill-McKee gets fewer non-zero elements and performs cholesky decomposition faster.

For larger positive definite symmetric mass matrices, we will find that renumbering has better results.

2.4.3 Cuthill-McKee and Reversed Cuthill-McKee for square2.msh



(a) Sparse matrix by square2.msh



(b) After Cholesky decomposition (without renumbering)

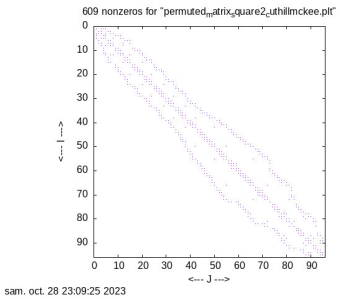
FIGURE 6 – Sparse matrix and fill-ins

Due to space limitations, the clarity of our image display is limited. In the assignment we submitted, there are original images in the **permuted plots** folder.

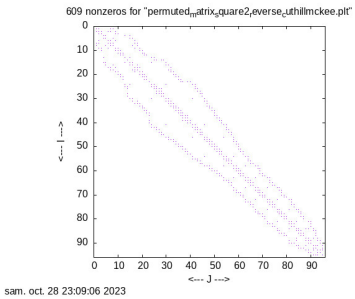
Method	**Cholesky Time(s)**	**NNZ in L**	**LB of L**	**Solution Error**
Original System	0.013174	1718	95	2.03426×10^{-15}
Cuthill-McKee Ordering	0.00334773	1107	18	1.60234×10^{-12}
Reversed Cuthill-McKee	0.00252485	940	18	1.70289×10^{-12}

TABLE 3 – Performance Comparison of Different Methods for square2.msh

Here we can find that the time to perform cholesky decomposition after using renumbering is only about 20% of that of directly decomposing the sparse matrix in CCS format, and the non-zero elements are only half of the original. This greatly reduces the storage required for calculation and improves the efficiency. The bandwidth of the strip matrix is only about 20%.



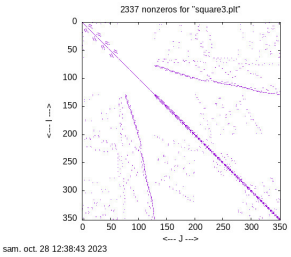
(a) Renumbering by cuthillmckee



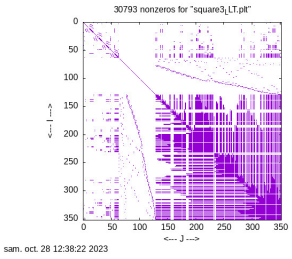
(b) Renumbering by reverse cuthillmckee

FIGURE 7 – Fill-reducing permutation by cuthillmckee and reverse cuthillmckee

2.4.4 Cuthill-McKee and Reversed Cuthill-McKee for square3.msh



(a) Sparse matrix by square3.msh



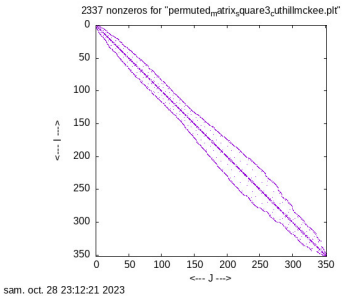
(b) After Cholesky decomposition (without renumbering)

FIGURE 8 – Sparse matrix and fill-ins

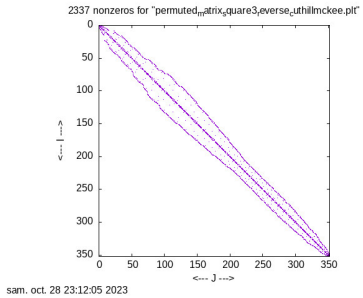
Method	Cholesky Time(s)	NNZ in L	LB of L	Solution Error
Original System	0.593443	15573	345	6.41756×10^{-15}
Cuthill-McKee Ordering	0.0486671	7421	33	1.51188×10^{-11}
Reversed Cuthill-McKee	0.0382461	6477	33	1.5517×10^{-11}

TABLE 4 – Performance Comparison of Different Methods for square3.msh

Here we can find that the time to perform cholesky decomposition after using renumbering is only about 10% of that of directly decomposing the sparse matrix in CCS format, and the non-zero elements are only less than half of the original. This greatly reduces the storage required for calculation and improves the efficiency. The bandwidth of the strip matrix is only about 10%.



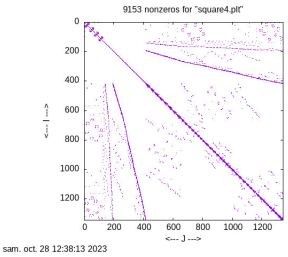
(a) Renumbering by cuthillmckee



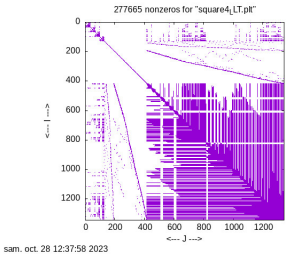
(b) Renumbering by reverse cuthillmckee

FIGURE 9 – Fill-reducing permutation by cuthillmckee and reverse cuthillmckee

2.4.5 Cuthill-McKee and Reversed Cuthill-McKee for square4.msh



(a) Sparse matrix by square4.msh



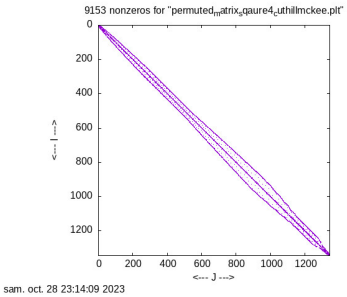
(b) After Cholesky decomposition (without renumbering)

FIGURE 10 – Sparse matrix and fill-ins

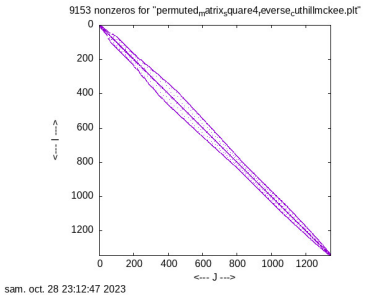
Method	Cholesky Time(s)	NNZ in L	LB of L	Solution Error
Original System	33.9548	139505	1317	1.35563×10^{-14}
Cuthill-McKee Ordering	0.865896	53155	65	1.30158×10^{-10}
Reversed Cuthill-McKee	0.675299	47035	65	1.23089×10^{-10}

TABLE 5 – Performance Comparison of Different Methods for square4.msh

Here we can find that the time to perform cholesky decomposition after using renumbering is only about 2% of that of directly decomposing the sparse matrix in CCS format, and the non-zero elements are only one third of the original. This greatly reduces the storage required for calculation and improves the efficiency. The bandwidth of the strip matrix is only about 5%.



(a) Renumbering by cuthillmckee



(b) Renumbering by reverse cuthillmckee

FIGURE 11 – Fill-reducing permutation by cuthillmckee and reverse cuthillmckee

So we can find that in actual engineering, especially for large mass matrices, renumbering can greatly improve our calculation efficiency and is very helpful for solving finite element problems.

3 Part 2 : Using an efficient library : superLU

3.1 Implementation of SuperLU

We use the same implementation as 'main_testCholesky_CCS.cc'. The key part is to convert .mtx format matrix to a SuperLU format matrix.

```
    // generate a matrix from command line argument. Since we
       are using Cholesky decomposition here, A should be
       symetric postiv definite
dmatrix_CCS Accs = generateFromCommandLineArguments(argc,
   argv);

// Variable to store the start and end of a sequence of code
    for timing purpose
std::chrono::time_point<std::chrono::system_clock> start,
   end;

int m = Accs.getNbLines();

int n = Accs.getNbColumns();
int nnz = Accs.nnz;
// B is the Right hand side matrix to solve for. here a
   random matrix, with m line and 3 column (we want to solve
    the system fo 3 right and side )
int nrhs = 3;
dmatrix_denseCM rhs = generateRandom(m,nrhs, 0., 1.);
```

dCreate_CompCol_Matrix : This function is used to create a matrix in Compressed Column Storage (CCS) format.

dCreate_Dense_Matrix : This function is used to create a dense (Dense) matrix.

1. **dCreate_CompCol_Matrix :**
   - '&A' : Pointer to the SuperMatrix structure to be created.
   - 'm' : Number of rows in the matrix.
   - 'n' : Number of columns in the matrix.
   - 'nnz' : Number of non-zero elements.
   - 'Accs.a.data()' : Array of values of non-zero elements.
   - 'Accs.lineindex.data()' : Array of row indices.
   - 'Accs.columnptr.data()' : Array of column pointers indicating the start and end positions of each column.
   - 'SLU_NC' : Specifies the storage format (here, Compressed Column format).
   - 'SLU_D' : Specifies the data type (here, double precision).
   - 'SLU_GE' : Specifies the matrix type (here, general matrix).
2. **dCreate_Dense_Matrix :**

- '&B' : Pointer to the SuperMatrix structure to be created.
- 'm' : Number of rows in the matrix.
- 'nrhs' : Number of columns in the matrix, here it represents the number of right-hand sides to be solved.
- 'rhs.data()' : Array of matrix elements.
- 'm' : Leading dimension of the matrix, usually the number of rows.
- 'SLU_DN' : Specifies the storage format (here, Dense format).
- 'SLU_D' : Specifies the data type (here, double precision).
- 'SLU_GE' : Specifies the matrix type (here, general matrix).

```
dCreate_CompCol_Matrix(&A, m, n, nnz, Accs.a.data(), Accs.
    lineindex.data(), Accs.columnptr.data(), SLU_NC, SLU_D,
    SLU_GE);

dCreate_Dense_Matrix(&B, m, nrhs, rhs.data(), m, SLU_DN,
    SLU_D, SLU_GE);
```

The 'dgssv' function is the main function in the SuperLU library used to solve the linear system $AX = B$. This function performs the following steps :

1. Factor the matrix A to obtain its LU decomposition form.
2. Using the obtained L and U matrices, and the given right-hand side matrix BB, solve the linear system and obtain the solution X.

- '&options' : Pointer to the options structure which controls various solver settings.
- '&A' : Pointer to the SuperMatrix structure representing the matrix A.
- 'perm_c' : Array for column permutation.
- 'perm_r' : Array for row permutation.
- '&L' : Pointer to the SuperMatrix structure to store the lower triangular matrix.
- '&U' : Pointer to the SuperMatrix structure to store the upper triangular matrix.
- '&B' : Pointer to the SuperMatrix structure representing the right-hand side matrix.
- '&stat' : Pointer to the statistics structure to hold various solver statistics.
- '&info' : Integer to store the exit status of the function.

```
dgssv(&options, &A, perm_c, perm_r, &L, &U, &B, &stat, &info);
```

The **dgstrs** function solves a system of linear equations obtained by LU decomposition. After the **dgssv** function is called, **dgstrs** can be used to solve systems with different right-hand side vectors without re-performing the LU decomposition. By exchanging the order of rows and columns, **dgstrs** uses the calculated L and U matrices to solve the linear system and obtain the solution to the system of equations.

- 'trans' : Specifies whether to transpose matrix A.
- 'L & U' : The calculated lower triangular matrix L and upper triangular matrix U.
- 'perm_c & perm_r' : Column and row permutation vectors, which are obtained during LU decomposition.
- 'B' : Right matrix.
- 'stat' : SuperLU state variable, used to store some intermediate results and statistical information.
- 'info' : Returns the error message when solving the linear system. If info is 0, it means the solving process is successful.

```
dgstrs (NOTRANS,  &L, &U, perm_c, perm_r, &B, &stat, &info);
```

Since we call dmatrix_CCS to cosntruct our matrix, we also need to modify our Makefile.

```
/*Makefile*/
$(OBJ)/exampleSuperLU.o : Makefile $(SRCMAIN)/exampleSuperLU.
   cc $(SRCLIB)/dmatrix_CCS.h $(SRCLIB)/mesh.h
    $(CXX) $(CXXFLAGS) -I$(SUPERLUINC) -I$(SRCLIB) -c $(
       SRCMAIN)/exampleSuperLU.cc -o $(OBJ)/exampleSuperLU.o $
       (OBJ)/mesh.o

$(BIN)/exampleSuperLU :  Makefile $(OBJ)/exampleSuperLU.o $(
   OBJ)/dmatrix_CCS.o $(OBJ)/dmatrix_denseCM.o $(OBJ)/graph.o
   $(OBJ)/mmio.o $(OBJ)/mesh.o
    $(CXX) $(CXXFLAGS)  $(OBJ)/exampleSuperLU.o $(OBJ)/
       dmatrix_CCS.o $(OBJ)/dmatrix_denseCM.o $(OBJ)/graph.o $
       (OBJ)/mmio.o $(SUPERLULIB) -lblas    $(OBJ)/mesh.o -o $
       (BIN)/exampleSuperLU
```

3.2 comparison with the dense case and sparse case

Since SuperLU only provides two significant digits of precision to display decomposition time by default, in order to measure decomposed time, we can use the 'std : :chrono' library to get the current time before and after calling the **dgssv** function. Specific steps are as follows :

```
start = std::chrono::system_clock::now();

dgssv(&options, &A, perm_c, perm_r, &L, &U, &B, &stat, &info);

end = std::chrono::system_clock::now();
std::chrono::duration<double> elapsed_seconds = end - start;
std::cout << "Time taken for decomposition: " <<
   elapsed_seconds.count() << "s\n";
```

Below is our choice of square4.msh, which can generate a larger, not "well-banded" sparse matrix through the fem class. What we find is to use the same function dLLT_denseCM LLT_A(A) to perform column-major format for dense cases. And the CCS format solution for the sparse case, the result is that the sparse solution without reordering takes longer than the dense case, only using cuthillmckee or reverse cuthillmckee is faster, of course superLU is super fast, the following is our explanation :

- **Matrix format :** Sparse and dense matrices are stored and processed differently. Generally, sparse matrix algorithms need to process additional index information, while dense matrix algorithms are usually simpler and more direct.
- **Data access mode :** Both column-major format and CCS format store matrix elements in columns, but since the CCS format is compressed, the data access mode may be different, which may affect computing efficiency and caching effects.

Method	Cholesky Time(s)	NNZ in L	LB of L	Solution Error
Classical sparse	35.2239	139505	1317	1.42149×10^{-14}
Cuthill-McKee Ordering	0.965479	53155	65	1.35761×10^{-10}
Reversed Cuthill-McKee	0.772969	47035	65	1.31342×10^{-10}
SuperLU(default)	0.00636731	31355	-	-
Dense Case	1.41259	-	-	4.66122×10^{-15}

TABLE 6 – Performance Comparison of Different Methods for 'square4.msh'

Here we are using the SuperLU of defalut options. We will discuss the impact of options on SuperLU below.

3.3 operations of options

Depending on our needs and the complexity of the problem, we may need to adjust some settings of options. For example, if we know that matrices have the same non-zero pattern, we might want to set 'options.Fact = SamePattern' to reuse the same symbolic decomposition. If we want to evaluate the quality of the solution, we may want to enable 'options.PivotGrowth = YES' and 'options.ConditionNumber = YES'. If we wish to reduce padding, we may want to try a different column arrangement strategy, such as 'options.ColPerm = COLAMD' or 'options.ColPerm = MMD_AT_PLUS_A'. Finally, if our system is symmetric, we may want to set 'options.SymmetricMode = YES' to take advantage of this. Each option affects the performance of the decomposition and solution process and the quality of the results, so some experimentation may be required to determine the best settings.

We explain the details and possible operations as following :

```
options.Fact = DOFACT;
options.Equil = YES;
options.ColPerm = COLAMD;
options.DiagPivotThresh = 1.0;
options.Trans = NOTRANS;
options.IterRefine = NOREFINE;
options.SymmetricMode = NO;
options.PivotGrowth = NO;
options.ConditionNumber = NO;
options.PrintStat = YES;
```

1. **Fact :**
   - **DOFACT :** Perform factorization.
   - **FACTORED :** Factorization is already done, use it as is.
   - **SamePattern :** Matrix has the same non-zero pattern, reuse symbolic factorization.
   - **SamePattern_SameRowPerm :** Similar to **SamePattern** but using the same row permutations.
2. **Equil :**
   - **YES/NO :** Equilibrate the system before factorization to improve the quality of the solution.
3. **ColPerm :**
   - Column permutation strategy to reduce fill-in and improve factorization quality.
4. **DiagPivotThresh :**
   - Controls pivoting, range from 0.0 to 1.0, 0.0 almost no pivoting, 1.0 a lot of pivoting.
5. **PivotGrowth :**

- **YES/NO :** Compute pivot growth to assess the quality of the solution.

6. **ConditionNumber :**
   - **YES/NO :** Compute an estimation of the condition number.
7. **IterRefine :**
   - Level of iterative refinement : **NOREFINE**, **SLU_SINGLE**, **SLU_DOUBLE**, or **SLU_EXTRA** to improve the solution quality through iterative methods.
8. **SymmetricMode :**
   - **YES/NO :** Utilize the symmetry of the system.
9. **PrintStat :**
   - **YES/NO :** Print statistics.

Given the situation with our mass matrix, we change SymmetricMode = YES.

Metric/Setting	**Default SuperLU**	**SuperLU Symmetric**
Time for Decomposition (s)	0.00636731	0.00609398
Nonzeros in L	31355	30078
Nonzeros in U	31355	30078
Nonzeros in L+U	61365	58811
FILL Ratio	6.7	6.4

TABLE 7 – Comparison between Default SuperLU and SuperLU with SymmetricMode = YES

From the comparison, we can see that when SymmetricMode is enabled, the decomposition time is slightly reduced, and the number of non-zero elements of L and U is also reduced, resulting in an overall reduction in fill rate. This means that SymmetricMode may provide more efficient storage and computation.

A Debugging

```
///Useful commands while debugging

gdb bin/main_testCholesky_CCS

///this one we can choose mesh
gdb bin/main_testCholesky_CCS data/square.msh )
```

Références

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

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

[3] Davis, Timothy A., Sivasankaran Rajamanickam, and Wissam M. Sid-Lakhdar. 'A Survey of Direct Methods for Sparse Linear Systems'. Acta Numerica 25 (1 May 2016) : 383–566.

[4] Saad, Yousef. 2003. Iterative Methods for Sparse Linear Systems. 2nd ed. Philadelphia : Society for Industrial and Applied Mathematics.

[5] Grigori, Laura. 2015. "Sparse Direct Solvers." Lecture presented at CS267, University of California, Berkeley, Spring 2015. [link]

[6] Davis, Tim. 2006. Direct Methods for Sparse Linear Systems. Philadelphia, PA : SIAM.

[7] 'CSparse' on GitHub [link]

[8] Tim Davis. 2013. "Sparse Matrix Algorithms - combinatorics + numerical methods + applications Math + X." Lecture presented at University of Florida. [link]