SE3352: Algorithm Design December 8, 2021

Notes 1 – Matrix Decomposition

Instructor: *Guoqiang Li* Scribes: *Sipeng Zhang*

1 Introduction

Matrix Decomposition is a very large topic, generally, and there are lots of different decomposition for a given matrix. Such as LU decomposition, QR decomposition, Cholesky decomposition etc. And LU decomposition is definitely a useful and powerful one. It can be used to solve matrix equations $Ax = b$ fast and calculate the determinant of a square matrix. Today, our notes will mainly focus on the LU decomposition and its practical improvement in industry.

2 Problem description

Definition 1 (row operation)**.** *the row operation performed on matrix is one of the following:*

- *swap two rows*
- *add one row to the other*

Definition 2 (Unit Lower triangular matrix)**.** *Given a matrix L whose order is $n \times n$, L is called a Unit Lower triangular matrix if and only if for any i satisfying $1 \leq i \leq n$, $L[i, i] = 1$, and for any i, j satisfying $1 \leq i < j \leq n$, $L[i, j] = 0$.*

Definition 3 (Upper triangular matrix)**.** *Given a matrix L whose order is $n \times n$, L is called a Upper triangular matrix if and only if for any i, j satisfying $1 \leq i < j \leq n$, $L[j, i] = 0$.*

Definition 4 (LU decomposition of a matrix)**.** *Given a matrix A whose order is $n \times n$, $A = LU$ is called a LU decomposition of matrix A if and only L is a Unit Lower triangular matrix and U is an Upper triangular matrix.*

3 Algorithms

There are lots of algorithms designed to determine the LU decomposition of a matrix, we will explore some of them step by step.In our discussion below, we assume the matrix A is fully-ranked because if not we will not get the decomposition. Also, we should note that the major task of LU decomposition is to solve $Ax = b$ fast and sound, when A is not fully-marked there is no exact solution to the equation. So our assumption is appropriate.

3.1 Gaussian elimination

3.1.1 algorithm

We first describe the algorithm:

- Start from the first row and consider the first element of the row: if it is zero, swap the rows and get a nonzero first element. Note that this process only fails if and only if all the elements of the first column are zero thus A is not fully-ranked, which is not possible in our context.
- Then repeatedly do the following: for row i ($1 < i \leq n$) , if the first element of row i is zero, just skip the row and proceed to the next row. Otherwise, we consider row i with a nonzero first element X_i, then add row 1 multiply $-X_i/P_1$ to row i, P_1 is the first element of row 1,we eliminate the first element of row i to zero.
- just continue the loop until we finally get an Upper triangular matrix.

At this time, we get the U-part of our LU decomposition, to prove that we get the L-part as well, we need some facts.

Lemma 5. *swap two rows is equivalent to left multiplication a identity matrix with row i and j swapped*

Proof. In other words, P is a matrix satisfying the following property:

1. for $1 \leq k \leq n, k \neq i, j$, $P[k, k] = 1$
2. $P[j, i] = P[i, j] = 1$
3. for all other pair (i, j), $P[i, j] = 0$

We can now easily confirm that $A_t = PA$ is the same matrix as A with row i and row j swapped. □

Lemma 6. *add one row to the other is equivalent to left multiplication a identity matrix with row i added to row j.*

Proof. The P satisfies following properties

1. for $1 \leq k \leq n$, $P[k, k] = 1$.
2. $P[j, i] = 1$.
3. for all other pair (i, j),$P[i, j] = 0$.

Again, we can do the matrix multiplication and confirm that $A_t = PA$ is the same as A with row i added to row j. □

Theorem 7. *the row operations performed on matrix are equivalent to pre-multiply a certain matrix P.*

Proof. We finish the proof with two kinds of row operations we defined above. □

Theorem 8. *the product of two Unit lower triangular matrix is a Unit lower triangular matrix.*

Proof. let A and B are Unit lower triangular matrix. We compute the product of A and B, let C = AB.

1. we compute all diagonal elements of C. C[i,i]=$\Sigma_{k=1}^{k=n} A[i,k]B[k,i]$. Note that A,B are Unit lower triangular matrix, so for $1 \leq k < i$, B[k,i] = 0 and for $i < k \leq n$, A [i,k] = 0, B[i,i]=1 and A[i,i]= 1. Use the three facts above, we get C[i,i]=1.

2. we now prove for $1 \leq i < j \leq n$, C[i,j]=0. C[i,j]=$\Sigma_{k=1}^{k=n} A[i,k]B[k,j]$, since for $1 \leq k < j$, B[k,j] = 0 and for $i < k \leq n$, A [i,k] = 0, So $1 \leq k < j$, A[i,k]B[k,j] = 0 and for $i < k \leq n$, A [i,k]B[k,j] = 0, we get C[i,j] = 0

□

Then, we clarify that we first do the row swap such that during the elimination, the diagonal element of A is nonzero. as we have proved this leads to PA, then we do the elimination by adding one row to the other. According the theorem 7, this can be done by a series of pre-multiply: $E_1E_2...E_nPA = U$, and by theorem 8, we get $PA = E_n^{-1}...E_2^{-1}E_1^{-1}U = LU$, note that the redundant P does not interfere with our goal because in practice we just first do the swap.

3.1.2 Complexity

We can easily get the Complexity of Gaussian Elimination is $O(n^3)$, because we eliminate every column use $O(n^2)$ multiplication and addition. This complexity in real world is not acceptable in that in practice n is incredibly large, typically 10^5. And another problem is the conditional number of the procedure above is very large which means the solution is susceptible to inaccuracy of the data caused by float point number operation.

3.2 Left Looking Elimination

Let us derive a left looking version of Gaussian elimination. We can further extend this approach to our real world practice. For now, given an input matrix A with order of $n \times n$ be represented as a product of two triangular matrices L and U. We write A as follow:

$$\begin{pmatrix} A_{11} & \boldsymbol{\alpha_{12}} & A_{13} \\ \boldsymbol{\alpha_{21}} & a_{22} & \boldsymbol{\alpha_{23}} \\ A_{31} & \boldsymbol{\alpha_{32}} & A_{33} \end{pmatrix} = \begin{pmatrix} L_{11} & 0 & 0 \\ \boldsymbol{l_{21}} & 1 & 0 \\ L_{31} & \boldsymbol{l_{32}} & L_{33} \end{pmatrix} \times \begin{pmatrix} U_{11} & \boldsymbol{u_{12}} & U_{13} \\ 0 & u_{22} & \boldsymbol{u_{23}} \\ 0 & 0 & U_{33} \end{pmatrix} \tag{1}$$

where A_{ij} is a block,$\boldsymbol{\alpha_{ij}}$ is a vector and a_{ij} is a scalar. The dimensions of different elements in the matrices are as follows:

- A_{11}, L_{11}, U_{11} are $k \times k$ blocks
- $\boldsymbol{\alpha_{12}}, \boldsymbol{u_{12}}$ are $k \times 1$ vectors
- A_{13}, U_{13} are $k \times n-(k+1)$ blocks
- $\boldsymbol{\alpha_{21}}, \boldsymbol{l_{21}}$ are $1 \times k$ row vectors
- a_{22}, u_{22} are scalars
- $\boldsymbol{\alpha_{23}}, \boldsymbol{u_{23}}$ are $1 \times n-(k+1)$ row vectors
- A_{31} L_{31} are $n-(k+1) \times k$ blocks

- $\boldsymbol{\alpha_{32}}, \boldsymbol{l_{32}}$ are $n-(k+1)\times 1$vectors
- A_{33}, L_{33}, U_{33} are $n-(k+1)\times n-(k+1)$ blocks

Now, we calculate the equation (1), get the following equations:

$$L_{11} \times U_{11} = A_{11} \tag{2}$$
$$L_{11} \times \boldsymbol{u_{12}} = \boldsymbol{\alpha_{12}} \tag{3}$$
$$L_{11} \times U_{13} = A_{13} \tag{4}$$
$$\boldsymbol{l_{21}} \times U_{11} = \boldsymbol{\alpha_{21}} \tag{5}$$
$$\boldsymbol{l_{21}} \times \boldsymbol{u_{12}} + u_{22} = a_{22} \tag{6}$$
$$\boldsymbol{l_{21}} \times U_{13} + \boldsymbol{u_{23}} = \boldsymbol{\alpha_{23}} \tag{7}$$
$$L_{31} \times U_{11} = A_{31} \tag{8}$$
$$L_{31} \times \boldsymbol{u_{12}} + \boldsymbol{l_{32}} \times u_{22} = \boldsymbol{\alpha_{32}} \tag{9}$$

We need to do some clarification here, the method is called left-looking elimination because it computes the k^{th} column of L and U using their $1, 2, ..., k-1$ columns. It just like a induction process, so the parameter mentioned above is exactly the k^{th} column we want to compute, and suppose we already know the first $k-1$ columns. Now we can extract the parts that interest us get the following equation:

$$\begin{pmatrix} L_{11} & 0 & 0 \\ \boldsymbol{l_{21}} & 1 & 0 \\ L_{31} & 0 & 1 \end{pmatrix} \times \begin{pmatrix} \boldsymbol{u_{12}} \\ u_{22} \\ \boldsymbol{l_{32}} \times u_{22} \end{pmatrix} = \begin{pmatrix} \boldsymbol{\alpha_{12}} \\ a_{22} \\ \boldsymbol{\alpha_{32}} \end{pmatrix} \tag{10}$$

Here we can see why we just need these parts:
$\boldsymbol{u_{12}}, u_{22}, \boldsymbol{l_{32}}$ is all what we need to get the k^{th} column of L and U, and it is clear how to get them:

1. compute $L_{11} \times \boldsymbol{u_{12}} = \boldsymbol{\alpha_{12}}$, we get $\boldsymbol{u_{12}}$
2. compute $u_{22} = a_{22} - \boldsymbol{l_{21}} \times \boldsymbol{u_{12}}$, we get u_{22}
3. compute $\boldsymbol{l_{32}} = \frac{1}{u_{22}}(\boldsymbol{\alpha_{32}} - L_{31} \times \boldsymbol{u_{12}})$,we get $\boldsymbol{l_{32}}$

And the step 2 and 3 are very easy, they just involve multiplication and addition. In step 1, we will solve a matrix equation. Note L_{11} is the upper $k \times k$ blocks of Unit lower triangular L, so L_{11} is also Unit lower triangular. Hence we can derive a simple algorithm8 to compute $\boldsymbol{u_{12}}$: We use this method to get the solution, combined with above process, we can get the LU decomposition of A use n iteration.

3.2.1 Complexity

Every time we compute $\boldsymbol{u_{12}}$, we use $O(n+$number of multiplications performed$)$, and we have n iterations, so it seems likely to be $O(n^3)$, but the vector addition can be improved to nearly $O(1)$, thus we can think this approach can achieve at most $O(n^2)$. Although far more better than the first one, it is not practical to put into use in industry as well.

Algorithm 1: Compute u_{12}

```
x ← b;
for i = 1 to n do
    if x(i) ≠ 0 then
        for j = i + 1 to n do
            x(j) = x(j) − L(j, i) ∗ x(i)
        end
    end
end
```

3.3 A Practical method:Gilbert Peierls Algorithm

3.3.1 Brief

Methods for LU decomposition we have mentioned above are not very efficient in large scale input, which is often the case in industry. And in real practice, a certain kind of problems share some common characteristics. Gilbert Peierls Algorithm is one of the algorithms that exploit these characteristic. Almost all further improvement algorithms are based on the idea proposed by Gilbert Peierls Algorithm.

Definition 9 (Sparse matrix)**.** *A matrix is called sparse if the nonzero elements is very few. In practice, we usually call a matrix sparse if it has $O(n)$ nonzero elements*

Typically, we store the matrix in a two dimension array. But when the matrix is sparse, most of the entries in the array is zero, thus we do not need to store them. We usually store sparse matrix using the column compressed form in the following definition and it only takes $O(n)$ space.

Definition 10 (Column compressed form)**.** *A column compressed form of a matrix consists of three vectors:A_p, A_i, A_x*

column	definition
A_x	contains all the nonzero elements of A, from column 1 to column n
A_i	contains all the nonzero elements column index, from column 1 to column n
A_p	contains all the ending index of a given column in A_i, except the first element is always 0

Example 11. *Given matrix*

$$\begin{pmatrix} 5 & 0 & 0 \\ 4 & 2 & 0 \\ 3 & 1 & 8 \end{pmatrix}$$

when presented in column compressed format it will be

column	value
A_x	$5, 4, 3, 2, 1, 8$
A_i	$0, 1, 2, 1, 2, 2$
A_p	$0, 3, 5, 6$

3.3.2 Gilbert Peierls Algorithm

The algorithm aims at decomposing an arbitrary non singular sparse matrix A as PA = LU(recall that we just do the row swap first) in time proportion to the flop count of the L and U

Definition 12 (flops(LU))**.** *the symbol flops(LU) is the number of nonzero multiplications performed when multiplying two matrices L and U*

We analysis the complexity of the algorithm based on the flops(LU). Note that we store our matrix in column compressed form and it means we must calculate the nonzero entries in L and U at the same time. It consists of two stages for determining every column of L and U. The first stage is a symbolic analysis that computes the nonzero pattern of the column k of the L and U. The second stage is the numerical factorization stage that involves solving the lower triangular system $Lx = b$

3.3.3 Symbolic analysis

Let us first recall our naive approach to solve $Lx = b$.

```
1 x ← b;
2 for i = 1 to n do
3 |   if x(i) ≠ 0 then
4 |   |   for j = i + 1 to n do
5 |   |   |   x(j) = x(j) − L(j, i) ∗ x(i)
6 |   |   end
7 |   end
8 end
```

The above algorithm takes time $O(n + \text{number of flops performed})$. The $O(n)$ term looks harmless, but $Lx = b$ is solved n times when we use left-looking elimination to get every column of L and U, leading to an unacceptable $O(n^2)$ term in the work to decompose A into L an U. To remove the $O(n)$ term, we must replace the algorithm with follow one which would reduce the $O(n)$ term to $O(\eta(x))$, where $\eta(x)$

```
1 x ← b;
2 for all i satisfying x(i) ≠ 0 do
3 |   for j = i + 1 to n do
4 |   |   x(j) = x(j) − L(j, i) ∗ x(i)
5 |   end
6 end
```

is the number of nonzero elements in x. Thus to solve $Lx = b$, we need to know the nonzero pattern of x before we compute x itself. Symbolic analysis helps us determine the nonzero pattern of x and all the improved algorithm uses the symbolic analysis as well.

Theorem 13. *Let $G = G(L_k)$ be the directed graph of L with $k-1$ vertices representing the already computed $k-1$ columns. $G(L_k)$ has an edge $j \to i$ if and only if $l_{ij} \neq 0$. Let $\beta = \{i | b_i \neq 0\}$ and $X = \{i | x_i \neq 0\}$, Now the elements of X is given by*

$$X = Reach_{G(L)}(\beta)$$

In other words, the nonzero pattern of X is computed by determining the vertices that are reachable from the vertices of the set β.

Proof. The proof of Theorem 10 is too tricky so we won't cover it here, you can find it in reference [3]. □

Now, the reachability problem can be solved using a classical depth first search in $G(L_k)$ from the vertices of the set β. During the depth first search, we also get a topological order of X (because the graph is directed). The topological order is useful for eliminating unknowns in the next step.

3.3.4 Numerical Factorization

Numerical factorization consists of solving the equation 10 for each column k of L and U. Normally we would solve for the unknowns in equation 10 in increasing order of the row index. The row indices/nonzero pattern computed by depth first search are not necessarily in increasing order. Sorting the indices would increase the time complexity beyond $O(flops(LU))$. However, the requirement of eliminating unknowns in increasing order can be relaxed to a topological order of the row indices. An unknown x_i can be computed, once all the unknowns x_j of which it is dependent on are computed. So, the unknowns can be solved in any topological order. The depth first search algorithm gives one such topological order which is sufficient for our case.

3.3.5 Complete picture

The whole algorithm can be summed to the following one:

```
1  L ← I;
2  for i = 1 to n do
3  |   X ← L\k_th column of A;
4  |   for j = 1 to i do
5  |   |   U(j,k) = x(j)
6  |   end
7  |   for j = k to n do
8  |   |   L(j,k) = x(j)/U(k,k)
9  |   end
10 end
```

$x = L\backslash b$ denotes the solution of: $Lx = b$. In this case, b is the k^{th} column of A. The total time complexity of the algorithm is $O(\eta(A) + flops(LU))$. $\eta(A)$ is the number of nonzeros in the matrix A and $flops(LU)$ is the flop count of the product of the matrices L and U. Recall that we also do a depth first search, it takes time proportion to the number of nodes in the graph, nearly $O(n)$. Here we claim that $flops(LU)$ is less than the nonzero element in L or U, it seems that this may reach an order of $O(n^2)$, but in practice it works quite fast, slightly above $O(n)$ according to many experiments in [2] and [1]. So here the $flops(LU)$ dominant the formula and we can assume it runs under $O(flops(LU))$ time complexity.

3.3.6 Test

To illustrate the better efficiency of Sparse LU decomposition over Gaussian elimination, we write a simple program in python. It calculate the LU decomposition of a sparse matrix of size 10000x10000

based on Sparse LU (SuperLU in python) and Gaussian elimination (scipy.linalg.Lu), and see major gap between the two methods. The time of Normal LU is 7.8711s while the sparse LU is only 0.0034s. The code can be reviewed in the last part.

4 Conclusion

4.1 summary for Gilbert Peierls Algorithm

The Gilbert Peierls Algorithm is the base decomposition algorithm for scientific computing package such as Matlab, numpy, OpenBLAS and SuiteSparse. We can conclude why it get favors of industry. Firstly, it exploit the fundamental properties that many matrices in industry are sparse, so we can use column compressed form to store the large matrices, which saves much space and assure less costs. Also, the algorithm is specially designed for sparse matrices as we can see if the $\eta(A)$ term in the complexity is large(in other words, a dense matrix), this algorithm will only incur more time complexity than previous one. So we must design algorithms that suit for a given situation and exploit special properties as much as we can to reduce complexity of problem.

4.2 future direction

4.2.1 Symmetric Pruning

As we can see in the complexity analysis of Gilbert Peierls Algorithm, the time needed to do the depth first search may be a barrier of short running time. So we may improve the algorithm by cutting the symbolic analysis time. The cost of depth first search can be cut down by pruning unnecessary edges in the graph of G(L). The idea is to replace $G(L)$ by a reduced graph by exploiting symmetry of A.

4.2.2 Ordering

Note that $O(flops(LU))$ is the dominant term in our complexity analysis and during the row swap, if we use different swap strategy that all satisfy there is no zero in the diagonal during elimination, we can get different L and U. So a natural way to improve is choose a best swap strategy that minimize the nonzero elements in L and U, thus reduce the $O(flops(LU))$ term. This future direction is called ordering.

References

[1] John R Gilbert and Tim Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM journal on scientific and statistical computing*, 9(5):862–874, 1988.

[2] Ekanathan Palamadai Natarajan. *KLU–A high performance sparse linear solver for circuit simulation problems*. PhD thesis, University of Florida, 2005.

[3] Donald J Rose, R Endre Tarjan, and George S Lueker. Algorithmic aspects of vertex elimination on graphs. *SIAM Journal on computing*, 5(2):266–283, 1976.

5 Appendix

```python
import numpy as np
from scipy import sparse
from scipy.sparse import csc_matrix, linalg as sla
from scipy.linalg import lu
import time


sparse_matrix=np.zeros((10000,10000))
for i in range(10000):
    sparse_matrix[i % 10000][(i + 1) % 10000] = 1

b = csc_matrix(sparse_matrix)
splu_start = time.time()
res = sla.splu(b)
splu_end = time.time()
lu_start = time.time()
p,l,u = lu(sparse_matrix)
lu_end = time.time()

print(f"use time:{splu_end - splu_start}")
print(f"use time:{lu_end - lu_start}")
```