CEE6501 — Lecture 2.2

Solving Linear Systems: $\mathbf{A}\mathbf{x} = \mathbf{b}$

Learning Objectives

By the end of this lecture, you will be able to:

- State why **explicit inversion** is usually a bad idea for solving $\mathbf{A}\mathbf{x} = \mathbf{b}$
- Explain and apply **direct elimination** methods (GE, Gauss–Jordan, LU, Cholesky, LDL^T, Thomas)
- Apply basic **iterative** solvers (Jacobi, Gauss–Seidel) and interpret convergence
- Connect matrix **structure** (symmetry, positive definiteness, bandedness) to solver choice

The Core Problem

In matrix structural analysis, everything reduces to solving:

$$\mathbf{K}\mathbf{u} = \mathbf{f}$$

- $\mathbf{K}$: (global) stiffness matrix
- $\mathbf{u}$: unknown displacements (DOFs)
- $\mathbf{f}$: applied nodal loads

Today: **how** we solve these efficiently and robustly.

Part 1 — Direct, Closed-Form Methods

Method 1: Direct Inversion

> *Mathematically valid to solve, via* $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$. *Practically discouraged.*

Why Direct Inversion is (Usually) a Bad Idea

Computing $\mathbf{A}^{-1}$ explicitly is typically:

- **slower** than factorization + substitution (later in lecture)
- **less accurate** (numerically)
- **unnecessary** when all you want is $\mathbf{x}$

In NumPy

We will use **NumPy** throughout the course.

> *Convention: solve $\mathbf{A}\mathbf{x} = \mathbf{b}$ using `np.linalg.solve(A, b)`, not `inv(A) @ b`.*

```
x = np.linalg.solve(A, b)   # preferred: factorization + substitution
# x = np.linalg.inv(A) @ b  # avoid: explicit inversion
```

Inversion vs. Solve (Same Math, Different Computation)

Both compute a solution, but one is usually preferred:

$$\mathbf{x} = \mathbf{A}^{-1}\mathbf{b} \quad \text{(explicit inverse)}$$

$$\mathbf{x} = \mathrm{solve}(\mathbf{A}, \mathbf{b}) \quad \text{(factorization + substitution)}$$

In [9]:
```python
import numpy as np
import time

def compare_solve_vs_inverse(A, b, repeat=1):
    # --- solve ---
    t0 = time.perf_counter()
    for _ in range(repeat):
        x_solve = np.linalg.solve(A, b)
    t_solve = (time.perf_counter() - t0) / repeat

    # --- inverse ---
    t0 = time.perf_counter()
    for _ in range(repeat):
        x_inv = np.linalg.inv(A) @ b
    t_inv = (time.perf_counter() - t0) / repeat

    return {
        # "x_solve": x_solve,
        # "x_inv": x_inv,
        "time_solve": t_solve,
        "time_inv": t_inv,
        "time_ratio_inv_over_solve": t_inv / t_solve
    }
```

```
In [10]: A_small = np.array([[4.0, 2.0, 0.0],
                             [2.0, 5.0, 1.0],
                             [0.0, 1.0, 3.0]])

         b_small = np.array([2.0, 4.0, 6.0])

         result_small = compare_solve_vs_inverse(A_small, b_small, repeat=1000)

         for k, v in result_small.items():
             print(f"{k}: {v}")
```

```
time_solve: 3.403374983463436e-06
time_inv: 4.283749993192032e-06
time_ratio_inv_over_solve: 1.2586770526334081
```

```
In [11]: np.random.seed(0)

         n = 100

         # Symmetric positive definite matrix
         M = np.random.randn(n, n)
         A_large = M.T @ M + n * np.eye(n)
         b_large = np.random.randn(n)

         result_large = compare_solve_vs_inverse(A_large, b_large, repeat=3)

         for k, v in result_large.items():
             print(f"{k}: {v}")
```

```
time_solve: 6.541666031504671e-05
time_inv: 0.0004647083405870944
time_ratio_inv_over_solve: 7.1038224566808905
```

Method 2: Cramer’s Rule

Conceptually Useful, Practically Limited

For a **small** system (especially 2×2 or 3×3), Cramer's rule is a clean closed form:

$$x_i = \frac{\det(\mathbf{A}_i)}{\det(\mathbf{A})}$$

where $\mathbf{A}_i$ is $\mathbf{A}$ with column i replaced by $\mathbf{b}$.

For large n:

- determinant costs scale poorly
- numerically fragile

> *Cramer's rule is a teaching tool, not a production solver.*

In [12]:
```
import numpy as np

A = np.array([[2.0, 1.0],
              [3.0, 4.0]])
b = np.array([5.0, 6.0])

detA = np.linalg.det(A)
A1 = A.copy(); A1[:, 0] = b
A2 = A.copy(); A2[:, 1] = b

x_cramer = np.array([np.linalg.det(A1)/detA, np.linalg.det(A2)/detA])
x_solve = np.linalg.solve(A, b)

print('det(A)=', detA)
print('x (Cramer)=', x_cramer)
print('x (solve)= ', x_solve)
print('residual ||Ax-b|| =', np.linalg.norm(A @ x_solve - b))
```

```
det(A)= 5.000000000000001
x (Cramer)= [ 2.8 -0.6]
x (solve)=  [ 2.8 -0.6]
residual ||Ax-b|| = 0.0
```

Part 2 — Direct Elimination Methods

> *The idea: transform $\mathbf{A}$ into a form that is easy to solve.*

Big Picture: Elimination

We use row operations to transform a linear system into a form that can be solved by substitution.

Two closely related tasks:

- Reduction of an augmented system $[\mathbf{A}|\mathbf{b}]$ (conceptual, one-off solves)
  - Method 1: Gaussian Elimination
  - Method 2: Gauss–Jordan Elimination
- Factorization of $\mathbf{A}$ (preferred for repeated solves)
  - Method 3: LU Decomposition
  - Method 4: Cholesky Factorization
  - Method 5: LDL^T Factorization
  - Method 6: Thomas Method (special cases)

Method 1: Gaussian Elimination (GE)

Goal

- transform the system

$$\mathbf{A}\mathbf{x} = \mathbf{b} \longrightarrow \mathbf{U}\mathbf{x} = \hat{\mathbf{b}}$$

- where $\mathbf{U}$ is upper triangular

Important notes

- GE is an elimination procedure, not a solver (IMPORTANT)
- it systematically applies row operations to expose matrix structure
- it is the foundation of:
    - LU factorization
    - pivoted direct solvers
    - most practical linear algebra methods

Gaussian Elimination — Procedure

1. Choose the current pivot on the diagonal (the active diagonal entry used to eliminate entries below it).
2. Eliminate all entries below the pivot using row operations.
3. Apply the same row operations to the right-hand side, updating $\mathbf{b}$.
4. Move to the next column and repeat until $\mathbf{A} \to \mathbf{U}$, $\mathbf{b} \to \hat{\mathbf{b}}$.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \xrightarrow{\text{Gaussian Elimination}} \mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} & u_{14} \\ 0 & u_{22} & u_{23} & u_{24} \\ 0 & 0 & u_{33} & u_{34} \\ 0 & 0 & 0 & u_{44} \end{bmatrix}$$

Solving step

- solve $\mathbf{U}\mathbf{x} = \hat{\mathbf{b}}$ by back-substitution

Method 2: Gauss–Jordan Elimination (GJ)

Goal

- reduce the augmented system

$$[\mathbf{A} \,|\, \mathbf{b}] \;\longrightarrow\; [\mathbf{I} \,|\, \mathbf{x}]$$

Important notes

- you operate directly on $\mathbf{b}$ (so $\mathbf{b}$ changes during elimination)
- for repeated solves with different $\mathbf{b}$, this is inefficient
- can also apply GJ to computes the inverse of a matrix (when it exists)

$$[\mathbf{A} \,|\, \mathbf{I}] \;\longrightarrow\; [\mathbf{I} \,|\, \mathbf{A}^{-1}]$$

Gauss–Jordan Elimination — Procedure

1. Form the augmented matrix $[\mathbf{A} \mid \mathbf{b}]$.
2. Choose a pivot and scale the pivot row so the pivot entry becomes 1.
3. Eliminate all entries below the pivot (like GE).
4. Eliminate all entries above the pivot (this is the “Jordan” part).
5. Repeat for the next column until $[\mathbf{A} \mid \mathbf{b}] \to [\mathbf{I} \mid \mathbf{x}]$.

$$\left[\ \mathbf{A} \ \middle|\ \mathbf{b}\ \right] = \left[\begin{array}{cccc|c} a_{11} & a_{12} & a_{13} & a_{14} & b_1 \\ a_{21} & a_{22} & a_{23} & a_{24} & b_2 \\ a_{31} & a_{32} & a_{33} & a_{34} & b_3 \\ a_{41} & a_{42} & a_{43} & a_{44} & b_4 \end{array}\right] \xrightarrow{\text{Gauss--Jordan Elimination}}$$

$$\left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & x_1 \\ 0 & 1 & 0 & 0 & x_2 \\ 0 & 0 & 1 & 0 & x_3 \\ 0 & 0 & 0 & 1 & x_4 \end{array}\right] = \left[\ \mathbf{I}\ \middle|\ \mathbf{x}\ \right]$$

Gauss–Jordan — Worked Example (start)

$$\begin{aligned} 5x_1 + 6x_2 - 3x_3 &= 66 \\ 9x_1 - x_2 + 2x_3 &= 8 \\ 8x_1 - 7x_2 + 4x_3 &= -39 \end{aligned}$$

$$\mathbf{A} = \begin{bmatrix} 5 & 6 & -3 \\ 9 & -1 & 2 \\ 8 & -7 & 4 \end{bmatrix}, \qquad \mathbf{b} = \begin{bmatrix} 66 \\ 8 \\ -39 \end{bmatrix}$$

Step 0: Form augmented matrix

$$[\mathbf{A} \mid \mathbf{b}] = \left[\begin{array}{ccc|c} 5 & 6 & -3 & 66 \\ 9 & -1 & 2 & 8 \\ 8 & -7 & 4 & -39 \end{array}\right]$$

Step 1: pivot in column 1, make the pivot 1

Row operation:

- $R_1 \leftarrow \frac{1}{5} R_1$

$$\left[\begin{array}{ccc|c} 1 & 1.2 & -0.6 & 13.2 \\ 9 & -1 & 2 & 8 \\ 8 & -7 & 4 & -39 \end{array}\right]$$

Step 2: eliminate column 1 below the pivot

Row operations:

- $R_2 \leftarrow R_2 - 9R_1$
- $R_3 \leftarrow R_3 - 8R_1$

$$\left[\begin{array}{ccc|c} 1 & 1.2 & -0.6 & 13.2 \\ 0 & -11.8 & 7.4 & -110.8 \\ 0 & -16.6 & 8.8 & -144.6 \end{array}\right]$$

Step 3: pivot in column 2, make the pivot 1

Row operation:

- $R_2 \leftarrow \frac{1}{-11.8} R_2$

$$\left[\begin{array}{ccc|c} 1 & 1.2 & -0.6 & 13.2 \\ 0 & 1 & -0.6271 & 9.39 \\ 0 & -16.6 & 8.8 & -144.6 \end{array}\right]$$

Step 4: Gauss–Jordan part, eliminate column 2 above and below the pivot

Row operations:

- $R_1 \leftarrow R_1 - 1.2R_2$
- $R_3 \leftarrow R_3 + 16.6R_2$

$$\left[\begin{array}{ccc|c} 1 & 0 & 0.1525 & 1.932 \\ 0 & 1 & -0.6271 & 9.39 \\ 0 & 0 & -1.61 & 11.27 \end{array}\right]$$

Step 5: pivot in column 3, make the pivot 1

Row operation:

- $R_3 \leftarrow \frac{1}{-1.61} R_3$

$$\left[\begin{array}{ccc|c} 1 & 0 & 0.1525 & 1.932 \\ 0 & 1 & -0.6271 & 9.39 \\ 0 & 0 & 1 & -7 \end{array}\right]$$

Step 6: eliminate column 3 above the pivot to reach $[\mathbf{I}|\mathbf{x}]$

Row operations:

- $R_1 \leftarrow R_1 - 0.1525 R_3$
- $R_2 \leftarrow R_2 + 0.6271 R_3$

$$\left[\begin{array}{ccc|c} 1 & 0 & 0 & 3 \\ 0 & 1 & 0 & 5 \\ 0 & 0 & 1 & -7 \end{array}\right] = \left[\begin{array}{c|c} \mathbf{I} & \mathbf{x} \end{array}\right]$$

So,

$$\mathbf{x} = \begin{bmatrix} 3 \\ 5 \\ -7 \end{bmatrix}$$

Gauss–Jordan Inverse (setup only)

Given

$$\mathbf{A} = \begin{bmatrix} 4 & 1 & 2 \\ 1 & 3 & 0 \\ 2 & 0 & 5 \end{bmatrix}$$

Form the augmented matrix $[\mathbf{A} \mid \mathbf{I}]$

$$[\mathbf{A} \mid \mathbf{I}] = \left[\begin{array}{ccc|ccc} 4 & 1 & 2 & 1 & 0 & 0 \\ 1 & 3 & 0 & 0 & 1 & 0 \\ 2 & 0 & 5 & 0 & 0 & 1 \end{array}\right]$$

Gauss–Jordan goal

$$[\mathbf{A} \mid \mathbf{I}] \longrightarrow \left[\mathbf{I} \mid \mathbf{A}^{-1}\right]$$

```
In [14]: A = np.array([[4.0, 1.0, 2.0],
                       [1.0, 3.0, 0.0],
                       [2.0, 0.0, 5.0]])

         invA_gj = gauss_jordan_inverse(A)
         invA_np = np.linalg.inv(A)

         print('inv(A) via Gauss–Jordan=\n', invA_gj)
         print('\ninv(A) via NumPy=\n', invA_np)
         print('\ncheck ||I - A inv(A)|| =', np.linalg.norm(np.eye(3) - A @ invA_
```

```
inv(A) via Gauss–Jordan=
 [[ 0.3488 -0.1163 -0.1395]
 [-0.1163  0.3721  0.0465]
 [-0.1395  0.0465  0.2558]]

inv(A) via NumPy=
 [[ 0.3488 -0.1163 -0.1395]
 [-0.1163  0.3721  0.0465]
 [-0.1395  0.0465  0.2558]]

check ||I - A inv(A)|| = 2.416688344895612e-16
```

Decomposition Theorem

> *Quick aside before we cover methods 3-6*

Elimination methods are based on the idea that a general square matrix can be factored into triangular (and sometimes diagonal) matrices.

The decomposition theorem states:

$$\mathbf{A} = \mathbf{LDU}$$

where

- $\mathbf{L}$ is unit lower triangular
- $\mathbf{D}$ is diagonal
- $\mathbf{U}$ is unit upper triangular

Example (3×3 case)

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & \ell_{32} & 1 \end{bmatrix} \begin{bmatrix} d_1 & 0 & 0 \\ 0 & d_2 & 0 \\ 0 & 0 & d_3 \end{bmatrix} \begin{bmatrix} 1 & u_{12} & u_{13} \\ 0 & 1 & u_{23} \\ 0 & 0 & 1 \end{bmatrix}$$

Variants of Matrix Decomposition

The specific elimination method depends on how the diagonal scaling is distributed among the triangular factors.

Starting from

$$\mathbf{A} = \mathbf{LDU},$$

the diagonal matrix $\mathbf{D}$ may be:

- associated with $\mathbf{L}$
- associated with $\mathbf{U}$
- split between both

As a result, a general square matrix may be factored as (generalization)

$$\mathbf{A} = \mathbf{LU}.$$

Important observations

- the matrices labeled $\mathbf{L}$ and $\mathbf{U}$ here are **not unique**
- they are **not necessarily the same** $\mathbf{L}$ and $\mathbf{U}$ appearing in $\mathbf{A} = \mathbf{LDU}$
- their exact form depends on how the diagonal scaling is distributed
- differences among elimination methods are largely differences in how this factorization is constructed

By regrouping factors, common variants are obtained:

$$\mathbf{A} = \mathbf{LU} \qquad \mathbf{A} = \mathbf{LDL}^T \qquad \mathbf{A} = \mathbf{LL}^T$$

Why is Matrix Decomposition Useful?

Once a matrix has been factorized, solving $\mathbf{Ax} = \mathbf{b}$ becomes a sequence of simple triangular solves.

Using

$$\mathbf{A} = \mathbf{LU},$$

we rewrite the system as

$$\mathbf{LUx} = \mathbf{b}.$$

Introduce an auxiliary vector $\hat{\mathbf{b}}$:

$$\mathbf{Ux} = \hat{\mathbf{b}}$$

Substitute into $\mathbf{LUx} = \mathbf{b}$:

$$\mathbf{L}\hat{\mathbf{b}} = \mathbf{b}$$

Solving: First Forward Substitution

Since $\mathbf{L}$ is a known lower triangular matrix and $\mathbf{b}$ is given, the auxiliary vector $\hat{\mathbf{b}}$ can be computed element by element using forward substitution.

$$\mathbf{L}\hat{\mathbf{b}} = \mathbf{b}.$$

$$\begin{bmatrix} \ell_{11} & 0 & 0 & \cdots & 0 \\ \ell_{21} & \ell_{22} & 0 & \cdots & 0 \\ \ell_{31} & \ell_{32} & \ell_{33} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & 0 \\ \ell_{n1} & \ell_{n2} & \ell_{n3} & \cdots & \ell_{nn} \end{bmatrix} \begin{bmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \hat{b}_3 \\ \vdots \\ \hat{b}_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_n \end{bmatrix}$$

Forward substitution

$$\hat{b}_i = \frac{b_i - \sum_{j=1}^{i-1} \ell_{ij}\,\hat{b}_j}{\ell_{ii}}, \qquad i = 1, \ldots, n$$

This allows $\hat{\mathbf{b}}$ to be computed sequentially from top to bottom.

Solving: Then Backward Substitution

Once $\hat{\mathbf{b}}$ is known, and $\mathbf{U}$ is a known upper triangular matrix, the solution vector $\mathbf{x}$ is obtained element by element using back substitution.

$$\mathbf{U}\mathbf{x} = \hat{\mathbf{b}}.$$

$$\begin{bmatrix} u_{11} & u_{12} & u_{13} & \cdots & u_{1n} \\ 0 & u_{22} & u_{23} & \cdots & u_{2n} \\ 0 & 0 & u_{33} & \cdots & u_{3n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & u_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} \hat{b}_1 \\ \hat{b}_2 \\ \hat{b}_3 \\ \vdots \\ \hat{b}_n \end{bmatrix}$$

Back substitution

$$x_i = \frac{\hat{b}_i - \sum_{j=i+1}^{n} u_{ij}\, x_j}{u_{ii}}, \qquad i = n, \ldots, 1$$

This allows $\mathbf{x}$ to be computed sequentially from bottom to top.

Why Factorization Helps for Repeated Solves

If the right-hand side changes ($\mathbf{b}$ changes), we do not need to refactor $\mathbf{A}$.

- $\mathbf{A}$ is factorized once
- each new $\mathbf{b}$ only requires:
    - forward substitution
    - back substitution

In contrast, Gauss–Jordan operates on the augmented system $[\mathbf{A} \mid \mathbf{b}]$, so a new $\mathbf{b}$ requires repeating the full elimination.

Example (operation counts from the code)

- $n = 30$, number of right-hand sides = 30
- LU total work (factor once + 30 solves): 74,215 ops
- Gauss–Jordan total work (redo elimination 30 times): 876,150 ops
- ratio: Gauss–Jordan / LU $\approx 11.81\times$

A Unified View

All of the following methods share the same structure:

- factorize the matrix once $\mathcal{O}(n^3)$
- for each new $\mathbf{b}$:
    - forward substitution $\mathcal{O}(n^2)$
    - back substitution $\mathcal{O}(n^2)$

Method 3: LU Decomposition

If $\mathbf{A}$ is square:

$$\mathbf{A} = \mathbf{L}\mathbf{U}$$

Important notes

- $\mathbf{L}$ and $\mathbf{U}$ are **not unique**

What “not unique” means

- there are **many valid pairs** $(\mathbf{L}, \mathbf{U})$ whose product equals $\mathbf{A}$
- different elimination choices (scaling, ordering, pivoting) produce different factors
- the matrices labeled $\mathbf{L}$ and $\mathbf{U}$ represent **roles**, not a single fixed decomposition

Consider the matrix

$$\mathbf{A} = \begin{bmatrix} 2 & 2 & 0 \\ 2 & 4 & 2 \\ 0 & 2 & 2 \end{bmatrix}$$

One valid LU factorization:

$$\mathbf{A} = \mathbf{L}_1\mathbf{U}_1, \qquad \mathbf{L}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 1 & 1 \end{bmatrix}, \quad \mathbf{U}_1 = \begin{bmatrix} 2 & 2 & 0 \\ 0 & 2 & 2 \\ 0 & 0 & 2 \end{bmatrix}$$

Another valid LU factorization (rescaling the diagonal):

$$\mathbf{A} = \mathbf{L}_2\mathbf{U}_2, \qquad \mathbf{L}_2 = \begin{bmatrix} 2 & 0 & 0 \\ 2 & 2 & 0 \\ 0 & 2 & 2 \end{bmatrix}, \quad \mathbf{U}_2 = \begin{bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

Both satisfy

$$\mathbf{L}_1\mathbf{U}_1 = \mathbf{L}_2\mathbf{U}_2 = \mathbf{A}.$$

LU Decomposition — Example Factorization

Start with a simple 3×3 matrix

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

Goal

$$\mathbf{A} = \mathbf{LU}$$

We create $\mathbf{U}$ by doing Gaussian Elimination on $\mathbf{A}$, and we build $\mathbf{L}$ by recording the elimination multipliers needed to form $\mathbf{U}$.

Why this works

- each elimination step subtracts a multiple of one row from another
- these row operations can be written as multiplication by a lower triangular matrix
- the multipliers used to eliminate entries below the diagonal are exactly the subdiagonal entries of $\mathbf{L}$

As a result

- Gaussian Elimination transforms $\mathbf{A}$ into $\mathbf{U}$
- the accumulated elimination operations form $\mathbf{L}$
- together, they satisfy

$$\mathbf{A} = \mathbf{L}\mathbf{U}$$

Step 1: Eliminate below the first pivot

First pivot

$$u_{11} = a_{11}$$

Elimination multipliers (what gets stored in $\mathbf{L}$)

$$\ell_{21} = \frac{a_{21}}{a_{11}}, \qquad \ell_{31} = \frac{a_{31}}{a_{11}}$$

Row operations (update the matrix to start forming $\mathbf{U}$)

$$R_2 \leftarrow R_2 - \ell_{21} R_1, \qquad R_3 \leftarrow R_3 - \ell_{31} R_1$$

Step 1: What $\mathbf{L}$ records

The multipliers used to eliminate entries become entries in $\mathbf{L}$.

After the first elimination stage:

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & 0 & 1 \end{bmatrix}$$

At this point, $\mathbf{U}$ is the partially eliminated matrix:

$$\mathbf{U} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a'_{22} & a'_{23} \\ 0 & a'_{32} & a'_{33} \end{bmatrix}$$

Step 2: Eliminate below the second pivot

Second pivot

$$u_{22} = a'_{22}$$

Multiplier for the entry below it (store in $\mathbf{L}$)

$$\ell_{32} = \frac{a'_{32}}{a'_{22}}$$

Row operation (finish forming $\mathbf{U}$)

$$R_3 \leftarrow R_3 - \ell_{32} R_2$$

Now the matrix is upper triangular:

$$\mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

Step 2: Final $\mathbf{L}$ and $\mathbf{U}$

Collect all stored multipliers:

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & \ell_{32} & 1 \end{bmatrix}$$

And the eliminated matrix is $\mathbf{U}$:

$$\mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

Interpretation

- $\mathbf{U}$ is what you get after Gaussian Elimination
- $\mathbf{L}$ stores the multipliers that created the zeros

Method 4: Cholesky Factorization

If $\mathbf{A}$ is **symmetric positive definite (SPD)**:

$$\mathbf{A} = \mathbf{L}\mathbf{U} = \mathbf{L}\mathbf{L}^T$$

Important notes

- symmetry implies the same elimination occurs above and below the diagonal
- the upper triangular factor satisfies

$$\mathbf{U} = \mathbf{L}^T$$

- minimal storage and fastest solves
- This is the fastest standard dense factorization for SPD systems.

In structural analysis:

- many stiffness matrices are SPD after applying boundary conditions (for typical linear elastic problems)

Constructing L

For an SPD matrix $\mathbf{A}$, the Cholesky factorization can be constructed column by column using the following recurrence relation

Diagonal entries

$$\ell_{ii} = \sqrt{a_{ii} - \sum_{k=1}^{i-1} \ell_{ik}^2}$$

Off-diagonal entries

$$\ell_{ji} = \frac{a_{ji} - \sum_{k=1}^{i-1} \ell_{jk}\,\ell_{ik}}{\ell_{ii}}, \qquad j = i+1, \ldots, n$$

But what if negative number in the root?

- this implies $\mathbf{A}$ is **not positive definite**
- the Cholesky factorization **fails**
- mathematically: $\mathbf{A}$ is indefinite

Structural interpretation

- in structural analysis, $\mathbf{A} = \mathbf{K}_{ff}$ (stiffness matrix)
- a negative value under the square root indicates **instability**
- the structure cannot resist certain deformation modes

Positive Definiteness — How to tell? (Engineering Intuition)

Given

$$\mathbf{A} = \begin{bmatrix} 4 & 1 & 0 \\ 1 & 3 & 1 \\ 0 & 1 & 2 \end{bmatrix}$$

Observations:

- $\mathbf{A}$ is **symmetric**
- All diagonal entries are **positive**
- Each diagonal term is **larger than the sum of off-diagonal terms in its row**

Structural interpretation

- Each DOF has more **self-stiffness** than coupling to neighboring DOFs
- Any nonzero displacement stores **positive strain energy**

Cholesky Example (Setup)

Compute the Cholesky factorization

$$\mathbf{A} = \mathbf{L}\mathbf{L}^T$$

for

$$\mathbf{A} = \begin{bmatrix} 100 & 30 & 20 & 10 \\ 30 & 178 & 201 & -36 \\ 20 & 201 & 485 & 21 \\ 10 & -36 & 21 & 350 \end{bmatrix}$$

We compute $\mathbf{L}$ column by column.

Recall: $\mathbf{L}$ stores the elimination coefficients used to zero entries below the diagonal in calculating $\mathbf{U}$

Column 1

Diagonal entry

$$\ell_{11} = \sqrt{a_{11}} = \sqrt{100} = 10$$

Below-diagonal entries

$$\ell_{21} = \frac{a_{21}}{\ell_{11}} = \frac{30}{10} = 3, \qquad \ell_{31} = \frac{a_{31}}{\ell_{11}} = \frac{20}{10} = 2, \qquad \ell_{41} = \frac{a_{41}}{\ell_{11}} = \frac{10}{10} = 1$$

After column 1

$$\mathbf{L} = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Column 2

Diagonal entry

$$\ell_{22} = \sqrt{a_{22} - \ell_{21}^2} = \sqrt{178 - 3^2} = \sqrt{169} = 13$$

Below-diagonal entries

$$\ell_{32} = \frac{a_{32} - \ell_{31}\ell_{21}}{\ell_{22}} = \frac{201 - (2 \cdot 3)}{13} = \frac{195}{13} = 15$$

$$\ell_{42} = \frac{a_{42} - \ell_{41}\ell_{21}}{\ell_{22}} = \frac{-36 - (1 \cdot 3)}{13} = \frac{-39}{13} = -3$$

After column 2

$$\mathbf{L} = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 3 & 13 & 0 & 0 \\ 2 & 15 & 0 & 0 \\ 1 & -3 & 0 & 0 \end{bmatrix}$$

Column 3

Diagonal entry

$$\ell_{33} = \sqrt{a_{33} - \ell_{31}^2 - \ell_{32}^2} = \sqrt{485 - 2^2 - 15^2} = \sqrt{485 - 4 - 225} = \sqrt{256} = 16$$

Below-diagonal entry

$$\ell_{43} = \frac{a_{43} - \ell_{41}\ell_{31} - \ell_{42}\ell_{32}}{\ell_{33}} = \frac{21 - (1 \cdot 2) - ((-3) \cdot 15)}{16} = \frac{21 - 2 + 45}{16} = \frac{64}{16}$$
$$= 4$$

After column 3

$$\mathbf{L} = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 3 & 13 & 0 & 0 \\ 2 & 15 & 16 & 0 \\ 1 & -3 & 4 & 0 \end{bmatrix}$$

Column 4

Diagonal entry

$$\ell_{44} = \sqrt{a_{44} - \ell_{41}^2 - \ell_{42}^2 - \ell_{43}^2} = \sqrt{350 - 1^2 - (-3)^2 - 4^2} = \sqrt{350 - 1 - 9 - 16}$$
$$= \sqrt{324} = 18$$

Final factor

$$\mathbf{L} = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 3 & 13 & 0 & 0 \\ 2 & 15 & 16 & 0 \\ 1 & -3 & 4 & 18 \end{bmatrix}$$

Check

Compute

$$\mathbf{L}\mathbf{L}^T = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 3 & 13 & 0 & 0 \\ 2 & 15 & 16 & 0 \\ 1 & -3 & 4 & 18 \end{bmatrix} \begin{bmatrix} 10 & 3 & 2 & 1 \\ 0 & 13 & 15 & -3 \\ 0 & 0 & 16 & 4 \\ 0 & 0 & 0 & 18 \end{bmatrix}$$

$$= \begin{bmatrix} 100 & 30 & 20 & 10 \\ 30 & 178 & 201 & -36 \\ 20 & 201 & 485 & 21 \\ 10 & -36 & 21 & 350 \end{bmatrix} = \mathbf{A}$$

Method 5: LDL^T Factorization

Also known as root-free Cholesky

If $\mathbf{A}$ is symmetric (does not need to be definite):

$$\mathbf{A} = \mathbf{LDU} = \mathbf{LDL}^T$$

Important notes

- symmetry implies the same elimination occurs above and below the diagonal
- the upper triangular factor satisfies

$$\mathbf{U} = \mathbf{L}^T$$

- $\mathbf{L}$ is unit lower triangular
- $\mathbf{D}$ is diagonal and retains the scaling
- improved numberical stability

Constructing L and D

For a symmetric matrix $\mathbf{A}$, the LDL^T factorization

$$\mathbf{A} = \mathbf{L}\mathbf{D}\mathbf{L}^T$$

can be constructed column by column using the following recurrence relations.

Diagonal entries of $\mathbf{D}$ (no square root required!)

$$d_{ii} = a_{ii} - \sum_{k=1}^{i-1} d_{kk}\,\ell_{ik}^2$$

Diagonal entries of $\mathbf{L}$

$$\ell_{ii} = 1$$

Off-diagonal entries of $\mathbf{L}$

$$\ell_{ji} = \frac{a_{ji} - \sum_{k=1}^{i-1} d_{kk}\,\ell_{jk}\,\ell_{ik}}{d_{ii}}, \qquad j = i+1, \ldots, n$$

Method 6: Thomas Method

Only in very special case, Tridiagonal Matrices

If $\mathbf{A}$ is **tridiagonal** (bandwidth = 3):

- only $a_{i,i-1}$, $a_{i,i}$, $a_{i,i+1}$ are nonzero

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & \cdots & 0 \\ a_{21} & a_{22} & a_{23} & 0 & \cdots & 0 \\ 0 & a_{32} & a_{33} & a_{34} & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & a_{n,n-1} & a_{nn-1} & a_{nn} \\ 0 & \cdots & 0 & 0 & a_{n,n-1} & a_{nn} \end{bmatrix}$$

Thomas algorithm is a specialized LU that runs in $\mathcal{O}(n)$ time.

This appears often in:

- 1D finite difference / finite element problems
- beam/rod discretizations with local coupling (each DOF interacts only with its neighbors)

Final Remark (All Direct Methods)

Once the matrix is factorized, the factors do not change; only $\mathbf{b}$ does.

LU

- $\mathbf{L}\hat{\mathbf{b}} = \mathbf{b}$
- $\mathbf{U}\mathbf{x} = \hat{\mathbf{b}}$

Cholesky

- $\mathbf{L}\hat{\mathbf{b}} = \mathbf{b}$
- $\mathbf{L}^T\mathbf{x} = \hat{\mathbf{b}}$

LDL^T

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

Same forward/backward substitution structure for all methods.

Part 3 — Iterative Methods

Replace one big solve with a sequence of cheap updates: $\mathbf{x}^{(k)} \to \mathbf{x}^{(k+1)}$.

Why Iterative Solvers?

Iterative methods become attractive when:

- $\mathbf{A}$ is **very large** and sparse
- you only need an approximate solution (within tolerance)
- factorization would cause too much fill-in (memory)

But:

- convergence is not guaranteed for all matrices
- performance depends strongly on conditioning and preconditioning
- can take longer

Method 1: Point Jacobi

The Point Jacobi method solves

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

by starting with an initial guess $\mathbf{x}^{(0)}$ and repeatedly updating each unknown using values from the **previous** iteration.

Key idea

- compute a new vector $\mathbf{x}^{(m)}$ using only $\mathbf{x}^{(m-1)}$
- all components are updated “in parallel” (no immediate reuse)

Requirement

- diagonal entries must be nonzero: $a_{ii} \neq 0$

Component Form (3×3 Example)

System

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Update formula (component form)

$$x_i^{(m)} = \frac{b_i - \sum_{j \neq i} a_{ij}\, x_j^{(m-1)}}{a_{ii}}, \qquad i = 1, \ldots, n$$

Equation Form (3×3 Example)

Start from the linear system at iteration m:

$$\begin{aligned} a_{11}x_1^{(m-1)} + a_{12}x_2^{(m-1)} + a_{13}x_3^{(m-1)} &= b_1 \\ a_{21}x_1^{(m-1)} + a_{22}x_2^{(m-1)} + a_{23}x_3^{(m-1)} &= b_2 \\ a_{31}x_1^{(m-1)} + a_{32}x_2^{(m-1)} + a_{33}x_3^{(m-1)} &= b_3 \end{aligned}$$

Jacobi updates (use only previous iteration values)

$$\boxed{x_1^{(m)}} = \frac{b_1 - a_{12}x_2^{(m-1)} - a_{13}x_3^{(m-1)}}{a_{11}}$$

$$\boxed{x_2^{(m)}} = \frac{b_2 - a_{21}x_1^{(m-1)} - a_{23}x_3^{(m-1)}}{a_{22}}$$

$$\boxed{x_3^{(m)}} = \frac{b_3 - a_{31}x_1^{(m-1)} - a_{32}x_2^{(m-1)}}{a_{33}}$$

One iteration step

- start with $\mathbf{x}^{(m-1)}$
- compute $x_1^{(m)}, x_2^{(m)}, x_3^{(m)}$
- collect into the new vector $\mathbf{x}^{(m)}$
- repeat iteration

Method 2: Point Gauss–Seidel

The Point Gauss–Seidel method solves

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

by starting with an initial guess $\mathbf{x}^{(0)}$ and repeatedly updating each unknown using the **newest available values**.

Key idea

- compute $\mathbf{x}^{(m)}$ by immediately reusing updated components
- values $x_1^{(m)}, \ldots, x_{i-1}^{(m)}$ are reused when computing $x_i^{(m)}$
- updates are **sequential**, not parallel

Requirement

- diagonal entries must be nonzero: $a_{ii} \neq 0$

Notes

- often converges faster than Point Jacobi
- less parallel, but more efficient per iteration

Component Form (3×3 Example)

System

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Update formula (component form)

$$x_i^{(m)} = \frac{b_i - \sum_{j<i} a_{ij}\, x_j^{(m)} - \sum_{j>i} a_{ij}\, x_j^{(m-1)}}{a_{ii}}, \qquad i = 1, \ldots, n$$

Equation Form (3×3 Example)

Start from the linear system at iteration m:

$$\begin{aligned} a_{11}x_1^{(m)} + a_{12}x_2^{(m-1)} + a_{13}x_3^{(m-1)} &= b_1 \\ a_{21}x_1^{(m)} + a_{22}x_2^{(m)} + a_{23}x_3^{(m-1)} &= b_2 \\ a_{31}x_1^{(m)} + a_{32}x_2^{(m)} + a_{33}x_3^{(m)} &= b_3 \end{aligned}$$

Gauss–Seidel updates
(reuse newest available values immediately)

$$\boxed{x_1^{(m)}} = \frac{b_1 - a_{12}x_2^{(m-1)} - a_{13}x_3^{(m-1)}}{a_{11}}$$

$$\boxed{x_2^{(m)}} = \frac{b_2 - a_{21}\boxed{x_1^{(m)}} - a_{23}x_3^{(m-1)}}{a_{22}}$$

$$\boxed{x_3^{(m)}} = \frac{b_3 - a_{31}\boxed{x_1^{(m)}} - a_{32}\boxed{x_2^{(m)}}}{a_{33}}$$

One iteration step

- start with $\mathbf{x}^{(m-1)}$
- compute $x_1^{(m)}$ and reuse it immediately
- compute $x_2^{(m)}$ using updated $x_1^{(m)}$
- compute $x_3^{(m)}$ using updated $x_1^{(m)}, x_2^{(m)}$
- collect into the new vector $\mathbf{x}^{(m)}$
- repeat iteration

Jacobi vs. Gauss–Seidel (Same Equations, Different Reuse)

Point Jacobi (parallel update)

$$\boxed{x_1^{(m)}} = \frac{b_1 - a_{12}x_2^{(m-1)} - a_{13}x_3^{(m-1)}}{a_{11}}$$

$$\boxed{x_2^{(m)}} = \frac{b_2 - a_{21}x_1^{(m-1)} - a_{23}x_3^{(m-1)}}{a_{22}}$$

$$\boxed{x_3^{(m)}} = \frac{b_3 - a_{31}x_1^{(m-1)} - a_{32}x_2^{(m-1)}}{a_{33}}$$

Point Gauss-Seidel (sequential reuse)

$$\boxed{x_1^{(m)}} = \frac{b_1 - a_{12}x_2^{(m-1)} - a_{13}x_3^{(m-1)}}{a_{11}}$$

$$\boxed{x_2^{(m)}} = \frac{b_2 - a_{21}\boxed{x_1^{(m)}} - a_{23}x_3^{(m-1)}}{a_{22}}$$

$$\boxed{x_3^{(m)}} = \frac{b_3 - a_{31}\boxed{x_1^{(m)}} - a_{32}\boxed{x_2^{(m)}}}{a_{33}}$$

Key difference

- Jacobi uses only $\mathbf{x}^{(m-1)}$
- Gauss–Seidel reuses the newest values immediately

Method 3: Conjugate Gradient (CG)

For **large, sparse, SPD** systems, CG is a standard choice:

- can converge in far fewer iterations than Jacobi/GS
- performance depends heavily on **preconditioning**

> *We will not cover this method in detail. Section 11.3.2 of McGuire text*

Choosing Convergence Criteria

> *Iterative methods are repeated until the solution is "close enough" to the true solution.*

Convergence Criterion — Relative Error

A commonly used convergence criterion is

$$\varepsilon_a < \zeta$$

where the approximate relative error is defined as

$$\varepsilon_a = \max_{i=1,\ldots,n} \left| \frac{x_i^{(m)} - x_i^{(m-1)}}{x_i^{(m)}} \right| \times 100\%.$$

Tolerance

$$\zeta = 0.5 \times 10^{\,2-q}\ \%$$

where q is the desired number of correct significant figures.

Notes

- convergence depends on the initial guess
- tighter tolerances require more iterations
- for positive definite matrices, convergence is guaranteed

Convergence Criterion — Residual Norm

An alternative (and often more physically meaningful) convergence criterion is based on the **residual**.

The residual vector is defined as

$$\mathbf{r}^{(m)} = \mathbf{A}\mathbf{x}^{(m)} - \mathbf{b}.$$

A commonly used stopping criterion is

$$\left\|\mathbf{r}^{(m)}\right\| < \varepsilon_r,$$

where $\|\cdot\|$ is typically the Euclidean (2-)norm.

Notes

- The residual measures **how well the current iterate satisfies the equations**
- $\mathbf{r} = \mathbf{0}$ corresponds to an exact solution
- A small residual means the solution is **nearly in equilibrium**

The **Euclidean (2-)norm** measures the magnitude of a vector as the square root of the sum of the squares of its components:

$$\|\mathbf{r}\|_2 = \sqrt{r_1^2 + r_2^2 + \cdots + r_n^2}.$$

In practice, both **relative change** and **residual norms** are often monitored together.

Relaxation and Over-Relaxation

> *To improve convergence behavior, relaxation schemes may be applied.*

After computing a new iterate $x_i^{(m)}$, the value is modified (weighted average)

$$x_i^{(m)} \leftarrow \beta\, x_i^{(m)} + (1 - \beta)\, x_i^{(m-1)}$$

Relaxation parameter β

- $0 < \beta < 1$ → **under-relaxation**
  (damps oscillations, stabilizes convergence)
- $\beta = 1$ → no relaxation (standard method)
- $1 < \beta < 2$ → **over-relaxation**
  (can significantly accelerate convergence)

Notes

- over-relaxation is widely used in structural analysis
- optimal β is problem-dependent
- improper choice of β may cause divergence

Wrap-Up: Choosing a Solver

Given $\mathbf{A}\mathbf{x} = \mathbf{b}$, ask:

1. **Is $\mathbf{A}$ symmetric?**
   - Yes → consider LDL^T or Cholesky
2. **Is $\mathbf{A}$ symmetric positive definite (SPD)?**
   - Yes → Cholesky (direct), Conjugate Gradient (iterative)
3. **Is $\mathbf{A}$ tridiagonal, banded, or sparse?**
   - Yes → specialized solvers
     (Thomas algorithm, banded solvers, frontal methods)
4. **Do I have many right-hand sides $\mathbf{b}$?**
   - Yes → factorize once (LU / LDL^T / Cholesky) and reuse substitutions

Looking Ahead

Next week, we begin assembling **stiffness matrices** for structural systems and applying these linear solvers to real structural workflows.