National Central University

Department of Mathematics

Numerical Linear Algebra

Numerical Method of Solving Linear system

Author: 111201528 王元亨

January 11, 2024

Contents

1 Introduction

1.1 Introduction of Numeral Calculations

Numerical computation involves utilizing mathematical algorithms and approximation methods, executed on computers, to solve practical mathematical problems. In this field, we often encounter continuous and complex mathematical models that are challenging to solve directly or lack analytical solutions. Hence, we rely on numerical computation techniques, particularly in addressing the following key issues:

1. **Root Finding:** We sometimes need to locate the roots (solutions) of equations. Methods such as the bisection method, Newton's method utilizing the concept of tangents, and the secant method employing two points are effective approaches for solving systems of nonlinear equations.

2. **Numerical Differentiation and Integration:** In practical applications, calculating the derivative or integral of a function is often necessary. Numerical differentiation uses finite difference methods to estimate derivatives, while numerical integration employs techniques to approximate integral values, especially when analytical integration is unattainable.

3. **Interpolation and Curve Fitting:** Interpolation methods are used to construct a continuous function passing through a set of known data points, while curve fitting aims to find a function that best fits given data points. Common techniques include the method of least squares and extremum-seeking methods.

4. **Numerical Differential Equations:** Many scientific and engineering problems can be modeled as differential equations. Numerical methods such as Euler's method and the Runge-Kutta method are employed to approximate solutions to differential equations.

5. **Eigenvalue Problems:** In certain applications, finding the eigenvalues and eigenvectors of a matrix is necessary, as seen in applications such as structural mechanics and quantum mechanics. Specific numerical methods are required for solving these problems.

6. **Linear System Solving:** Gaussian elimination is a direct method used to solve linear systems, transforming the coefficient matrix into upper triangular form through a series of row operations, followed by back-substitution. Iterative methods and gradient descent are alternatives for approximating solutions to linear systems.

In numerical computation, attention must also be paid to errors introduced during the calculation process, particularly rounding errors due to finite precision representation. These errors are inevitable, requiring careful handling to ensure the reliability of results. Overall, numerical computation plays a crucial role in addressing complex problems and simulating real-world scenarios.

1.2 Background (definition & proposition)

- **Tridiagonal matrices:** Tridiagonal matrix is a band matrix that has nonzero elements only on the main diagonal, the subdiagonal/lower diagonal (the first diagonal

below this), and the supradiagonal/upper diagonal (the first diagonal above the main diagonal)

$$\begin{bmatrix} 2 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 2 \end{bmatrix} \text{ is a tridiagonal matrix.}$$

- **Symmetric positive definite (SPD):** A square matrix is called symmetric positive definite if it is symmetric and all its eigenvalues λ are positive, that is $\lambda > 0$

$$A = \begin{bmatrix} 7 & 1 & -1 & 2 \\ 1 & 8 & 0 & -2 \\ -1 & 0 & 4 & -1 \\ 2 & -2 & -1 & 6 \end{bmatrix} \text{ is symmetric positive definite.}$$

- **Diagonally dominant:** A matrix $A = (a_{ij})_{n\times n}$ is diagonally dominant if $|a_{ii}| \geq \sum_{j\neq i} |a_{ij}|$, for all $1 \leq i \leq n$.

$$A = \begin{bmatrix} 7 & 1 & -1 & 2 \\ 1 & 8 & 0 & -2 \\ -1 & 0 & 4 & -1 \\ 2 & -2 & -1 & 6 \end{bmatrix} \text{ is diagonally dominant.}$$

- **Pivoting:** Pivoting is a technique that involves swapping rows or columns of a matrix to avoid dividing by a small or zero pivot element.

- **Gershgorin theorem:** Given a matrix A, we suppose that $D(i) = \sum_{j\neq i} |a_{ij}|$. The eigenvalues λ of matrix A are located in its n Gershgorin disks. The center of each Gershgorin disk is the diagonal element a_{ii} of the matrix with the radius D(i).

- **Theorem on nonsingular matrix properties**

  For an $n \times n$ matrix A, the following properties are equivalent:

  1. The inverse of A exists; that is, A is nonsingular.
  2. The determinant of A is nonzero.
  3. The rows of A form a basis for $\mathbb{R}^n$.
  4. The columns of A form a basis for $\mathbb{R}^n$.
  5. As a map from $\mathbb{R}^n$ to $\mathbb{R}^n$, A is injective (one to one).
  6. As a map from $\mathbb{R}^n$ to $\mathbb{R}^n$, A is surjective (onto).
  7. The equation $Ax = 0$ implies $x = 0$.
  8. For each $b \in \mathbb{R}^n$, there is exactly one $x \in \mathbb{R}^n$ such that $Ax = b$.
  9. A is a product of elementary matrices.
  10. 0 is not an eigenvalue of A.

- **Condition number of a matrix** (ill-conditioned and well-conditioned)

  The condition number of a matrix A is

  $$\kappa(A) = \|A\|_2 \|A^{-1}\|_2 = \frac{\text{max eigenvalue of A}}{\text{min eigenvalue of A}}$$

  The larger the condition number, the more ill-conditioned the system.

- **norms:** We call **norm** on the space E any function $\|cdot\|$ maps form E into $\mathbb{R}_+$ and satisfying the following axioms.

  1. Homogeneity: $\forall (a, x) \in \mathbb{K} \times E, \|ax\| = |a| \cdot \|x\|$.
  2. Triangular inequality: $\forall x, y \in E, \|x + y\| \leq \|x\| + \|y\|$.
  3. Separation property: $\forall x \in E, \|x\| = 0$if and only if $x = 0$.

- **Spectral radius theorem:** The spectral radius function satisfies the equation:

  $$\rho(A) = \inf_{\|\cdot\|} \|A\|$$

  in which the infimum is taken over all subordinate matrix norms.

- **Alternative Theorem on Spectral Radius:**

  $$\rho(A) \leq \|A\|$$

  for any subordinate matrix norm.

- **Residual and error vector:** For linear system $Ax = b$ having the true solution x and a computed solution $\bar{x}$, we define

  $$e = \bar{x} - x \qquad \text{error vector} \tag{1}$$
  $$r = A\bar{x} - b \qquad \text{residual vector} \tag{2}$$

- Test matrix in MATLAB

  - **Hadamard matrix:** `H = hadamard(N)` is a Hadamard matrix of order N, that is, a matrix H with elements +1 or -1 such that `H'*H = N*EYE(N)`. n must be an integer and power of 2, $n = 2^k, k \in \mathbb{N}$
  - **Toeplitz matrix:** `toeplitz(C,R)` is a non-symmetric Toeplitz matrix having C as itsfirst column and R as its first row.
  - **Hankel matrix:** `hankel(C,R)` is a Hankel matrix whose first column is C and whose last row is R.
  - **Magic square:** `magic(N)` is an N-by-N matrix constructed from the integers1 through N^2 with equal row, column, and diagonal sums.Produces valid magic squares for all $N > 0$ except N = 2.
  - **Hilbert matrix:** `H = hilb(N)`is the N-by-N matrix with elements 1/(i+j-1), which is a famous example of a badly conditioned matrix. The `INVHILB` function calculates the exact inverse.

matrix	dim	non-singular	diagonally dominant	symmetric	SPD
`hilb`	5×5	no	no	yes	yes
`hadamard`	4×4	yes	no	no	no
`toeplitz`	5×5	yes	not always	yes	not always
`magic`	5×5	no	no	no	no
`hankel`	5×5	not always	not always	yes	not always

Table 1: properties of test matrix in MATLAB

2 Methods

2.1 Problem statement

A system of linear equations is a type of system of mathematical equations that conforms to the following form:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n & = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n & = b_2 \\ & \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n & = b_m \end{cases}$$

If expressed using concepts from linear algebra, the system of linear equations can be written as $\mathbf{Ax} = \mathbf{b}$.

$$A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

Our objective is to determine the solution vector $\mathbf{x}$ such that $\mathbf{Ax} = \mathbf{b}$ holds. However, the structure of matrix $\mathbf{A}$ is intricate, making the direct solution of $\mathbf{Ax} = \mathbf{b}$ a challenging task. Therefore, we need to explore alternative approaches based on the properties of $\mathbf{A}$. In this report, we make the assumption that $\mathbf{A}$ is non-singular, which facilitates a more convenient examination of methods for solving $\mathbf{Ax} = \mathbf{b}$.

In the upcoming section, we will delve into the techniques for solving $\mathbf{Ax} = \mathbf{b}$ through various methods, including direct methods, iterative methods, and the gradient descent method.

2.2 Direct methods

2.2.1 Naive Guassian Elimination

1. **Forward Elimination:** The basic forward elimination procedure using k to operate on equation k+1, k+2, . . . , n is

$$\begin{cases} a_{ij} \leftarrow a_{ij} - \dfrac{a_{ik}}{a_{kk}} a_{kj}, & (k \le j \le n, k < i < n) \\ b_i \leftarrow b_i - \dfrac{a_{ik}}{a_{kk}} b_k, \end{cases} \tag{3}$$

2. **Back Substitution:** We assume $a_{kk} \neq 0$. The back substitution procedure is

$$x_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j=i+1}^{n} a_{ij} x_j \right) \quad (i = n-1, n-2, \ldots, 1)$$

2.2.2 Matrix Factorization

1. **LU decomposition:** LU decomposition factors a matrix as the product of a lower triangular matrix and an upper triangular matrix.

$$A = LU = \begin{bmatrix} l_{11} & 0 & \cdots & 0 \\ l_{21} & l_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & l_{nn} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{bmatrix}$$

   Specifically, given a matrix A, the LU decomposition is expressed as A = LU, which L is lower triangular matrix and U is an upper triangular matrix. In fact, not every matrix has LU decomposition. If the matrix is singular, then they have no LU decomposition. This problem can be solved by pivoting or permuting the order of the columns of A, resulting in a PLU decomposition of A.

2. **Doolitle decomposition:** Doolittle Factorization is a special form of LU factorization, which the elements on the main diagonal of lower triangular matrix L are all 1, we call unit lower triangular matrix.

3. **Court decomposition:** Court Factorization is a special form of LU factorization, which the elements on the main diagonal of upper triangular matrix L are all 1, we call unit upper triangular matrix.

4. **LDL$^{\mathbf{T}}$ decomposition:** Let S be a positive-definite symmetric matrix. Then S has unique decompositions $S = LDL^T$ where L is unit lower-unitriangular, D is diagonal with positive diagonal entries.

5. **Cholesky decomposition:** The Cholesky decomposition of a Hermitian positive-definite matrix A, is a decomposition of the form $A = LL^*$, where L is a lower triangular matrix with real and positive diagonal entries, and L* denotes the conjugate transpose of L. Every Hermitian positive-definite matrix (and thus also every real-valued symmetric positive-definite matrix) has a unique Cholesky decomposition.

2.2.3 Some property of LDL$^{\mathbf{T}}$ decomposition

- If S be a positive-definite symmetric matrix, then S has unique decompositions $S = LDL^T$ and $S = L_1 L_1^T$, where L is unit lower-triangular, D is diagonal with positive diagonal entries, and L_1 is lower-triangular with positive diagonal entries ($L_1 = L\sqrt{D}$).
- The LU decomposition of symmetric positive definete matrices leads to the Cholesky decomposition.
- Every symmetric positive definite matrix has a Cholesky decomposition.
- A matrix has a Cholesky decomposition if and only if it is symmetric positive definite.

Since A is a symmetric positive definite matrix, then we factorize A into LDL^T, where L is unit lower-unitriangular, D is diagonal with positive diagonal entries. So the equation becomes $LDL^Tx = b$. Then the first step to solve the equation is using forward substitution to solve $LD(L^Tx) = b$, we then obtain the equation $L^Tx = z$. The second step is using back substitution to solve $L^Tx = z$ since L^T is a upper-uintriangular matrix. Hence we can obtian x.

2.2.4 Pseudocode of LU decomposition

Algorithm 1 LU Decomposition Pseudocode

procedure LU_DECOMPOSITION($n, (a_{ij})$)
  for $k \leftarrow 1$ to n **do**
    Specify a nonzero value for either l_{kk} or u_{kk} and compute the other from
    $l_{kk}u_{kk} = a_{kk} - \sum_{s=1}^{k-1} l_{ks}u_{sk}$
    for $j \leftarrow k+1$ to n **do**
      $u_{kj} \leftarrow \left(a_{kj} - \sum_{s=1}^{k-1} l_{ks}u_{sj}\right) \frac{1}{l_{kk}}$
    end for
    for $i \leftarrow k+1$ to n **do**
      $l_{ik} \leftarrow \left(a_{ik} - \sum_{s=1}^{k-1} l_{is}u_{sk}\right) \frac{1}{u_{kk}}$
    end for
  end for
  output $(l_{ij}), (u_{ij})$
end procedure

2.2.5 Pseudocode of LDL$^{\mathbf{T}}$ decomposition

Algorithm 2 LDL^T Decomposition

Given a SPD matrix $A = (a_{ij})_{1 \le i,j \le n}$
Initialize $D = (d_i)_{1 \le i \le n}$ as a vector of diagonal matrix and $L = (l_{ij})_{1 \le i,j \le n}$ as a lower triangular matrix.
$d_1 = a_{11}$
for $j = 1$ **to** n **do**
  $l_{jj} = 1$
  $d_j \leftarrow a_{jj} - \sum_{v=1}^{j-1} d_v l_{jv}^2$
  for $i = j+1$ **to** n **do**
    $l_{ji} = 0$
    $l_{ji} \leftarrow \left(a_{ij} - \sum_{v=1}^{j-1} l_{iv}d_v l_{jv}\right) \frac{1}{d_j}$
  end for
end for

If matrix A is a tri-diagonal SPD matrix, we can use simplified LDL^T decomposition to reduce the amount of calculation.

Algorithm 3 Pseudocode of simplified LDL^T decomposition

1: $d_1 = a_{11}$
2: **for** $k = 2$ to n **do**
3: $\quad l_{k-1} = \frac{a_{k,k-1}}{d_{k-1}}$
4: $\quad d_k = a_{k,k} - e_{k-1}a_{k,k-1}$
5: **end for**
6: $L = \text{eye}(n) + \text{diag}(l, -1)$
7: $D = \text{diag}(d)$

2.3 Iterative methods

2.3.1 What is iterative methods?

A general iterative algorithm for solving $\mathbf{Ax = b}$ goes as follows: Select a nonsingular matrix $\mathbf{Q}$, and having chosen an arbitrary starting vector $x^{(0)}$, generate vectors$x^{(1)}$, $x^{(2)}$, ... recursively from the equation

$$Qx^{(k)} = (Q - A)x^{(k-1)} + b \quad (k = 1, 2, 3, \dots)$$

To see that this is sensible, suppose that the sequence $x^{(k)}$ converge to a vector x^*, then by taking the limit as $k \to \infty$ in above equation, we get

$$Qx^* = (Q - A)x^* + b$$

This leads to $Ax^* = b$. Thus, if the sequence $x^{(k)}$ converges, its limit is a solution to the original $\mathbf{Ax = b}$.

If Q is invertible, then $x^{(k)} = (I - Q^{-1}A)x^{(k-1)} + Q^{-1}b$, we call $I - Q^{-1}A$ iterative matrix.

We split the matrix A into the sum of a nonzero diagonal matrix D, a strictly lower triangular matrix C_L, and a strictly upper triangular matrix C_U such that

$$A = D - C_L - C_U$$

Here, $D = diag(A)$, $C_L = (-a_{ij})_{i>j}$, $C_U = (-a_{ij})_{i<j}$

- Richardson iteration: Use $Q = I$.
- Jacobi method: Use $Q = D$.
- Gauss-Seidel method: Use $Q = D - C_L$
- SOR (successive overrelaxation) method: Use $Q = \frac{1}{\omega}D - C_L$, $\omega \in \mathbb{R}$ is called relaxtion factor.

To solve the system $Ax = b$, given the iterative matrix is $G = I - Q^{-1}A$, and $c = Q^{-1}b$. Let ϵ be tolerant error.

1. Given initial vector $x^{(0)}$
2. Iterative step: If $\|x^{(k)} - c\| > \epsilon$, then $x^{(k+1)} = Gx^{(k)} + c$.
   Else, stop.

- Jacobi method:

$$x_i^{(k)} = \frac{1}{a_{ii}} \left[-\sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^{(k-1)} + b_i \right], \forall 1 \leq i \leq n$$

- Gauss-Seidel method:

$$x_i^{(k)} = \frac{1}{a_{ii}} \left[-\sum_{\substack{j=1 \\ j<i}}^{n} a_{ij} x_j^{(k)} - \sum_{\substack{j=1 \\ j>i}}^{n} a_{ij} x_j^{(k-1)} + b_i \right], \forall 1 \leq i \leq n$$

- SOR (successive overrelaxation) method:

$$x_i^{(k)} = \frac{\omega}{a_{ii}} \left[-\sum_{\substack{j=1 \\ j<i}}^{n} a_{ij} x_j^{(k)} - \sum_{\substack{j=1 \\ j>i}}^{n} a_{ij} x_j^{(k-1)} + b_i \right] + (1-\omega) x_i^{(k-1)}, \forall 1 \leq i \leq n$$

2.3.2 Convergent condition

1. **Theorem on iterative method convergence**

   If $\|G\| < 1$ for some subordinate matrix norm, then the sequence produced by $x^{(k)} = Gx^{(k-1)} + c$ converges to the solution of Ax = b for any initial vector $x^{(0)}$.

2. We have $\forall G \in \mathcal{M}_n(\mathbb{C}), |\lambda| \leq \|G\|_M, \forall \lambda \in \sigma(G)$. Thus, we can use spectral radius $\rho(G) = \max_{\lambda \in \sigma(G)} |\lambda|$ to determine whether the sequence converge or not.

3. Since $\mathcal{M}_n(\mathbb{C})$ is finite dimensional norm space, all norm in $\mathcal{M}_n(\mathbb{C})$ are equivalent. Hence, if one kind of norm of $\mathcal{M}_n(\mathbb{C})$ is strictly less than 1, then the sequence produced by $x^{(k)} = Gx^{(k-1)} + c$ converges to the solution of Ax = b for any initial vector $x^{(0)}$.

4. If A is diagonally dominant, then the sequence produced by the Jacobi iteration and Gauss-Seidel method converges to the solution of Ax = b for any starting vector.

5. In the SOR method, suppose that the splitting matrix Q is chosen to be $\alpha D - C$, where α is a real parameter, D is any positive definite Hermitian matrix, and C is any matrix satisfying $C + C^* = D - A$. If A is positive definite Hermitian, if Q is nonsingular, and if $\alpha < \frac{1}{2}$, then the SOR iteration converges for any starting vector. The parameter α is usually denoted by $\frac{1}{\omega}$ The condition becomes $0 < \omega < 2$.

2.4 Gradient Descend Method

2.4.1 What is gradient descent method (steepest descent method)?

Through the negative gradient as the steepest slope, gradually find the (local) minimum (optimal solution)

Algorithm 4 Optimization Algorithm with For Loop

1: Given an initial guess x_0, maximum iteration maxIter, tolerence ϵ.
2: **for** $k = 0$ **to** maxIter **do**
3: Determine a search direction, $d_k = -\frac{\nabla f(x_k)}{\|\nabla f(x_k)\|}$.
4: Find the adaptive step size $\alpha_k = \frac{<d_k, d_k>}{<Ad_k, d_k>}$.
5: $x_{k+1} \leftarrow x_k + \alpha_k d_k$.
6: **if** $\|\nabla f(x_k)\| < \epsilon$ **then** stop.
7: **end if**
8: **end for**

2.4.2 Pseudocode of gradient descent method

2.4.3 Why is called "steepest descent"?

That is to say why $-\nabla f(x_k)$ is called steepest descent? Consider function $f : \mathbb{R}^n \longrightarrow \mathbb{R}$, the directional derivative at point $x \in \mathbb{R}^n$ in direction d is

$$D_d f(x) =< \nabla f(x), \frac{d}{\|d\|} >$$

By Cauchy-Schwarz inequality, we have

$$< \nabla f(x), \frac{d}{\|d\|} >\leq \|\nabla f(x)\| \|\frac{d}{\|d\|}\| = \|\nabla f(x)\|$$

The equality occurs when $d = \nabla f(x)$, then

$$< \nabla f(x), \frac{\nabla f(x)}{\|\nabla f(x)\|} >= \|\nabla f(x)\|$$

2.4.4 How to determine the step size α_k to ensure the convergence?

If the fixed step size α is too large, the algorithm will overshoot. That is to say, assuming $\alpha = 100$, I may only need 1 step to reach the best solution today, but the algorithm is forced to take a step size of 100 every time, so it can only oscillate near the best solution forever and can't find the best solution. If the fixed step size α is too small, although the oscillation problem is solved to some extent, the convergence speed will become extremely slow.

1. **Backtracking line search**

   Given $\alpha_0 = 1$, fix a parameter $0 < \beta < 1$. By Armijo (sufficient decrease) condition, then while

   $$f(x_{k+1}) = f(x_k - \alpha_k d_k) > f(x_k) - \frac{\alpha_k}{2}\|d_k\|\|\nabla f(x_k)\|$$

   update $t = \beta t$.

2. **Exact line search**

   At each iteration, do the best e can along the direction of the gradient such that $\alpha_k = \min f(x_k + \alpha_k d_k)$. But in most cases, this x cannot be found, so we will be more inclined to use Backtracking line search.

In our report, we suppose that $f(x) = \frac{1}{2} < Ax, x > - < b, x >$, the gradient of f is $\nabla f(x) = b - Ax$, and $\alpha_k = \frac{<d_k, d_k>}{<Ad_k, d_k>}$.

3 Results

3.1 Test matrix

1. To find $x \in \mathbb{R}^{999}$ such that $Ax = b$ for $A \in \mathbb{R}^{999\times 999}$ and $b \in \mathbb{R}^{999}$ given by

$$A_1 = \begin{bmatrix} 2 & -1 & 0 & \dots & 0 & 0 \\ -1 & 2 & -1 & \dots & 0 & 0 \\ 0 & -1 & 2 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 2 & -1 \\ 0 & 0 & 0 & \dots & -1 & 2 \end{bmatrix}_{999\times 999}, \quad b_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \\ 0 \end{bmatrix}_{1\times 999}$$

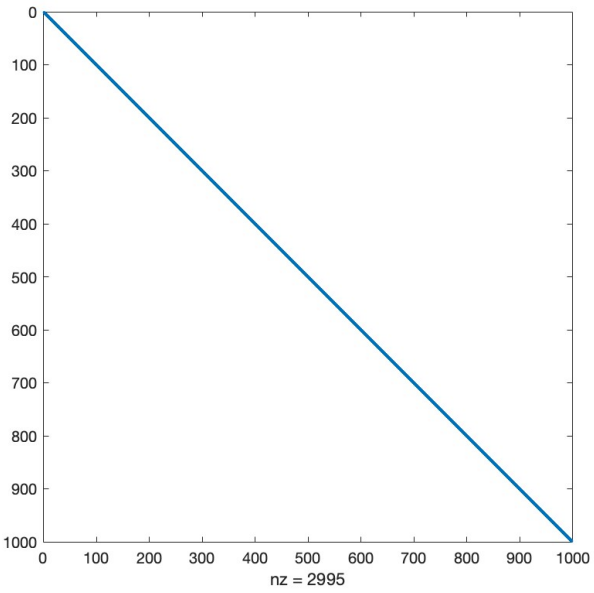


Figure 1: Structure of test matrix 1

Use MATLAB backslash, we get the solution of the system is

$$x_1 = \begin{bmatrix} 0.998 & 0.997 & 0.996 & \cdots & 0.002 & 0.001 \end{bmatrix}^T$$

$\kappa(A)$	$\|\cdot\|_1$	$\|\cdot\|_2$	$\|\cdot\|_\infty$	$\rho(A)$	$\min_{\lambda\in\sigma(G)} \lvert\lambda\rvert$	SPD
4.0528×10^5	4	4	4	4	9.8696e-06	yes

Table 2: some properties about A_1

2. **Hadamard matrix:** `H = hadamard(N)` is a Hadamard matrix of order N, that is, a matrix H with elements +1 or -1 such that `H'*H = N*EYE(N)`. n must be an integer and power of 2, $n = 2^k, k \in \mathbb{N}$. We choose $n = 512 = 2^9$.

$$A_2 = \begin{bmatrix} +1 & +1 & +1 & \cdots & +1 \\ +1 & -1 & +1 & \cdots & -1 \\ +1 & +1 & -1 & \cdots & -1 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ +1 & -1 & -1 & \cdots & +1 \end{bmatrix}_{512\times512}, \quad b_2 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}_{1\times512}$$

Use MATLAB backslash, we get the solution of the system is

$$x_2 = \begin{bmatrix} 0.002 & 0.002 & 0.002 & \cdots & 0.002 & 0.002 \end{bmatrix}^T$$

$\kappa(A)$	$\|\cdot\|_1$	$\|\cdot\|_2$	$\|\cdot\|_\infty$	$\rho(A)$	$\min_{\lambda\in\sigma(G)} \lvert\lambda\rvert$	SPD
1	512	22.6274	512	22.6274	-22.6274	no

Table 3: some properties about A_2

3.2 Comparison of general LU and simplified LDL$^{\mathbf{T}}$ in terms of computing time for test matrix 1

Since test matrix 1 is a tri-diagonal SPD matrix, we can use simplified LDL^T decomposition and Guassian elimination to solve this system of equation.

Comparison Result and Discussion

- Time of LU solver: 0.050699
- Time of LDL^T solver: 0.073782

Since the LU solution is built-in within MATLAB, it is likely optimized for software operations, resulting in faster calculation times. On the other hand, the LDL^T solution, implemented by ourselves, exhibits a slower computation time compared to the LU solution.

From a time complexity perspective, the total calculation amount of general LU decomposition is

$$\begin{aligned} &2((n-1)^2 + (n-2)^2 + (n-3)^2) + \cdots + 1^2 \\ &= 2 \times \frac{n(n-1)(2n-1)}{6} \\ &= \frac{2}{3}n^3 - n^2 + 3n \end{aligned}$$

The time complexity general LU decomposition is $\mathbf{O}(\frac{2}{3}n^3)$. And the total calculation amount

of LDL^T decomposition is

$$\begin{aligned}
&(n^2 - n) + ((n-1)^2 - (n-1)) + ((n-2)^2 - (n-2)) + \cdots + (1^2 - 1) \\
&= (n^2 + (n-1)^2 + \cdots + 1^2) - (n + (n-1) + \cdots + 1) \\
&= \frac{n(n+1)(2n+1)}{6} - \frac{n(n+1)}{2} \\
&= \frac{1}{3}n^3 - \frac{1}{3}n
\end{aligned}$$

The time complexity general LU decomposition is $\mathbf{O}(\frac{1}{3}n^3)$. So we get that the complexity of LDL^T decomposition is half that of LU decomposition. The time complexity of the simplified LDL^T solution is only $\mathbf{O}(n)$. Using simplified LDL^T solution can effectively reduce the amount of calculation.

3.3 Comparison of Jacobi, GS, SOR, gradient descent and CG (MATLAB) in terms of computing time for test problem 1 and 2.

Comparison Result and Discussion

1. **Test problem 1**

	Jacobi	Gauss-Seidel	SOR	gradient descent	CG
residual	9.9996e-05	9.9964e-05		9.9996e-05	4.1e-14
iterative times	9677	2386		873	999

Table 4: solution of $A_1 x_1 = b_1$

The result of preconditioned conjugate gradients method `pcg`

```
pcg converged at iteration 999 to a solution with relative residual 4.1e-14.
```

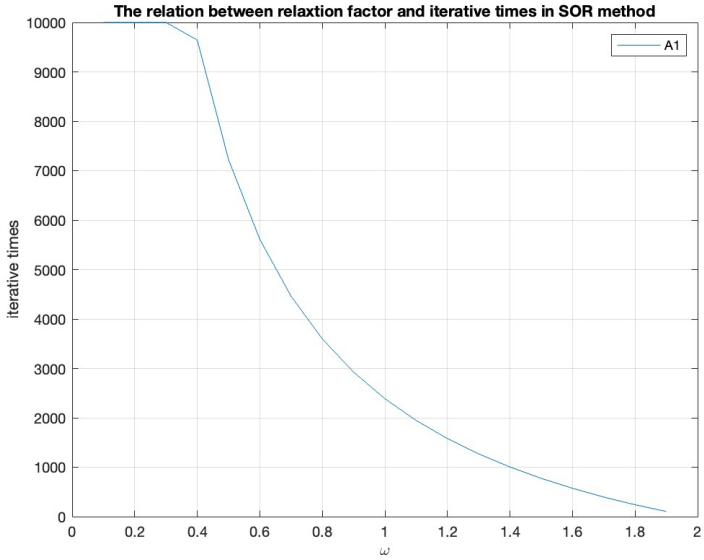
ω	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1.0
iteration times of A_1	10000	10000	10000	9644	7224	5611	4459	3595	2923	2386
ω	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	
iteration times of A_1	1946	1579	1269	1004	774	573	369	240	105	

Table 5: The relation between relaxtion factor and iterative times in SOR method

The provided matrix exhibits convergence for Jacobi, Gauss-Seidel, and SOR methods, as the norm of its iteration matrices is consistently less than 1. Among the three iterative techniques, Jacobi requires the highest number of iterations, followed by Gauss-Seidel. In the case of the SOR method, a larger relaxation factor leads to fewer iterations and faster convergence. It's worth noting that using values of $\omega > 1$ in SOR accelerates the convergence of a slow-converging process, while values of $\omega < 1$ are often employed to establish convergence in a diverging iterative process or expedite the convergence of an overshooting process.

2. **Test problem 2**

   The result of preconditioned conjugate gradients method `pcg`



	Jacobi	Gauss-Seidel	SOR	gradient descent	CG
tolerant error	0.0001	0.0001	0.0001		
iterative times	10000	10000	10000	10000	2

Table 6: solution of $A_2x_2 = b_2$

```
pcg converged at iteration 999 to a solution with relative residual 4.1e-14.
x = [0 0 0 ... 0]
```

For this matrix, because the norm of its iteration matrix is greater than one, the above iteration methods cannot converge. The function $f(x) = \frac{1}{2} < Ax, x > - < b, x >$ has no minimum value, so the gradient descent method must not converge.

4 Conclusions

The advantages and disadvantages of each method are comprehensively compared below.

- LU decomposition: Compared to Gaussian elimination, LU decomposition involves relatively less computation and can effectively address problems such as solving linear systems, matrix inversion, and eigenvalue computation. However, not all matrices are suitable for LU decomposition; it can only be applied to nonsingular matrices.
- LDLT decomposition: Similar to LU decomposition, LDLT decomposition offers lower computational complexity. However, it imposes stricter constraints on matrix properties, requiring the matrix to be symmetric positive definite (SPD) for this method to be applicable.
- Iterative methods: In contrast to direct methods, iterative methods have a broader range of applications and higher computational efficiency. However, they do not guarantee convergence for all initial values and matrices, and the precision of the solution is limited.

- Gradient Descent: Characterized by higher computational efficiency and widespread applicability, gradient descent requires accurate selection of the iteration step size for convergence. Convergence is contingent on precise choices in the iteration process.

These numerical techniques play crucial roles in solving various mathematical problems, each with its own advantages and considerations. The choice of method depends on the specific properties of the problem and the desired balance between computational efficiency and solution accuracy.

There are numerous numerical methods for solving linear systems, each with distinct limitations and advantages. Adequate analysis and understanding of the specific problem are essential for choosing an appropriate method, ensuring more accurate solutions. It is crucial to tailor the method selection to the characteristics of the problem at hand in order to achieve precision in the solutions obtained.

5 Appendix

5.1 MATLAB Code of Test Problem Main funcion

```
% Initialize the matrices
n = 999;
A1 = diag(2*ones(1, n)) + diag((-1)*ones(1, n-1), 1) + diag((-1)*ones(1, n-1), -1);
b1 = eye(n, 1);

A2 = hadamard(512);
b2 = eye(512, 1);

spy(A);

% Determine that matrix A is symmetric positive definite
is_spd = issymmetric(A) && all(eig(A) > zeros(n, 1));
if is_spd
    disp('A is symmetric positive definite');
else
    disp('A is not symmetric positive definite');
end

% Solve Ax=b by LU factorization
tic;
tstart = tic;
[L, U, P] = lu(A);
z1 = L\P*b;
x1 = U\z1;
t = toc(tstart);
fprintf('time of LU solver: %f\n', t);


% Solve Ax=b by LDLT Decomposition
% and forward substitution for tridiagonal matrix
```

```
tic;
tstart = tic;
[L, D, LT] = LDLTTridiagDeco(A);
B = FSTridiagonal(L, b);
x3 = D*LT\B;
t = toc(tstart);
fprintf('time of LDLT solver: %f\n', t);

% determine whether the matrix is SPD and diagonally dominant or not
is_spd_A1 = isSPD(A1);
is_spd_A2 = isSPD(A2);
is_dd_A1 = isDiagonallyDominant(A1);
is_dd_A2 = isDiagonallyDominant(A2);

% separate the matrix into strictly upper triangular, diagonal, strictly
% lower triangular parts.
[DA1, UA1, LA1] = splitMatrix(A1);
[DA2, UA2, LA2] = splitMatrix(A2);

% find iterative matrix
G1_richarson = eye(size(A1)) - A1;
G1_jacobi = eye(size(A1)) - DA1\A1;
G1_gauss_seidel = eye(size(A1)) - (DA1 - LA1)\A1;

G2_richarson = eye(size(A2)) - A2;
G2_jacobi = eye(size(A2)) - DA2\A2;
G2_gauss_seidel = eye(size(A2)) - (DA2 - LA2)\A2;

% compute the one, two, and infinity norm of iterative matrix
[G1_richarson_norm1, G1_richarson_norm2, G1_richarson_norminf]
    = calculateMatrixNorms(G1_richarson);
[G1_jacobi_norm1, G1_jacobi_norm2, G1_jacobi_norminf]
    = calculateMatrixNorms(G1_jacobi);
[G1_gauss_seidel_norm1, G1_gauss_seidel_norm2, G1_gauss_seidel_norminf]
    = calculateMatrixNorms(G1_gauss_seidel);

[G2_richarson_norm1, G2_richarson_norm2, G2_richarson_norminf]
    = calculateMatrixNorms(G2_richarson);
[G2_jacobi_norm1, G2_jacobi_norm2, G2_jacobi_norminf]
    = calculateMatrixNorms(G2_jacobi);
[G2_gauss_seidel_norm1, G2_gauss_seidel_norm2, G2_gauss_seidel_norminf]
    = calculateMatrixNorms(G2_gauss_seidel);

% compute spectral radius of iterative matrix
SR_G1_richarson = max(abs(eig(G1_richarson)));
SR_G1_jacobi = max(abs(eig(G1_jacobi)));
SR_G1_gauss_seidel = max(abs(eig(G1_gauss_seidel)));
```

```
SR_G2_richarson = max(abs(eig(G2_richarson)));
SR_G2_jacobi = max(abs(eig(G2_jacobi)));
SR_G2_gauss_seidel = max(abs(eig(G2_gauss_seidel)));

omega = 0.1:0.1:1.9;

% Test different method solving Ax=b
disp("A1*x1 = b1");
x1 = A1\b1;
disp(x1);
[x1_j, iter1_j] = jacobiMethod(A1, b1, 0.0001, 10000);
disp(x1_j);
[x1_g, iter1_g] = gaussSeidelMethod(A1, b1, 0.0001, 10000);
disp(x1_g);

x_init1 = 5*b1;
[x_optimal1, f_optimal1, history_x1, history_y1, iterations1]
= gradientDescent(A1, b1, x_init1, 1, 1e-6, 1000);
disp(iterations1);

iter1 = zeros(1, 19);
for i = 1:19
    [x1_s, iter1_s] = sorMethod(A1, b1, omega(i), 0.0001, 10000);
    iter1(i) = iter1_s;
end
plot(omega, iter1);
hold on;

disp("A2*x2 = b2");
x2 = A2\b2;
disp(x2);
[x2_j, iter2_j] = jacobiMethod(A2, b2, 0.0001, 10000);
disp(x2_j);
[x2_g, iter2_g] = gaussSeidelMethod(A2, b2, 0.0001, 10000);
disp(x2_g);

x_init2 = 5*b2;
[x_optimal2, f_optimal2, history_x2, history_y2, iterations2]
= gradientDescent(A2, b2, x_init2, 1, 1e-6, 10000);
disp(iterations2);

iter2 = zeros(1, 19);
for i = 1:19
    [x2_s, iter2_s] = sorMethod(A2, b2, omega(i), 0.0001, 10000);
    iter2(i) = iter2_s;
end

plot(omega, iter2);
```

```
hold on;
grid on;
legend('A1', 'A2');
xlabel('${\omega}$', 'Interpreter', 'latex');
ylabel('iterative times');
title('The relation between relaxtion factor and iterative times in SOR method');
```

5.2 MATLAB Code of Simplified LDL^T Decomposition Function

```
function [L, D, LT] = LDLTTridiagDeco(A)
    % Determine whether A is a square matrix or not
    [m, n] = size(A);
    if m ~= n
        error('A is not a square matrix.');
    end

    % Determine whather A is tridiagonal matrix or not
    is_tridiagonal = all(all(tril(A, -2) == 0)) && all(all(triu(A, 2) == 0));

    if ~is_tridiagonal
        error('A is not a tridiagonal matrix');
    end

    % Determine that matrix A is symmetric positive definite
    is_spd = issymmetric(A) && all(eig(A) > zeros(n, 1));
    if ~is_spd
        error('A is symmetric positive definite');
    end

    % Initialize matrix D and vector e
    e = zeros(n-1, 1);
    D = zeros(n, n);
    D(1, 1) = A(1, 1);

    % LDLT Decomposition
    for k = 2:n
        e(k-1) = A(k, k-1)/D(k-1, k-1);
        D(k, k) = A(k, k) - e(k-1)*A(k, k-1);
    end

    % calculate L and LT
    L = eye(n) + diag(e, -1);
    LT = L';

end
```

5.3 MATLAB Code of Forward Substitution Method for Tri-diagonal Matrix

```
function B = FSTridiagonal(A, b)
    [m, n] = size(A);
    [p, q] = size(b);
    if n ~= p
        error('Error');
    end

    % Determine whather A is tridiagonal matrix or not
    is_tridiagonal = all(all(tril(A, -2) == 0)) && all(all(triu(A, 2) == 0));

    if ~is_tridiagonal
        error('A is not a tridiagonal matrix');
    end

    % Initialize vector B
    B = zeros(n, 1);
    B(1) = b(1)/A(1, 1);
    % Forward substitution
    for i = 2:n
        w = 0;
        for j = 1:i-1
            w = A(i, j)*B(j);
        end
        B(i) = (b(i) - w)/A(i, i);
    end
end
```

5.4 MATLAB Code of Jacobi method

```
function [x, iter] = jacobiMethod(A, b, tol, maxIter)
    % Solve Ax=b by Jacobi iterative method.
    % tol: tolerant error
    % maxIter: the maximum number of iterations
    % x: the solution of Ax=b
    % iter: total times of iterations


    % Determine whether A is a square matrix or not
    [m, n] = size(A);
    if m ~= n
        error('The input matrix A have to be a square matrix.');
    end

    % Initialization
    x = zeros(n, 1);
```

```
    iter = 0;

    % Iterative step
    while iter < maxIter
        x_old = x;
        for i = 1:n
            sigma = A(i, :) * x_old - A(i, i) * x_old(i);
            x(i) = (b(i) - sigma) / A(i, i);
        end

        % Check for convergence
        if norm(b - A*x, inf) < tol
            break;
        end

        iter = iter + 1;
    end

    % Display a warning if the specified convergence criterion is not reached
    if iter == maxIter
        warning('The convergence criterion was not reached.');
    end
end
```

5.5 MATLAB Code of Gauss-Seidel method

```
function [x, iter] = gaussSeidelMethod(A, b, tol, maxIter)
    % Solve Ax=b by Gauss Seidel iterative method.
    % tol: tolerant error
    % maxIter: the maximum number of iterations
    % x: the solution of Ax=b
    % iter: total times of iterations

    % Determine whether A is a square matrix or not
    [m, n] = size(A);
    if m ~= n
        error('The input matrix A have to be a square matrix.');
    end

    % Initialization
    x = zeros(n, 1);
    iter = 0;

    % Iterative step
    while iter < maxIter
        x_old = x;
        for i = 1:n
            sigma1 = A(i, 1:i-1) * x(1:i-1);
```

```
            sigma2 = A(i, i+1:end) * x_old(i+1:end);
            x(i) = (b(i) - sigma1 - sigma2) / A(i, i);
        end

        % Check for convergence
        if norm(b - A*x, inf) < tol
            break;
        end

        iter = iter + 1;
    end

    % Display a warning if the specified convergence criterion is not reached
    if iter == maxIter
        warning('The convergence criterion was not reached.');
    end
end
```

5.6 MATLAB Code of SOR method

```
function [x, iter] = sorMethod(A, b, omega, tol, maxIter)
    % Solve Ax=b by SOR iterative method.
    % omega: relaxtion factor
    % tol: tolerant error
    % maxIter: the maximum number of iterations
    % x: the solution of Ax=b
    % iter: total times of iterations

    % Determine whether A is a square matrix or not
    [m, n] = size(A);
    if m ~= n
        error('The input matrix A have to be a square matrix.');
    end

    % Initialization
    x = zeros(n, 1);
    iter = 0;

    % Iterative step
    while iter < maxIter
        x_old = x;
        for i = 1:n
            sigma1 = A(i, 1:i-1) * x(1:i-1);
            sigma2 = A(i, i+1:end) * x_old(i+1:end);
            x(i) = (1 - omega)*x_old(i) + (omega/A(i, i))*(b(i) - sigma1 - sigma2);
        end

        % Check for convergence
```

```
        if norm(b - A*x, inf) < tol
            break;
        end

        iter = iter + 1;
    end

    % Display a warning if the specified convergence criterion is not reached
    if iter == maxIter
        warning('The convergence criterion was not reached.');
    end
end
```

5.7 MATLAB Code of check norm of iterative matrix

```
function isSPD = isSPD(A)
    % check the matrix A is symmetric positive definite
    [m, n] = size(A);
    if m ~= n
        error('The input matrix have to be a square matrix.');
    end
    isSPD = issymmetric(A) && all(eig(A) > zeros(n, 1));
end

function isDiagonallyDominant = isDiagonallyDominant(matrix)
    % 確保輸入是方陣
    [m, n] = size(matrix);
    if m ~= n
        error('The input matrix have to be a square matrix.');
    end

    isDiagonallyDominant = true;

    for i = 1:n
        rowSum = sum(abs(matrix(i,:))) - abs(matrix(i,i));
        if abs(matrix(i,i)) <= rowSum
            isDiagonallyDominant = false;
            break;
        end
    end
end

function [diagonal, upperTriangle, lowerTriangle] = splitMatrix(A)
    % 確保輸入是方陣
    [m, n] = size(A);
    if m ~= n
        error('輸入矩陣必須是方陣。');
    end
```

```
    % 初始化對角、上三角和下三角矩陣
    diagonal = diag(diag(A));
    upperTriangle = -triu(A, 1);  % 上三角
    lowerTriangle = -tril(A, -1); % 下三角
end

function [norm_1, norm_2, norm_inf] = calculateMatrixNorms(A)
    % 1 norm
    norm_1 = norm(A, 1);

    % 2 norm
    norm_2 = norm(A, 2);

    % infinity norm
    norm_inf = norm(A, Inf);
end
```

5.8 MATLAB Code of Gradient Descend Method

```
function [x_optimal, f_optimal, history_x, history_y, iterations]
    = gradientDescent(A, b, x_initial, alpha, tolerance, maxIter)
    % Gradient Descent Method

    % Objective function
    f = @(x) 0.5 * x' * A * x - b' * x;

    % Initialize variables
    history_x = null(maxIter, 1);
    history_y = null(maxIter, 1);
    x_k = x_initial;
    iterations = 0;

    % Gradient descent iterations
    while true

        % Compute the gradient at the current point
        gradient = b - A * x_k;

        alpha = (gradient'*gradient)/(gradient'*A*gradient);

        % Update the solution
        x_k1 = x_k + alpha * gradient;

        % Check for convergence
        if norm(A*x_k1 - b, 2) < tolerance
            break;
        end
```

```
        % Update variables for the next iteration
        iterations = iterations + 1;
        history_x(iterations) = x_k(1);
        history_y(iterations) = x_k(2);
        x_k = x_k1;
        if iterations >= maxIter
            break;
        end
    end

    % Final optimal values
    x_optimal = x_k;
    f_optimal = f(x_optimal);
end
```

References

[1] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial*, SIAM, Philadelphia, 2000.

[2] X.-C. Cai and D. E. Keyes, *Nonlinearly preconditioned inexact Newton algorithms*, SIAM J. Sci. Comput., 24 (2002), pp. 183-200.

[3] J. Dennis and R. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, SIAM, Philadelphia, 1996.

[4] F.-N. Hwang and X.-C. Cai, *Improving robustness and parallel scalability of Newton method through nonlinear preconditioning*, Lecture Notes in Computational Science and Engineering, ed. R. Kornhuber, R. H. W. Hoppe, D. E. Keyes, J. Periaux, O. Pironneau and J. Xu, Springer-Verlag, Haidelberg, (2004), pp. 201-208.

[5] R. S. Tuminaro, H. F. Walker, and J. N. Shadid, *On backtracking failure in Newton-GMRES methods with a demonstration for the Navier-Stokes Equations*, J. Comput. Phys., 180 (2002), pp. 549-558.