# MATH5312 - Advanced Numerical Methods II

**Taught by Jiafeng Cai**

**Notes by Aaron Wang**

## Contents

# 1 February 2nd, 2021

## 1.1 Class Logistics

This course is a continuation of MATH5311. In MATH5311, we introduced how to discretize differential equations. In the course, we focus on the algebraic side of numerical analysis, introducing some more advanced solvers for these numerical methods.

### 1.1.1 Assessment Scheme

**Biweekly Homework** - 60%

**Take-home Final Exam** - 40%

### 1.1.2 Reference Books

**Lecture Notes** - Main content that will be covered

**Youset Saad: Iterative Methods for Sparse Linear Systems** - Free ebook can be found on Canvas

**Golub, Van Lean: Matrix Computations**

**Trefethen, Bau: Numerical Linear Algebra**

## 1.2 Direct Methods for Systems of Lienar Equations

The goal of this chapter is very simple, solve:

$$Ax = b$$

where

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \in \mathbb{R}^{n \times n}$$

and:

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^n$$

which is the unknown vector, and:

$$b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \in \mathbb{R}^n$$

is the right hand side (RHS).

The solution of system of linear equations is a fundamental task in numerical analysis. If we want to solve a non-linear equation, we can do it iteratively, where each step is solving a linear system. In this chapter, we will introduce **Direct Methods**.

**Definition 1.1.** A **Direct Method** is a method in which, if there is no truncation error in the computer, can solve the system exactly in finite time.

One example of a direct method is **Gaussian Elimination**.

**Example 1.2**

Consider:

$$\begin{cases} 2x_1 + x_2 - x_3 = 1 \\ 4x_1 + 5x_2 - 3x_3 = -3 \\ -2x_1 + 5x_2 - 2x_3 = -8 \end{cases}$$

To perform Gaussian Elimination, we eliminate $x_1$ from Equations 2 and 3 by subtracting Equation 1, and then $x_2$ from Equation 3 by subtracting Equation 2. Doing so, we get:

$$\begin{cases} 2x_1 + x_2 - x_3 = 1 \\ 3x_2 - x_3 = -5 \\ -x_3 = 3 \end{cases}$$

After this, we can solve $x_3$ and then using its value solve $x_2$ and $x_3$.

Typically we use matrix terminology to reformulate the methods. We do this by writing the **augmented matrix** of the above linear system:

$$\left[\begin{array}{ccc|c} 2 & 1 & -1 & -1 \\ 4 & 5 & -2 & -3 \\ -2 & 5 & -3 & -8 \end{array}\right]$$

**Lemma 1.3**

Row Operations are equivalent to left matrix multiplications.

**Example 1.4**

In Example 1.2, the row operation to eliminate $x_1$ for Equations 2 and 3 can be represented as:

$$\begin{bmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix} \left[\begin{array}{ccc|c} 2 & 1 & -1 & -1 \\ 4 & 5 & -2 & -3 \\ -2 & 5 & -3 & -8 \end{array}\right] = \left[\begin{array}{ccc|c} 2 & 1 & -1 & -1 \\ 0 & 3 & -1 & -5 \\ 0 & 6 & -3 & -7 \end{array}\right]$$

As such, if $A$ is the augmented matrix and we want to solve the system $Ax = b$, we have:

$$L_{n-1} \dots L_1 A x = L_{n-1} \dots L_1 b$$

and until the LHS is an upper triangular matrix.

**Remark 1.5 —** Note that the row operations $L_i$ are lower triangular matrix. In particular, it is a rank 1 modification of the identity matrix:

$$L_i = I - ve_i^T$$

where $v \in \mathbb{R}^n$ and $v_1 = v_2 = \ldots = v_i = 0$ and $e_i$ is the $i$-th unit vector.

**Definition 1.6.** $L_i$ are called **Gauss Transformations**.

In summary, Gaussian Elimination consists of the following steps:

1. Left multiply Gaussian Transformations to reduce $A$ to an upper triangular matrix:

$$A^{(1)} = A = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \ldots & a_{1n}^{(1)} \\ a_{21}^{(1)} & a_{22}^{(1)} & \ldots & a_{2n}^{(1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1}^{(1)} & a_{n2}^{(1)} & \ldots & a_{nn}^{(1)} \end{bmatrix} \text{ and } b^{(1)} = b = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ \vdots \\ b_n^{(1)} \end{bmatrix}$$

   In step 1, we have:

$$L_1 = \begin{bmatrix} 1 & & & \\ -\frac{a_{21}^{(1)}}{a_{11}^{(1)}} & 1 & & \\ \vdots & & \ddots & \\ -\frac{a_{n1}^{(1)}}{a_{11}^{(1)}} & 0 & \ldots & 1 \end{bmatrix} = I - \ell_1 e_1^T, \quad \ell 1 = \begin{bmatrix} 0 \\ \frac{a_{21}^{(1)}}{a_{11}^{(1)}} \\ \vdots \\ \frac{a_{n1}^{(1)}}{a_{11}^{(1)}} \end{bmatrix}$$

   After step 1, we have:

$$L_1 A^{(1)} = A^{(2)} = \begin{bmatrix} a_{11}^{(1)} & a_{12}^{(1)} & \ldots & a_{1n}^{(1)} \\ 0 & a_{22}^{(2)} & \ldots & a_{2n}^{(2)} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & a_{n2}^{(2)} & \ldots & a_{nn}^{(2)} \end{bmatrix} \text{ and } L_1 b^{(1)} = b^{(2)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(2)} \\ \vdots \\ b_n^{(2)} \end{bmatrix}$$

   We can continue this process until the LHS is upper left triangular. The pseudocode of this first part is given in 1.

2. For part 2, we solve the resulting upper triangular linear system:

$$A^{(n)} x = b^{(n)}$$

   This is quite straightforward, as we just start from the last row and solve each equation, since we are only introducing one unknown with each equation. The pseudocode of this second part is given in 2.

**Remark 1.7 —** Note that a naive implementation would take $O(n^3)$ memory, as three indexes are used for $a_{ij}^{(k)}$.

We can implement this more efficiently by dropping the superscript, since we do not change it. In other words, we use $a_{ij}^{(k)}$ to overwrite $a_{ij}$. This will take up the upper triangular part of the matrix. Since the lower triangular part of the matrix is 0, we can use write $\ell_{ik}$ to this memory. If we do not need $b$ anymore, we can overwrite it as well, meaning that we can do Gaussian Elimination without requiring additional memory.

**Algorithm 1** Gaussian Elimination Part 1

```
for k = 1 : n − 1 do
    for i = k + 1 : n do
        ℓ_ik = a_ik^(k) / a_kk^(k)
    end for
    for i = k + 1 : n do
        for j = k + 1 : n do
            a_ij^(k+1) = a_ij^(k+1) − ℓ_ik a_kj^(k)        ▷ Compute A^(k+1) = L_k A^(k)
        end for
    end for
    for i = k + 1 : n do
        b_i^(k+1) = b_i^(k+1) − ℓ_ik b_i^(k)               ▷ Compute b^(k+1) = L_k b^(k)
    end for
end for
```

**Algorithm 2** Gaussian Elimination Part 2

```
for k = n : −1 : 1 do
    for j = n : −1 : k + 1 do
        y_k = y_k − u_kj x_j
    end for
    x_k = y_k / a_kk
end for
```

# 2 February 4th, 2021

## 2.1 Efficient Gaussian Elimination

Last time, we introduced a memory efficient way to perform Gaussian Elimination by reusing the inputs. However, this implementation is **not** the most time efficient. To do this, we need to consider the computer architecture. Recall that computers have a few different layers of memory:

- CPU register
- Cache
- RAM
- Disk

The lower the level, the larger the memory but communicating is slower. As such, to improve the efficiency, we should minimize the communication between the different levels of memory.

**Remark 2.1 —** This principle of memory management should always be considered when performing matrix multiplication.

Note that we can reformulate the previous algorithms into matrix operations. First we perform a vector scaling, then we perform a rank 1 update, and then a matrix vector

product. Since these are very basic operations, there are many very efficient implementations of these **Basic Linear Algebra Subroutines (BLAS)**. There are a few different levels of these BLAS.

**BLAS1** : Vector operations, e.g.

$$\beta = x^T y, y = \beta x + y, \quad \text{where } x, y \in \mathbb{R}^n, !\beta \in \mathbb{R}$$

**BLAS2** : Matrix-Vector operations

$$y = Ax + y, \quad \text{where } A \in \mathbb{R}^{m\times n}, \ x, y \text{ are vectors}$$

**BLAS3** : Matrix-Matrix operations

$$C = AB + C, \quad \text{where } A, B, C \text{ are matricies}$$

These BLAS are implemented to be optimized on the given computer architecture.

**Remark 2.2 —** Since BLAS3 can be constructed using BLAS2, etc. in terms of the efficiency, we have:

$$\text{BLAS3} > \text{BLAS2} > \text{BLAS1}$$

where efficiency is the time needed

**Remark 2.3 —** BLAS will come with the CPU.

**AMD** Core Math Library (ACML)

**Intel** Math Kernel Lib (MKL)

Replacing the algorithm with BLAS, we have:

**Algorithm 3** Gaussian Elimination with BLAS

**for** $k = 1 : n-1$ **do**
  $A(k+1:n, k) = A(k+1:n, k)/A(k, k)$
  $A(k+1:n, k+1:n) = A(k+1:n, k+1:n) - A(k+1:n, k)A(k, k+1:n)$
  $b(k+1:n, 1) = b(k+1:n, 1) - b(k.1)A(k+1:n, k)$
**end for**
**for** $k = n : -1 : 1$ **do**
  $b(k) = b(k) - A(k, k+1:n)b(k+1:n)$
  $b(k) = b(k)/A(k, k)$
**end for**

## 2.2 Complexity of Gaussian Elimination

For this,we want to see how many scalar operations are needed. Counting the cost, we have:

$$\sum_{k=1}^{n-1} \left[(n-k) + 2(n-k)^2 + 2(n-k)\right] + \sum_{k=1}^{n} \left[2(n-k)+1\right]$$

$$\begin{aligned}
&\sum_{k=1}^{n-1}\left[(n-k)+2(n-k)^2+2(n-k)\right]+\sum_{k=1}^{n}\left[2(n-k)+1\right]\\
&=3\sum_{k=1}^{n-1}(n-k)+2\sum_{k=1}^{n-1}(n-k)^2+2\sum_{k=1}^{n-1}k+\sum_{k=1}^{n}1\\
&=\frac{2}{3}n^3+\frac{3}{2}n^2-\frac{1}{3}n\\
&=\frac{2}{3}n^3+O(n^2)\\
&=O(n^3).
\end{aligned}$$

**Theorem 2.4**

The computational complexity of Gaussian Elimination is $O(n^3)$.

## 2.3 LU Decomposition view of Gaussian Elimination

Remember we can express Gaussian Elimination as:

$$L_{n-1}\dots L_2L_1Ax=L_{n-1}\dots L_1b$$

Note that $L_{n-1}\dots L_2L_1A=A^{(n)}$ which is upper triangular. Let us denote this by:

$$U=\begin{bmatrix}u_{11}&u_{12}&\dots&u_{1n}\\&u_{22}&\dots&u_{2n}\\&&\ddots&\vdots\\&&&u_{nn}\end{bmatrix}\in\mathbb{R}^{n\times n}$$

Note that since each $L_i$ is a Gaussian transformation which is invertable, we have:

$$A=L_1^{-1}L_2^{-1}\dots L_{n-1}^{-1}U$$

**Theorem 2.5**

$$L_k^{-1}=I+\ell_ke_k^T$$

*Proof.*

$$\begin{aligned}
&(I-\ell_ke_k^T)(I+\ell_ke_k^T)\\
&=I-\ell_ke_k^T+\ell_ke_k^T-\ell_ke_k^T\ell_ke_k^T\\
&=I.
\end{aligned}$$

□

**Theorem 2.6**

$$L_1^{-1}L_2^{-1}\dots L_{n-1}^{-1} = I + \sum_{i=1}^{n-1} \ell_i e_i^T$$

which is lower triangular.

*Proof.* Direct computation. □

**Theorem 2.7**

Gaussian Elimination gives:

$$A = LU$$

where $L$ is unit lower triangular, and $U$ is upper triangular. This is called the **LU Decomposition**.

**Remark 2.8 —** Note that in the memory efficient implementation of GE, we are computing $L$ and $U$ and it is stored in the original matrix.

With this, we have:

$$Ax = b \iff LUx = b \iff \begin{cases} Ly = b \\ Ux = y \end{cases}$$

**Theorem 2.9**

The computation cost is: $\begin{cases} \text{LU decomposition} & \frac{2}{3}n^3 + O(n^2) \\ \text{Forward Substitution} & O(n^2) \\ \text{Backward Substitution} & O(n^2) \end{cases}$

**Remark 2.10 —** If we use the same coefficient matrix, we only need to compute the decomposition once.

This LU decomposition view of Gaussian Elimination has a number of advantages:

1. If we want to solve $Ax_i = b_i$ for $i = 1, \dots m$, then we only need to do LU decomposition once
2. LU decomposition can be used for other matrix computation tasks. e.g.
   (a) $\det(A) = \det(LU) = \det(L)\det(U) = u_{11}u_{22}\dots u_{nn}$
   (b) $A^{-1} = U^{-1}L^{-1}$

# 3 February 9th, 2021

## 3.1 Pivoting LU Decomposition

**Example 3.1**

Consider $Ax = b$:

$$\begin{bmatrix} 2 & 1 & -1 \\ 4 & 5 & -3 \\ -2 & 5 & -2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -3 \\ -8 \end{bmatrix}$$

At each step, we have:

1. $$A = \begin{bmatrix} 2 & 1 & -1 \\ 2 & 3 & -1 \\ -1 & 6 & -3 \end{bmatrix}$$

2. $$A = \begin{bmatrix} 2 & 1 & -1 \\ 2 & 3 & -1 \\ -1 & 2 & -1 \end{bmatrix}$$

Thus, we have:

$$L = \begin{bmatrix} 1 & & \\ 2 & 1 & \\ -1 & 2 & 1 \end{bmatrix} \quad U = \begin{bmatrix} 2 & 1 & -1 \\ & 3 & -1 \\ & & -1 \end{bmatrix}$$

Now we solve $Ly = b$ by forward substitution:

$$\begin{bmatrix} 1 & & \\ 2 & 1 & \\ -1 & 2 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -3 \\ -8 \end{bmatrix} \implies \begin{cases} y_1 = 1 \\ y_2 = -5 \\ y_3 = 3 \end{cases}$$

Then we solve $Ux = y$ by back substitution:

$$\begin{bmatrix} 2 & 1 & -1 \\ & 3 & -1 \\ & & -1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 1 \\ -5 \\ 3 \end{bmatrix} \implies \begin{cases} y_1 = 1/3 \\ y_2 = -8/3 \\ y_3 = -3 \end{cases}$$

In LU Decomposition, $A(k,k)$ at step $k$ is in the denominator, which means:

1. The procedure cannot continue if $A(k,k) = 0$

   **Example 3.2**

   $$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

2. If $A(k,k)$ is small, then the computation becomes inaccurate.

As such, LU Decomposition is unstable unless $A(k,k)$ is large.

**Definition 3.3.** We call $A(k,k)$ at step $k$ a **pivot entry**.

In order to avoid small pivots, we use "pivoting", where we interchange the row or columns of the matrix. There are many pivoting schemes, for example Row Pivoting, where at each step $k$, we choose the largest entry in the $k$ column under $A(k,k)$.

**Algorithm 4** Row Pivoting

1: **for** $k = 1 : n-1$ **do**
2: Find the max entry in abs. value in $A(k:n,k)$ denoted by $i_k$
3: $A(i_k,:) \iff A(k,:)$ ▷ Swap row $i_k$ and $k$
4: $A(k+1:n,k) = A(k+1:n,k)/A(k,k)$
5: $A(k+1:n,k+1:n) = A(k+1:n,k+1:n) - A(k+1:n,k)A(k,k+1:n)$
6: **end for**

Row interchanging (row $i_k \iff$ row $k$) is equivalent to Left multiplying by a permutation matrix:

$$P_k = \begin{bmatrix} 1 & & & & & & \\ & \ddots & & & & & \\ & & 0 & & 1 & & \\ & & & \ddots & & & \\ & & 1 & & 0 & & \\ & & & & & \ddots & \\ & & & & & & 1 \end{bmatrix}$$

Thus when we perform the LU decomposition, we have:

$$L_{n-1}P_{n-1}\dots L_2P_2L_1P_1A = U$$

However, this is too complicated, so we will do some simplification. It is easy to check that $P_kP_k = I$, since we are swapping the two rows again, and that $P_k^T = P_k$.

**Lemma 3.4**

$$P_{n-1}P_{n-2}\dots P_{k+1}L_kP_{k+1}\dots P_{n-2}P_{n-1} = I - (P_{n-1}P_{n-2}\dots P_{k+1}\ell_k)e_k^T$$

*Proof.* We have:

$$\begin{aligned} L_k &= I - \ell e_k^T \\ P_{k+1}L_kP_{k+1} &= P_{k+1}(I - \ell_k e_k^T)P_{k_1} = I - P_{k+1}\ell_k e_k^T P_{k+1} \\ &= I - (P_{k+1}\ell_k)e_k^T \\ &\vdots \end{aligned}$$

□

If we denote $P_{n-1}P_{n-2}\dots P_{k+1}\ell_k = \tilde{\ell_k}$, we have:

$$\begin{aligned} &L_{n-1}P_{n-1}\dots L_2P_2L_1P_1A \\ &= L_{n-1}(P_{n-1}L_{n-2}P_{n-1})(P_{n-1}P_{n-2}\dots L_2P_2L_1P_1A \\ &= \tilde{L_{n-1}}\tilde{L_{n-2}}\dots\tilde{L_1}P_{n-1}P_{n-2}\dots P_1A. \end{aligned}$$

Denoting $P_{n-1}P_{n-2}\dots P_1 = P$, we have:

$$PA = \tilde{L_1^{-1}}\tilde{L_2^{-1}}\dots\tilde{L_{n-1}^{-1}}U$$

Giving us

$$PA = LU$$

**Example 3.5**

Using the same example as before, $A = \begin{bmatrix} 2 & 1 & -1 \\ 4 & 5 & -3 \\ -2 & 5 & -2 \end{bmatrix}$ we have:

1. $$\begin{bmatrix} 2 & 1 & -1 \\ 4 & 5 & -3 \\ -2 & 5 & -2 \end{bmatrix} \xrightarrow{\text{row 1}\leftrightarrow\text{row 2}} \begin{bmatrix} 4 & 5 & -3 \\ 2 & 1 & -1 \\ -2 & 5 & -2 \end{bmatrix} \to \begin{bmatrix} 4 & 5 & -3 \\ \frac{1}{2} & -\frac{3}{2} & \frac{1}{2} \\ -\frac{1}{2} & \frac{15}{2} & -\frac{7}{2} \end{bmatrix}$$

2. $$\begin{bmatrix} 4 & 5 & -3 \\ \frac{1}{2} & -\frac{3}{2} & \frac{1}{2} \\ -\frac{1}{2} & \frac{15}{2} & -\frac{7}{2} \end{bmatrix} \xrightarrow{\text{row 2}\leftrightarrow\text{row 3}} \begin{bmatrix} 4 & 5 & -3 \\ \frac{1}{2} & -\frac{3}{2} & \frac{1}{2} \\ -\frac{1}{2} & \frac{15}{2} & -\frac{7}{2} \end{bmatrix} \to \begin{bmatrix} 4 & 5 & -3 \\ -\frac{1}{2} & \frac{15}{2} & -\frac{7}{2} \\ \frac{1}{2} & -\frac{1}{5} & -\frac{1}{5} \end{bmatrix}$$

Giving us an output:

$$PA = \begin{bmatrix} 4 & 5 & -3 \\ -\frac{1}{2} & \frac{15}{2} & -\frac{7}{2} \\ \frac{1}{2} & -\frac{1}{5} & -\frac{1}{5} \end{bmatrix}$$

With this, we have:

$$L = \begin{bmatrix} 1 & & \\ -\frac{1}{2} & 1 & \\ \frac{1}{2} & -\frac{1}{5} & 1 \end{bmatrix} \quad U = \begin{bmatrix} 4 & 5 & -3 \\ & \frac{15}{2} & -\frac{7}{2} \\ & & -\frac{1}{5} \end{bmatrix} \quad P = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{bmatrix}$$

**Remark 3.6 —** All entries in $L$ have an abs. value smaller than 1.

There are also other pivoting strategy, such as **full pivoting** where you swap the columns as well go get the maximum pivot in the lower submatrix. This would give us:

$$PAQ = LU$$

where $Q$ is also a permutation matrix.

**Remark 3.7 —** Full pivoting not only rearranges the equation, but it also rearranges the unknowns.

**Remark 3.8 —** Full pivoting is more stable than row pivoting but it is computationally more expensive.

For any non-singular matrix, we can solve it using pivoting LU decomposition. However, we can consider the structure of the matrix to improve the algorithm.

## 3.2 LU Decomposition on SPD Matrices

**Definition 3.9.** $A \in \mathbb{R}^{n\times n}$ is **symmetric positive definite (SPD)** if it is:

- Symmetric: $A = A^T$
- Positive Definite: $x^T Ax > 0$ for all $x \in \mathbb{R}^n$ and $x \neq 0$

**Example 3.10**

Examples of SPD matrices include:

- Discrete Laplacian is SPD
- Normal equation of Least Squares
- Hessian of strictly convex functions

For general matrices, the LU decomposition is $\frac{2}{3}n^3 + O(n^2)$. For SPD matrices, we can reduce this to $\frac{1}{3}n^3 + O(n^2)$, meaning we can reduce the computation cost by half. In addition, no pivoting is necessary for SPD matrices.

There is a slight modification of LU for SPD matrices called the **Cholesky Decomposition**.

## 3.3 Cholesky Decomposition

Assume $A \in \mathbb{R}^{n\times n}$ is SPD. Then there exists a decomposition such that:

$$A = LL^T$$

where $L$ is a lower triangular matrix. With Cholesky decomposition, we can solve the linear system:

$$Ax = b \iff LL^T x = b \iff \begin{cases} Ly = b \\ L^T x = y \end{cases}$$

Since $A = LL^T$ with:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & & a_{nn} \end{bmatrix} \quad L = \begin{bmatrix} \ell_{11} & & & \\ \ell_{21} & \ell_{22} & & \\ \vdots & \vdots & \ddots & \\ \ell_{n1} & \ell_{n2} & & \ell_{nn} \end{bmatrix} \quad L^T = \begin{bmatrix} \ell_{11} & \ell_{21} & \dots & \ell_{n_1} \\ & \ell_{22} & \dots & \ell_{n_2} \\ & & \ddots & \vdots \\ & & & \ell_{nn} \end{bmatrix}$$

We have:

- $a_{11} = \ell_{11}^2 \implies \ell_{11} = \sqrt{a_{11}}$
- $a_{21} = \ell_{21}\ell_{11} \implies \ell_{21} = a_{21}/\ell_{11}$
- $a_{n1} = \ell_{n1}\ell_{11} \implies \ell_{n1} = a_{n1}/\ell_{11}$

To be continued in the next lecture.

# 4 February 11th, 2021

## 4.1 LU Decomposition Continued

Recall that:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & & \vdots \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \quad L = L = \begin{bmatrix} \ell_{11} & & & \\ \ell_{21} & \ell_{22} & & \\ \vdots & \vdots & \ddots & \\ \ell_{n1} & \ell_{n2} & & \ell_{nn} \end{bmatrix} \quad L^T = \begin{bmatrix} \ell_{11} & \ell_{21} & \dots & \ell_{n_1} \\ & \ell_{22} & \dots & \ell_{n_2} \\ & & \ddots & \vdots \\ & & & \ell_{nn} \end{bmatrix}$$

Observing the first column of $LL^T$, we have:

- $a_{11} = \ell_{11}^2 \implies \ell_{11} = \sqrt{a_{11}}$
- $a_{21} = \ell_{21}\ell_{11} \implies \ell_{21} = a_{21}/\ell_{11}$
- $a_{n1} = \ell_{n1}\ell_{11} \implies \ell_{n1} = a_{n1}/\ell_{11}$

**Remark 4.1 —** Note that we only need to compare the lower triangular part of $LL^T$, since it is symmetric.

For the second column, note that we have:

$$a_{k2} = \ell_{k1}\ell_{21} + \ell_{k2}\ell_{22}$$

As such, we have:

- $a_{22} = \ell_{21}^2 + \ell_{22}^2 \implies \ell_{22} = (a_{22} - \ell_{21}^2)^{\frac{1}{2}}$
- $\ell_{k2} = (a_{k2} - \ell_{k1}\ell_{21})\ell_{22}$ for $k = 3, \dots, n$.

We can continue this process, and for the $k$-th column, we have:

$$a_{kk} = \ell_{k1}^2 + \ell_{k2}^2 + \dots + \ell_{kk}^2 \implies \ell_{kk} = (a_{kk} - \sum_{i=1}^{k-1} \ell_{ki}^2)^{\frac{1}{2}}$$

and:

$$a_{ik} = \sum_{j=1}^{k} \ell_{ij}\ell_{kj} \implies \ell_{kj} = a_{ik} - (\sum_{j=1}^{k-1} \ell_{ij}\ell_{kj})/\ell_{kk}$$

for $i = k+1, \dots, n$. Thus, we have Algorithm 5.

However, Algorithm 5 is not memory efficient. To create a memory efficient version, we should use the lower triangular part of $L$ to overwrite that of $A$. This gives us Algorithm 6.

**Remark 4.2 —** The only difference between Algorithm 5 and 6 is that we change $\ell$ to $a$.

Using BLAS, we have Algorithm 7.

**Algorithm 5** Cholesky Decomposition Version 1

**for** $k = 1 : n$ **do**
    $\ell_{kk} = (a_{kk} - \sum_{i=1}^{k-1} \ell_{ki}^2)^{1/2}$
    **for** $i = k + 1 : n$ **do**
        $\ell_{ik} = (a_{ik} - \sum_{j=1}^{k-1} \ell_{ij}\ell_{kj})/\ell_{kk}$
    **end for**
**end for**

**Algorithm 6** Cholesky Decomposition Version 2 - Memory Efficient

**for** $k = 1 : n$ **do**
    $a_{kk} = (a_{kk} - \sum_{i=1}^{k-1} a_{ki}^2)^{1/2}$
    **for** $i = k + 1 : n$ **do**
        $a_{ik} = (a_{ik} - \sum_{j=1}^{k-1} a_{ij}a_{kj})/a_{kk}$
    **end for**
**end for**

**Algorithm 7** Cholesky Decomposition Version 3 - BLAS

**for** $k = 1 : n$ **do**
    $A(k, k) = A(k, k) - \text{norm}(A(k, 1 : k - 1), 2)^2$
    $A(k, k) = (A(k, k))^{1/2}$
    $A(k + 1 : n, k) = A(k + 1 : n, k) - A(k + 1 : n, 1 : k - 1)(A(k, 1 : k - 1))^T$
    $A(k + 1 : n, k) = A(k + 1 : n, k)/A(k, k)$
**end for**

**Theorem 4.3**

Cholesky decomposition has a complexity of:

$$\frac{1}{3}n^3 + O(n^2)$$

which is only half of LU for general $A$. In addition, the memory required is also halved.

**Example 4.4**

Consider $A = \begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix}$. We have:

- Step 1: $\begin{bmatrix} \sqrt{2} & & \\ \frac{1}{\sqrt{2}} & 2 & \\ 0 & 1 & 2 \end{bmatrix}$
- Step 2: $\begin{bmatrix} \sqrt{2} & & \\ \frac{1}{\sqrt{2}} & \sqrt{\frac{3}{2}} & \\ 0 & \sqrt{\frac{2}{3}} & 2 \end{bmatrix}$
- Step 3: $\begin{bmatrix} \sqrt{2} & & \\ \frac{1}{\sqrt{2}} & \sqrt{\frac{3}{2}} & \\ 0 & 1 & \sqrt{\frac{4}{3}} \end{bmatrix}$

This gives us:

$$L = \begin{bmatrix} \sqrt{2} & & \\ \frac{1}{\sqrt{2}} & \sqrt{\frac{3}{2}} & \\ 0 & 1 & \sqrt{\frac{4}{3}} \end{bmatrix}$$

**Remark 4.5 —** The SPD-ness of $A$ guarantees that $a_{kk} - \sum_{i=1}^{k-1} \ell_{ki}^2$ is positive.

## 4.2 LU on Banded Matrices

**Definition 4.6.** A matrix is **banded** if there exists an integer $p > 0$ such that:

$$a_{ij} \neq 0 \implies |i - j| \leq p$$

**Example 4.7**

. For $p = 1$, the matrix is called a **tri-diagonal matrix**:

$$A = \begin{bmatrix} \times & \times & & & & \\ \times & \times & \ddots & & & \\ & \ddots & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & \times \\ & & & & \times & \times \end{bmatrix} \in \mathbb{R}^{n \times n}$$

If we apply LU decomposition to a tri-diagonal matrix, we have:

- Step 1:

$$A(2,1) = A(2,1)/A(1,1)$$
$$A(2,2) = A(2,2) - A(2,1)A(1,2)$$

- Step $k$:

$$A(k+1,k) = A(k+1,k)/A(kk)$$
$$A(k+1,k+1) = A(k+1,k+1) - A(k+1,k)A(k,k+1)$$

This leads to Algorithm 8

**Remark 4.8 —** Note that the resulting matrix is also tri-diagonal.

**Algorithm 8** LU Decomposition for Tri-Diagonal Matrix

```
for k = 1 : n − 1 do
    A(k + 1, k) = A(k + 1, k)/A(k, k)
    A(k + 1, k + 1) = A(k + 1, k + 1) − A(k + 1, k)A(k, k + 1)
end for
```

**Theorem 4.9**
The computation complexity for Algorithm 8 is $\sum_{k=1}^{n-1} 3 = 3(n-1) \sim O(n)$.

**Theorem 4.10**
In general, for a banded matrix with band width $p$, the computation cost is $O(np^2)$.

**Example 4.11**
The discrete Laplacian matrix in 1-D is a banded matrix with $p = 1$ (tri-diagonal).

**Example 4.12**
The discrete Laplacian in 2-D on a grid is:

$$A_1 \otimes I + I \otimes A_2 \in \mathbb{R}^{N \times N},$$

where $N = n^2$, $A_1, A_2 \in \mathbb{R}^{n \times n}$ are the 1-D discrete Laplacian matrix for each dimension, it is a banded matrix with $p = n$.

**Remark 4.13 —** If we applied GE to example 4.12, it would be $O(Np^2) = O(N^2)$, which is significantly lower than $O(N^3)$.

Our goal is to improve this cost to $O(N)$.

## 4.3 LU on Spare Matrix

**Definition 4.14.** A **sparse matrix** is a matrix with many zero entries.

**Remark 4.15 —** Banded matrices with small bandwith can be considered to be sparse.

Sparse matrices arise from many numerical solutions of PDE, as differential operators are local, making many of the entries zero. Sometimes, we can exploit the structure of sparse matrix to speed up LU decomposition. However, LU does not always work well.

**Example 4.16**

Consider $\begin{bmatrix} 5 & 4 & 3 & 2 \\ 4 & 4 & & \\ 3 & & 3 & \\ 2 & & & 2 \end{bmatrix}$. Note that this is sparse. However, when we divide the first row, we will modify the lower sub matrix, making the final result a dense matrix. Thus it would be the same as applying LU to a general dense matrix, as the sparse structure is destroyed.

Now, we should ask, can we keep as many zeros as possible in LU? The answer is yes, by re-ordering both the equations and the unknowns, i.e. permute the matrix.

**Example 4.17**

If we consider the matrix in 4.16, we could permute it so we get:

$$\begin{bmatrix} 2 & & & 2 \\ & 3 & & 3 \\ & & 4 & 4 \\ 2 & 3 & 4 & 5 \end{bmatrix}$$

If we perform LU on this matrix, the sparse structure is preserved.

Now we must ask, how should we re-order to allow LU to produce the minimum number of **fill-in's**? This is a very difficult topic to answer, and was an active research topic. To answer this, we use graph theory. This is beyond the scope of this course.

# Index