POLITECNICO
MILANO 1863

DEPARTMENT
OF CHEMISTRY MATERIALS
AND CHEMICAL
ENGINEERING

Linear System Of Equations
Part 1

Calcoli di Processo dell' Ingegneria Chimica

Timoteo Dinelli

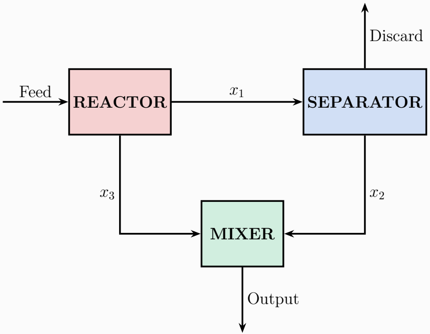
9th of October 2025

Department of Chemistry, Materials and Chemical Engineering, "Giulio Natta", Politecnico di Milano.

email: timoteo.dinelli@polimi.it

Motivation: Why Linear Systems?

Consider a chemical process with multiple unit operations. Mass balance equations for each component form a system of equations:



System of Equations:

$$\begin{cases} \text{Feed} - x_1 - x_3 = 0 \\ x_1 - x_2 - \text{Discard} = 0 \\ x_2 + x_3 - \text{Output} = 0 \end{cases}$$

Matrix form: $\mathbf{Ax} = \mathbf{b}$

$$\begin{bmatrix} -1 & 0 & -1 \\ 1 & -1 & 0 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -\text{Feed} \\ \text{Discard} \\ \text{Output} \end{bmatrix}$$

Where x_1, x_2, x_3 represent flow rates. Such systems appear everywhere in engineering: heat transfer, electrical circuits, structural analysis, and process optimization.

How do we solve these efficiently and accurately?

Linear System Of Equations

A system of linear equations consists of several linear equations that must all be satisfied simultaneously. A solution is a vector whose elements, when substituted for the unknowns, satisfy all equations.

From the classical representation to the **matrix** form:

$$\begin{cases} a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,n}x_n = b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,n}x_n = b_2 \\ \qquad \vdots \\ a_{n,1}x_1 + a_{n,2}x_2 + \ldots + a_{n,n}x_n = b_n \end{cases} \qquad \begin{bmatrix} a_{1,1} & a_{1,2} & \ldots & a_{1,n} \\ a_{2,1} & a_{2,2} & \ldots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n,1} & a_{n,2} & \ldots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

$$\mathrm{A}\,\mathrm{x} = \mathrm{b}$$

Why Not Just Invert the Matrix?

The "obvious" solution would be:

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

This approach is impractical! Here's why:

- **Computational cost:** Computing $\mathbf{A}^{-1}$ requires $\sim O(n^3)$ operations, same as solving the system directly
- **Numerical instability:** Direct inversion amplifies rounding errors, especially for ill-conditioned matrices
- **Memory:** Storing the full inverse matrix requires n^2 memory locations
- **Singularity:** If $\det(\mathbf{A}) = 0$, the inverse doesn't exist

The Goal: Triangular Systems

Consider this simple 3×3 upper triangular system:

$$\begin{cases} 3x + 89y + 66z = 87 \\ 65y + 9z = 7 \\ 46z = 3 \end{cases} \qquad \begin{bmatrix} 3 & 89 & 66 \\ 0 & 65 & 9 \\ 0 & 0 & 46 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} 87 \\ 7 \\ 3 \end{bmatrix}$$

This is **easy to solve** by back-substitution:

$$z = \frac{3}{46} \approx 0.065$$
$$y = \frac{7 - 9z}{65} \approx 0.098$$
$$x = \frac{87 - 89y - 66z}{3} \approx 26.09$$

Cost: Only $O(n^2)$ operations!

Our goal: Transform any system into triangular form.

Gauss Elimination: General Algorithm

Given $\mathbf{Ax} = \mathbf{b}$, form the augmented matrix $\mathbf{A}^* = [\mathbf{A} \,|\, \mathbf{b}]$

$$\mathbf{A}^* = [\mathbf{A} \,|\, \mathbf{b}] = \left[\begin{array}{ccc|c} a_{1,1}^{(0)} & \dots & a_{1,n}^{(0)} & b_1^{(0)} \\ \vdots & \ddots & \vdots & \vdots \\ a_{n,1}^{(0)} & \dots & a_{n,n}^{(0)} & b_n^{(0)} \end{array}\right]$$

Superscript (k) indicates the state after k elimination steps. After $n-1$ elimination steps, we obtain:

$$\mathbf{A}^* = \left[\begin{array}{cccc|c} a_{1,1}^{(0)} & \dots & \dots & a_{1,n}^{(0)} & b_1^{(0)} \\ 0 & a_{2,2}^{(1)} & \dots & a_{2,n}^{(1)} & b_2^{(1)} \\ \vdots & \dots & \ddots & \vdots & \vdots \\ 0 & \dots & 0 & a_{n,n}^{(n-1)} & b_n^{(n-1)} \end{array}\right]$$

At step k: eliminate column k below the diagonal using multipliers formula

$$m_{i,k} = \frac{a_{i,k}^{(k-1)}}{a_{k,k}^{(k-1)}}$$

Why Triangular Matrices?

Key insight: Triangular systems are trivial to solve!

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,n-1} & a_{1,n} \\ 0 & a_{2,2} & \dots & a_{2,n-1} & a_{2,n} \\ 0 & 0 & \dots & a_{3,n-1} & a_{3,n} \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & a_{n,n} \end{bmatrix} \mathbf{x} = \mathbf{b}^*$$

Back-substitution algorithm:

$$x_n = \frac{b_n^*}{a_{n,n}} \qquad x_i = \frac{1}{a_{i,i}} \left(b_i^* - \sum_{j=i+1}^{n} a_{i,j} x_j \right) \quad \text{for } i = n-1, n-2, \dots, 1$$

LU Factorization: The Idea

Instead of modifying $\mathbf{A}$ repeatedly, decompose it once:

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

where $\mathbf{L}$ is lower triangular (with 1's on diagonal) and $\mathbf{U}$ is upper triangular.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \qquad \mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ \ell_{21} & 1 & 0 \\ \ell_{31} & \ell_{32} & 1 \end{bmatrix} \qquad \mathbf{U} = \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

Connection to Gauss elimination: The multipliers $m_{i,k}$ from Gauss elimination become the entries $\ell_{i,k}$ of $\mathbf{L}$. Matrix $\mathbf{U}$ is the final upper triangular form.

Solving with LU Decomposition

Original problem: $\mathbf{A}\mathbf{x} = \mathbf{b}$ $\rightarrow$ Substitute $\mathbf{A} = \mathbf{L}\mathbf{U}$ $\rightarrow$ Equation $\mathbf{L}\mathbf{U}\mathbf{x} = \mathbf{b}$

Two-step solution process:

Step 1: Forward substitution - Solve $\mathbf{L}\mathbf{y} = \mathbf{b}$ for $\mathbf{y}$

$$y_i = b_i - \sum_{j=1}^{i-1} \ell_{i,j} y_j \quad \text{for } i = 1, 2, \ldots, n$$

Step 2: Back-substitution - Solve $\mathbf{U}\mathbf{x} = \mathbf{y}$ for $\mathbf{x}$

$$x_i = \frac{1}{u_{i,i}} \left(y_i - \sum_{j=i+1}^{n} u_{i,j} x_j \right) \quad \text{for } i = n, n-1, \ldots, 1$$

Each step costs $O(n^2)$ operations. The decomposition costs $O(n^3)$ but is done only once!

Computational Complexity Comparison

Method	Operations	Comment
Direct inversion ($\mathbf{A}^{-1}$)	$\sim \frac{2n^3}{3}$	Numerically unstable
Gauss elimination	$\sim \frac{n^3}{3}$	Good for single **b**
LU decomposition	$\sim \frac{n^3}{3}$	Reusable for multiple **b**
Forward/back substitution	$\sim n^2$	Using existing **L**, **U**

Why Use LU Decomposition?

- **Multiple right-hand sides:** Once $\mathbf{A} = \mathbf{LU}$ is computed, solving for different $\mathbf{b}$ vectors costs only $O(n^2)$ each (useful in optimization, time-stepping schemes, Newton methods)
- **Transpose systems:** Can solve $\mathbf{A}^T\mathbf{x} = \mathbf{c}$ using $\mathbf{A}^T = \mathbf{U}^T\mathbf{L}^T$ without new factorization
- **Matrix properties:** Easy to compute $\det(\mathbf{A}) = \prod_{i=1}^{n} u_{ii}$ and check invertibility
- **Efficient updates:** Special techniques can update $\mathbf{L}$ and $\mathbf{U}$ when $\mathbf{A}$ is slightly modified (rank-1 updates, Sherman-Morrison formula)
- **MATLAB note:** The built-in `[L,U,P] = lu(A)` function includes pivoting (permutation matrix $\mathbf{P}$) for numerical stability. Always check documentation for output format!

When Methods Can Fail

Singular matrices: If $\det(A) = 0$, the system has either:

- No solution (inconsistent)
- Infinitely many solutions (underdetermined)

Numerical issues during elimination:

- Zero pivot: If $a_{k,k}^{(k-1)} = 0$, division by zero occurs
- Small pivot: If $a_{k,k}^{(k-1)} \approx 0$, amplifies rounding errors

Solution: Partial pivoting

- At each step, swap rows to bring the largest element to the pivot position
- Improves numerical stability significantly
- MATLAB's `lu(A)` and `linsolve(A, b)` use pivoting by default

Exercises

Exercise 1: Triangular System Solver

Implement a function that solves upper triangular systems using back-substitution.

Function signature:

Function: x = solve_upper_triangular(U, b)

Input: $n \times n$ upper triangular matrix **U**, vector **b** of size $n \times 1$

Output: Solution vector **x** of size $n \times 1$

Algorithm hints:

Start from the last equation: $x_n = b_n / U_{n,n}$

Use a `for` loop with index `i` from `n-1` down to `1`

For each x_i: subtract contributions from already-computed x_j (where $j > i$)

Formula: $x_i = (b_i - \sum_{j=i+1}^{n} U_{i,j} \cdot x_j)/U_{i,i}$

Test: $\begin{bmatrix} 2 & 3 \\ 0 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 8 \\ 12 \end{bmatrix}$

Answer: $x_1 = -0.5$, $x_2 = 3$

Exercise 2: Gauss Elimination

Transform matrix **A** into upper triangular form using Gauss elimination.

Function signature:

- **Function:** [U, b_new] = gauss_eliminate(A, b)
- **Input:** $n \times n$ matrix **A**, vector **b** of size $n \times 1$
- **Output:** Upper triangular matrix **U**, modified vector **b_new**

Note: This version does not include pivoting. Assumes all pivot elements are non-zero.

Exercise 3: Complete Linear Solver

Combine your functions into a complete solver and compare with MATLAB.

Function signature:

- **Function:** `x = my_linear_solver(A, b)`
- **Should call:** `gauss_eliminate` then `solve_upper_triangular`

Test systems:

$$\begin{bmatrix} 1 & 2 & -1 & 2 \\ 1 & 0 & 2 & 1 \\ 2 & 1 & 0 & -2 \\ 0 & 0 & -1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ t \end{bmatrix} = \begin{bmatrix} 3 \\ 1 \\ 1 \\ 2 \end{bmatrix} \qquad \begin{bmatrix} 1 & 45 & 0 & -1 \\ 1 & 0 & 0 & -3 \\ 1 & 1 & 1 & 0 \\ 1 & -1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ t \end{bmatrix} = \begin{bmatrix} 6 \\ 12 \\ -6 \\ 12 \end{bmatrix}$$

Verification: Compare your results with MATLAB's built-in:

`x_matlab = A \ b` (recommended), `x_matlab = linsolve(A, b)`

Exercise 4: LU Decomposition

Implement LU decomposition and integrate it into your solver.

Function signature:

- **Function:** `[L, U] = my_lu_decompose(A)`
- **Input:** $n \times n$ matrix **A**
- **Output:** Lower triangular **L** (with 1's on diagonal), upper triangular **U**

Algorithm hints:

- Initialize: `L = eye(n)`, `U = A`
- For each column **k** from 1 to `n-1`:
    - For each row `i` from `k+1` to `n`:
    - Store multiplier in **L**: `L(i,k) = U(i,k) / U(k,k)`
    - Eliminate in **U**: `U(i,:) = U(i,:) - L(i,k) * U(k,:)`
- Create the solver assembling the decomposition and the solution routines.

Expected Solutions

Use these to verify your implementations are correct!

Test System 1:

$$\begin{cases} x + 2y - z + 2t = 3 \\ x + 2z + t = 1 \\ 2x + y - 2t = 1 \\ -z + t = 2 \end{cases}$$

Solution: $x = 2, y = -1, z = -1, t = 1$

Test System 2:

$$\begin{cases} x + 45y - t = 6 \\ x - 3t = 12 \\ x + y + z = -6 \\ x - y + z + t = 12 \end{cases}$$

Solution: $x = 60.8571, y = -0.8571, z = -66, t = 16.2857$

Coding Best Practices

Tips for your implementation:

- **Error checking:** Verify matrix dimensions match before operations
- **Zero pivots:** Add a check: `if abs(U(k,k)) < eps, error('Zero pivot'); end`
- **Vectorization:** In MATLAB, `U(i,:) = U(i,:) - m*U(k,:)` is more efficient than element-wise loops
- **Testing:** Create simple 2×2 test cases first, then scale up
- **Residual check:** Compute $\|\mathbf{Ax} - \mathbf{b}\|$ to verify accuracy
- **Comparison:** Always compare with `A\b` for validation

Coding Best Practices

Common mistakes to avoid:

- Not initializing output vectors (use `x = zeros(n,1)`)
- Loop indices in wrong direction for back-substitution
- Forgetting to update both A and b during elimination

Thank you for your attention!