Akademia Górniczo-Hutnicza
im. Stanisława Staszica w Krakowie

AGH University of Krakow

Numerical Methods in Physics

Project 02
Solving Systems of Linear Equations Using LU Decomposition

Hamzeh Khanpour
hamzeh.khanpour@fis.agh.edu.pl
https://view.fis.agh.edu.pl/staff/hamzeh.khanpour/

Faculty of Physics and Applied Computer Science
AGH University of Krakow

16 October 2026

Why this laboratory?

Two questions

How do we solve $A\mathbf{x} = \mathbf{b}$ using LU decomposition in C++?

How do we check the solution and its sensitivity to the input data?

Scientific workflow

Model $\longrightarrow$ factorize $\longrightarrow$ solve $\longrightarrow$ verify $\longrightarrow$ interpret.

Course sources

S01: Tomasz Chwiej, Numerical Methods in Physics course page.
S02: Project 02 handout: Shaping the polynomial, 12 March 2026.
S03: Lecture 2: Direct methods for solving linear systems.

Technical sources

Netlib LAPACKE: C interface to LAPACK.
LAPACK routines: `DGETRF` (pivoted LU) and `DGETRS` (solve using LU).
N. J. Higham: *What Is Backward Error?*

The goal: solve a system and trust the result

A direct method turns a coupled system into a finite sequence of algebraic operations. Our unknowns are collected in a vector:

$$A\boldsymbol{x} = \boldsymbol{b}.$$

Understand the method, not only the function call

You will connect elimination to LU, handle row permutations, call a numerical library, and check the result using the original equations.

The laboratory application

Construct the coefficients of a fifth-degree shaping polynomial. The examples teach each tool; you assemble the assigned program yourself.

Why direct methods matter in physics

Discretizing a differential equation replaces a continuous field by values at finitely many nodes. The unknown field values then satisfy coupled algebraic equations.

$$-V_{i-1} + 2V_i - V_{i+1} = h^2 s_i.$$

Here h is a grid spacing and s_i is a prescribed source.

A two-node illustration

With zero boundary values,

$$\begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix} \begin{pmatrix} V_1 \\ V_2 \end{pmatrix} = h^2 \begin{pmatrix} s_1 \\ s_2 \end{pmatrix}.$$

This motivates linear solvers; it is not today's assessed problem.

One matrix, several experiments

When A is unchanged but the source $\boldsymbol{b}$ changes, reuse the factors. There is no need to form an explicit inverse.

How this 135-minute laboratory works

Time	Activity
0-10	Aim, files, and essential readiness.
10-40	Selected hand calculations and complete C++ demonstrations.
40-50	Official tasks, outputs, and project briefing.
50-120	**70 minutes: develop and test your own program.**
120-135	Numerical checks, interpretation, and home-report guidance.

Use the slides as a reference

The 56-page deck is not a 56-page uninterrupted lecture. Detailed listings and optional material remain available while you work.

Finished examples; an independent project

In `examples/`

E01–E09 are complete toy programs. E10 is optional. Compile each file separately, predict its output, run it, then explain its checks.

In your own project folder

Build the prescribed polynomial system, compute your coefficients and determinant, save your results, and investigate the required node changes.

```
pwd
ls examples
mkdir -p bin data plots
g++ --version
```

Run these commands in the Linux terminal. Never link all ten `.cpp` files together: each has its own `main()`.

Anatomy of a linear system

$$\underbrace{\begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}}_{A:\,2\times2} \underbrace{\begin{pmatrix} x_1 \\ x_2 \end{pmatrix}}_{\boldsymbol{x}:\,2\times1} = \underbrace{\begin{pmatrix} 4 \\ 3 \end{pmatrix}}_{\boldsymbol{b}:\,2\times1}.$$

The matrix contains the coefficients; the right-hand side contains the known values. Each row represents one equation. The column order fixes the order of the unknowns.

Two indexing conventions

Mathematical components here are x_1, x_2. In C++, they are stored in `x[0]` and `x[1]`. A matrix entry becomes an array offset; it is not a new unknown.

A necessary qualification

A square system has a unique solution only when A is nonsingular. Equal numbers of equations and unknowns alone are not enough.

A small system solved completely by hand

$$2x_1 + x_2 = 4 \qquad \text{(equation 1)},$$
$$x_1 + x_2 = 3 \qquad \text{(equation 2)}.$$

Subtract equation 2 from equation 1:

$$(2x_1 + x_2) - (x_1 + x_2) = 4 - 3 \quad \Longrightarrow \quad x_1 = 1.$$

Substitute into the second equation:

$$x_2 = 3 - x_1 = 3 - 1 = 2.$$

Always return to the original equations

$2(1) + 2 = 4$ and $1 + 2 = 3$. Both conditions hold, so $\boldsymbol{x} = (1, 2)^T$. E01 implements this same subtraction and checks both rows.

E01 | Compile, run, and verify

Excerpt: `01_small_linear_system.cpp`, lines 11–16. Full program is in `examples/`.

```
    const std::array<double, 4> A = {2, 1, 1, 1};
    const std::array<double, 2> b = {4, 3};
    // Subtract equation 2 from equation 1, then substitute.
    const double x0 = b[0] - b[1];
    const double x1 = b[1] - x0;
    const std::array<double, 2> x = {x0, x1};
```

```
g++ -std=c++17 -Wall -Wextra -Wpedantic -O2 \
  examples/01_small_linear_system.cpp \
  -o bin/01_small_linear_system
./bin/01_small_linear_system
```

Expected output and interpretation

`x = 1 2` and `residual_inf = 0`.
The complete source also evaluates $b_i - \sum_j A_{ij} x_j$, checks finite values, and returns a nonzero exit code on a failed check.

This program uses the algebra of this fixed 2×2 example; it is not a general linear-system solver.

Why use Gaussian elimination?

For more unknowns, systematic row operations are easier to organize than ad hoc substitution. We will use one fixed example throughout the next steps:

$$A = \begin{pmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{pmatrix}, \qquad \boldsymbol{b} = \begin{pmatrix} 4 \\ 10 \\ 24 \end{pmatrix}.$$

Our route

Remove the entries below each pivot, obtain an upper triangular system, then solve from its last equation upward.

Prediction before programming

The exact solution is $(1, 1, 1)^T$: the row sums are 4, 10, and 24. This known answer lets us check every stage of E03.

Elementary row operations preserve the solution

An equation can be exchanged with another, multiplied by a nonzero number, or replaced by itself plus a multiple of another equation. Apply every operation to **both** the matrix row and its right-hand side.

$$R_i \leftarrow R_i - m_{ik} R_k, \qquad m_{ik} = \frac{A_{ik}}{A_{kk}}.$$

The denominator is the current **pivot**. The multiplier is chosen so that the new entry in column k is zero:

$$A_{ik} - \frac{A_{ik}}{A_{kk}} A_{kk} = 0.$$

One explicit calculation

For our first two rows, $m_{21} = 4/2 = 2$.

$$[4, 3, 3 \mid 10] - 2[2, 1, 1 \mid 4] = [0, 1, 1 \mid 2].$$

A zero pivot requires a different row; see E05.

Start with the augmented matrix; the vertical bar separates A from $\boldsymbol{b}$.

$$\left[\begin{array}{ccc|c} 2 & 1 & 1 & 4 \\ 4 & 3 & 3 & 10 \\ 8 & 7 & 9 & 24 \end{array}\right].$$

Use $m_{21} = 2$ and $m_{31} = 4$:

$$\begin{aligned} R_2 \leftarrow R_2 - 2R_1 : &\qquad [4,3,3 \mid 10] - [4,2,2 \mid 8] = [0,1,1 \mid 2], \\ R_3 \leftarrow R_3 - 4R_1 : &\qquad [8,7,9 \mid 24] - [8,4,4 \mid 16] = [0,3,5 \mid 8]. \end{aligned}$$

$$\Longrightarrow \quad \left[\begin{array}{ccc|c} 2 & 1 & 1 & 4 \\ 0 & 1 & 1 & 2 \\ 0 & 3 & 5 & 8 \end{array}\right].$$

The first column is now triangular. The second pivot is the updated entry 1.

The remaining multiplier is $m_{32} = 3/1 = 3$:

$$R_3 \leftarrow R_3 - 3R_2 : \quad [0, 3, 5 \mid 8] - [0, 3, 3 \mid 6] = [0, 0, 2 \mid 2].$$

$$\left[\begin{array}{ccc|c} 2 & 1 & 1 & 4 \\ 0 & 1 & 1 & 2 \\ 0 & 0 & 2 & 2 \end{array}\right] \quad \Longrightarrow \quad \begin{cases} 2x_3 = 2 & \Rightarrow x_3 = 1, \\ x_2 + x_3 = 2 & \Rightarrow x_2 = 1, \\ 2x_1 + x_2 + x_3 = 4 & \Rightarrow x_1 = 1. \end{cases}$$

```
make bin/03_gaussian_elimination
./bin/03_gaussian_elimination
```

E03 prints the multipliers, all nine entries of U, the transformed right-hand side $(4, 2, 2)^T$, and a check using the original equations. It deliberately has no pivoting for this safe toy fixture.

Row-major storage and leading dimensions

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \quad \longleftrightarrow \quad \texttt{a = [1,2,3,4,5,6]}.$$

With zero-based indices, contiguous rows of length three obey

$$\boxed{\texttt{a[i*lda+j]} = A_{ij}}, \qquad \texttt{lda} = 3.$$

Offset	0	1	2	3	4	5
Value	1	2	3	4	5	6
Logical entry	$(0,0)$	$(0,1)$	$(0,2)$	$(1,0)$	$(1,1)$	$(1,2)$

Leading dimension means physical row stride

`lda` is not the number of rows. If padding extends each row to five slots, the same entry $A_{1,2} = 6$ is at offset $1 \cdot 5 + 2 = 7$.

Excerpt: 02_row_major_storage.cpp, lines 9–16. Full program is in examples/.

```
    const int rows = 2, cols = 3, lda = 3;
    std::array<double, 6> a{};
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            a[i*lda+j] = i*cols+j+1;
        }
    }
    std::cout << "row_major =";
```

```
make bin/02_row_major_storage
./bin/02_row_major_storage
```

Contiguous rows	1 2 3 4 5 6
With stride five	1 2 3 −99 −99 4 5 6 −99 −99

Read one address before reading a whole matrix

The full file prints all six logical entries and offsets, then checks both layout and untouched padding. For the official square matrix without padding, use `lda=n`.

Lower and upper triangular systems (I)

Lower: solve from the top

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 3 & 1 \end{pmatrix}.$$

Entries above the diagonal vanish. The first equation contains only y_1.

Upper: solve from the bottom

$$U = \begin{pmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix}.$$

Entries below the diagonal vanish. The last equation contains only x_3.

The diagonal matters

These triangular systems have unique solutions because their diagonal entries are nonzero. In this LU convention, the diagonal of L is fixed to one.

Lower and upper triangular systems (II)

How do L and U come from A?

For our example, the original system is

$$A\mathbf{x} = \mathbf{b}, \qquad A = \begin{pmatrix} 2 & 1 & 1 \\ 4 & 3 & 3 \\ 8 & 7 & 9 \end{pmatrix}, \qquad \mathbf{b} = \begin{pmatrix} 4 \\ 10 \\ 24 \end{pmatrix}.$$

Gaussian elimination factorizes the matrix A as

$$A = LU = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 3 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix}.$$

Therefore, $A\mathbf{x} = \mathbf{b} \implies LU\mathbf{x} = \mathbf{b}$.
Now define $U\mathbf{x} = \mathbf{y}$.
Then we solve the problem in two easier steps:

$$L\mathbf{y} = \mathbf{b}, \qquad U\mathbf{x} = \mathbf{y}.$$

For a unit lower triangular matrix,

$$y_i = b_i - \sum_{j=1}^{i-1} L_{ij} y_j.$$

With our L and $\boldsymbol{b} = (4, 10, 24)^T$, work downward:

$$\begin{aligned} y_1 &= 4, \\ 2y_1 + y_2 = 10 \quad &\Rightarrow \quad y_2 = 10 - 2(4) = 2, \\ 4y_1 + 3y_2 + y_3 = 24 \quad &\Rightarrow \quad y_3 = 24 - 4(4) - 3(2) = 2. \end{aligned}$$

$$\boxed{\boldsymbol{y} = (4, 2, 2)^T}$$

Why this loop order works

When calculating y_i, all y_j with $j < i$ are already known. For a non-unit lower diagonal, divide the remaining sum by L_{ii}. E04 shows this general division.

For an upper triangular system,

$$x_i = \frac{y_i - \sum_{j=i+1}^{n} U_{ij} x_j}{U_{ii}}.$$

Use $\boldsymbol{y} = (4, 2, 2)^T$ and work upward:

$$\begin{aligned} x_3 &= 2/2 = 1, \\ x_2 &= (2 - 1 \cdot x_3)/1 = (2 - 1)/1 = 1, \\ x_1 &= (4 - 1 \cdot x_2 - 1 \cdot x_3)/2 = (4 - 1 - 1)/2 = 1. \end{aligned}$$

$$\boxed{\boldsymbol{x} = (1, 1, 1)^T}, \qquad A\boldsymbol{x} = (4, 10, 24)^T.$$

C++ detail

Use a signed loop index when counting backward to zero. An unsigned index does not become negative; it wraps to a very large value.

E04 | Triangular substitution in C++

Excerpt: 04_triangular_substitution.cpp, lines 16–29. Full program is in examples/.

```
    // Forward substitution: earlier y values are already available.
    for (int i = 0; i < n; ++i) {
        double sum = b[i];
        for (int j = 0; j < i; ++j) sum -= L[i*n+j]*y[j];
        if (L[i*n+i] == 0) return 1;
        y[i] = sum / L[i*n+i];
    }
    // Use a signed index: an unsigned i would underflow below zero.
    for (int i = n-1; i >= 0; --i) {
        double sum = y[i];
        for (int j = i+1; j < n; ++j) sum -= U[i*n+j]*x[j];
        if (U[i*n+i] == 0) return 1;
        x[i] = sum / U[i*n+i];
    }
```

```
make bin/04_triangular_substitution
./bin/04_triangular_substitution
```

Expected: `y = 4 2 2`, `x = 1 1 1`. The complete independent file declares the arrays and checks $L\boldsymbol{y} = \boldsymbol{b}$ and $U\boldsymbol{x} = \boldsymbol{y}$.

The elimination step produced U. Keep its multipliers in a lower triangular matrix L:

$$A = LU.$$

Then the same original system can be written

$$L\underbrace{(U\boldsymbol{x})}_{\boldsymbol{y}} = \boldsymbol{b} \quad \Longrightarrow \quad L\boldsymbol{y} = \boldsymbol{b}, \quad U\boldsymbol{x} = \boldsymbol{y}.$$

Separate factorization from solution

Compute the factors from A once. Each new right-hand side requires only forward and backward substitution. Do not explicitly calculate A^{-1} just to solve $A\boldsymbol{x} = \boldsymbol{b}$.

This unpivoted form is suitable for the current teaching matrix. A general library factorization also records row permutations.

Construct L and U from the recorded multipliers

The row operations used $m_{21} = 2$, $m_{31} = 4$, and $m_{32} = 3$. Place these values below the unit diagonal of L:

$$L = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 3 & 1 \end{pmatrix}, \qquad U = \begin{pmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix}.$$

Where the signs go

Elimination subtracts $m_{ik}R_k$. The lower factor stores the multiplier m_{ik} itself, not its negative. Multiplication by L reconstructs the rows before elimination.

For this example $A = LU$ exactly. E03 produces U from row operations; E04 uses these same triangular factors.

Verify the factorization by multiplication

Use row–column products; do not multiply entries element by element.

$$LU = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 4 & 3 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix}.$$

Every reconstructed row is a combination of rows of U:

$$\begin{aligned} (LU)_{1,:} &= [2, 1, 1], \\ (LU)_{2,:} &= 2[2, 1, 1] + [0, 1, 1] = [4, 3, 3], \\ (LU)_{3,:} &= 4[2, 1, 1] + 3[0, 1, 1] + [0, 0, 2] = [8, 7, 9]. \end{aligned}$$

A complete check

All nine entries agree with the original A. Checking only the diagonal would miss many indexing errors. The LAPACKE demonstration E06 similarly checks every entry of $PA - LU$.

Elimination multipliers are reusable information

Operation	Multiplier	Stored location
$R_2 \leftarrow R_2 - 2R_1$	2	L_{21}
$R_3 \leftarrow R_3 - 4R_1$	4	L_{31}
$R_3 \leftarrow R_3 - 3R_2$	3	L_{32}

The last multiplier uses the **updated** second row. It is not obtained by dividing two unrelated entries of the original matrix.

Connect the two programs

E03 modifies both A and $\boldsymbol{b}$ during elimination. E04 instead starts with the saved factors and reproduces the transformed right-hand side through $L\boldsymbol{y} = \boldsymbol{b}$.

In a general pivoted factorization, later swaps also affect previously stored multiplier rows. Let the library manage that bookkeeping.

Why pivoting becomes necessary

$$A = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}, \qquad \det(A) = -2 \neq 0.$$

This matrix is nonsingular, but the first unpivoted multiplier would divide by zero. A failed unpivoted step does not prove that the system has no solution.

Partial pivoting

In active column k, choose a row $p \geq k$ with the largest $|A_{pk}|$ and interchange rows k and p. Only rows are exchanged; columns and the order of unknowns stay fixed.

What pivoting does not do

It avoids a zero pivot when a suitable row exists and limits elimination multipliers. It does not turn an inherently ill-conditioned problem into a well-conditioned one.

For $\boldsymbol{b} = (2, 8)^T$, swap the two complete augmented rows:

$$\left[\begin{array}{cc|c} 0 & 1 & 2 \\ 2 & 3 & 8 \end{array}\right] \longrightarrow \left[\begin{array}{cc|c} 2 & 3 & 8 \\ 0 & 1 & 2 \end{array}\right].$$

The transformed system is already triangular:

$$x_2 = 2, \qquad x_1 = \frac{8 - 3(2)}{2} = 1.$$

```
make bin/05_partial_pivoting
./bin/05_partial_pivoting
```

Do not swap the unknowns

Row swaps reorder equations, not variables. The full E05 program selects the pivot row, swaps $\boldsymbol{b}$ consistently, and reports one swap, $\det(A) = -2$, and a zero residual.

Let P apply the row swaps in the order used by elimination. We use

$$\boxed{PA = LU}, \qquad P = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \text{ for E05.}$$

Since $A\boldsymbol{x} = \boldsymbol{b}$, multiplying the whole system by P gives

$$LU\boldsymbol{x} = P\boldsymbol{b} \quad \Longrightarrow \quad L\boldsymbol{y} = P\boldsymbol{b}, \quad U\boldsymbol{x} = \boldsymbol{y}.$$

The library performs the solve permutations

Pass the original $\boldsymbol{b}$ and the returned `ipiv` to `dgetrs`. Do not manually permute $\boldsymbol{b}$ first, or the swaps would be applied twice.

Netlib writes $A = P_N LU$; this is equivalent with $P = P_N^{-1}$. A single swap is self-inverse, but a sequence of swaps need not be.

Pivot data are row-swap instructions

For E06, the library returns `ipiv = 2 2`. The values are **one-based**, even though they are accessed from C++.

C++ index	Value	Instruction
0	2	Swap logical row 0 with row $2 - 1 = 1$.
1	2	Swap row 1 with row $2 - 1 = 1$: no change.

A swap log is not a final row ordering

Apply the swaps sequentially to reconstruct PA. Count an interchange when `ipiv[i] != i+1`; do not count both entries as swaps.

For a manual reconstruction, convert each pivot with `p=ipiv[i]-1`. The call to `dgetrs` expects the unmodified, one-based array.

Determinants of triangular matrices

A triangular determinant is the product of its diagonal entries:

$$\det(U) = \prod_{i=1}^{n} U_{ii}.$$

For the unpivoted 3×3 example,

$$\det(U) = 2 \cdot 1 \cdot 2 = 4, \qquad \det(L) = 1 \cdot 1 \cdot 1 = 1.$$

Therefore $A = LU$ implies $\det(A) = 4$.

Why the shortcut works

In the determinant expansion, triangular zeros remove all products except the product of diagonal entries. A unit lower triangular factor contributes only a factor of one.

For nonsingularity, the exact zero/nonzero distinction matters. The numerical magnitude of a determinant is not a measure of conditioning.

A row swap changes the determinant sign

With $PA = LU$ and s interchanges,

$$\det(P)\det(A) = \det(L)\det(U), \qquad \det(P) = (-1)^s.$$

Since $\det(L) = 1$ and $(-1)^{-s} = (-1)^s$,

$$\boxed{\det(A) = (-1)^s \prod_i U_{ii}}.$$

In E05, $U = \begin{pmatrix} 2 & 3 \\ 0 & 1 \end{pmatrix}$ and $s = 1$:

$$\det(A) = (-1)\,(2)(1) = -2.$$

Correction to the handout formula

The diagonal product alone applies when no row interchanges occur. The required `dgetrf` uses partial pivoting, so include the permutation sign.

BLAS, LAPACK, and LAPACKE have different roles

BLAS	Basic vector and matrix kernels used inside many algorithms.
LAPACK	Factorizations and solvers built from numerical kernels.
LAPACKE	A C interface that lets C++ call LAPACK with explicit storage layout.

Two calls, two distinct jobs

`LAPACKE_dgetrf`: factor the matrix and record pivots.
`LAPACKE_dgetrs`: solve using those factors and pivots.

The prefix `d` denotes real double precision. The laboratory requires these calls separately so that you can examine the LU factors and determinant.

Include <lapacke.h>. Compile one source at a time from the package root; a standard system installation commonly uses:

```
g++ -std=c++17 -Wall -Wextra -Wpedantic -O2 \
  examples/07_lapacke_solve.cpp \
  -o bin/07_lapacke_solve -llapacke -llapack -lblas -lm
./bin/07_lapacke_solve
```

`-I/path/include`	Search here for header declarations.
`-L/path/lib`	Search here for libraries during linking.
`-lname`	Link library `libname`.

Check your actual installation

Missing `lapacke.h` is an include problem; an undefined reference is a linking problem. A shared library must also be found at runtime. See the README for installation notes and checks.

Row-major dimensions | A and b are not alike

Array	Logical shape	Physical stride	E07 value
`lu`	$n \times n$	`lda` $\geq n$	2
`x` (initially b)	$n \times$ `nrhs`	`ldb` $\geq$ `nrhs`	1
`ipiv`	n entries	`lapack_int`	length 2

For one right-hand side, `nrhs=1`. Row-major storage for B has one entry in each row, hence `ldb=1`, not `n`.

Use the integer type provided by the header

Use `lapack_int` for dimensions and pivots. Its width must match the linked library; do not mix incompatible LP64 and ILP64 interfaces.

For multiple right-hand sides, B_{ir} is stored as `b[i*ldb+r]`. The factors in `lu` are reused.

LAPACKE_dgetrf | Factor the matrix



```
lapack_int LAPACKE_dgetrf(int matrix_layout,
    lapack_int m, lapack_int n, double* a,
    lapack_int lda, lapack_int* ipiv);
```

On input, a contains the original matrix. On output, the same array contains the factors: strictly below the diagonal lies L; on and above it lies U. The ones on the diagonal of L are implicit.

Input, output, status

`m`,`n`: dimensions. `lda`: physical stride.
`ipiv`: row-swap instructions. The returned `info` reports completion or an error.

The real-double array type is `double*`. Preserve the original matrix before this call whenever you will verify $A\boldsymbol{x} = \boldsymbol{b}$.

E06 | Factorization code, part 1

Excerpt – setup, call, status: `06_lapacke_lu_factorization.cpp`, lines 14–25. Full program is in `examples/`.

```
const lapack_int n = 2, lda = 2;
const std::array<double, 4> A_original = {1,2, 3,4};
auto lu = A_original;
std::array<lapack_int, 2> ipiv{};
const lapack_int info = LAPACKE_dgetrf(
    LAPACK_ROW_MAJOR, n, n, lu.data(), lda, ipiv.data());
std::cout << "info = " << info << '\n';
if (info != 0) {
    std::cerr << "E06: factorization failed; inspect info above.\n";
    return 1;
}
std::array<double, 4> L{}, U{};
```

The complete program includes `<lapacke.h>` and standard headers. Its array sizes and leading dimension correspond to the 2×2 matrix $\left(\begin{smallmatrix} 1 & 2 \\ 3 & 4 \end{smallmatrix}\right)$. It does not solve for a right-hand side yet.

E06 | Factorization code, part 2

Excerpt – apply pivots and unpack factors: `06_lapacke_lu_factorization.cpp`, lines 26–36. Full program is in `examples/`.

```
    auto PA = A_original;
    for (lapack_int i = 0; i < n; ++i) {
        const lapack_int p = ipiv[i]-1; // LAPACK values are ONE-based.
        if (p < i || p >= n) return 1;
        for (lapack_int j = 0; j < n; ++j)
            std::swap(PA[i*n+j], PA[p*n+j]);
        for (lapack_int j = 0; j < n; ++j) {
            L[i*n+j] = (i > j) ? lu[i*n+j] : (i == j ? 1.0 : 0.0);
            U[i*n+j] = (i <= j) ? lu[i*n+j] : 0.0;
        }
    }
```

```
make bin/06_lapacke_lu_factorization
./bin/06_lapacke_lu_factorization
```

Expected packed rows: $(3, 4)$ and $(1/3, 2/3)$; `ipiv = 2 2`. The complete source checks every entry of $PA - LU$ before printing `PASS E06`.

Reconstruct L, U, and P from the packed array

E06 returns, to rounding accuracy,

$$\texttt{lu} \longleftrightarrow \begin{pmatrix} 3 & 4 \\ 1/3 & 2/3 \end{pmatrix}.$$

Extract the two factors and apply the recorded swap:

$$L = \begin{pmatrix} 1 & 0 \\ 1/3 & 1 \end{pmatrix}, \quad U = \begin{pmatrix} 3 & 4 \\ 0 & 2/3 \end{pmatrix}, \quad PA = \begin{pmatrix} 3 & 4 \\ 1 & 2 \end{pmatrix}.$$

The lower row of LU is

$$\tfrac{1}{3}(3,4) + (0, \tfrac{2}{3}) = (1, \tfrac{4}{3} + \tfrac{2}{3}) = (1, 2).$$

Read the diagonal correctly

`lu[0]` and `lu[3]` belong to U, not L. The lower-factor diagonal is reconstructed as $(1, 1)$; it is not stored in the packed array.

LAPACKE_dgetrs | Solve using the factors



```
lapack_int LAPACKE_dgetrs(int matrix_layout, char trans,
    lapack_int n, lapack_int nrhs, const double* a,
    lapack_int lda, const lapack_int* ipiv,
    double* b, lapack_int ldb);
```

Pass the **factored** matrix and its pivot array. With `trans='N'`, solve the original system $A\boldsymbol{x} = \boldsymbol{b}$ without transposing it.

What changes?

The factors and pivots are read, not recomputed. The right-hand-side array is overwritten with the solution. An unchanged A can therefore be reused for another right-hand side.

Do not call `dgetrs` on the unfactorized matrix. Do not modify `lu` or `ipiv` between the factorization and solve calls.

E07 | Solve code, part 1

Excerpt – preserve inputs and factor once: `07_lapacke_solve.cpp`, lines 14–25. Full program is in `examples/`.

```
const lapack_int n = 2, nrhs = 1, lda = 2, ldb = 1;
const std::array<double, 4> A_original = {1,2, 3,4};
const std::array<double, 2> b_original = {5,11};
auto lu = A_original;
auto x = b_original; // x initially contains b; dgetrs overwrites it.
std::array<lapack_int, 2> ipiv{};
const lapack_int factor_info = LAPACKE_dgetrf(
    LAPACK_ROW_MAJOR, n, n, lu.data(), lda, ipiv.data());
if (factor_info != 0) {
    std::cerr << "E07: dgetrf info=" << factor_info << '\n';
    return 1; // Do not solve with failed or singular factors.
}
```

We set $\boldsymbol{b} = (5, 11)^T$, so the exact solution is $(1, 2)^T$. The arrays `A_original` and `b_original` remain available for independent verification.

E07 | Solve code, part 2

Excerpt – solve and handle status: `07_lapacke_solve.cpp`, lines 26–32. Full program is in `examples/`.

```
    const lapack_int solve_info = LAPACKE_dgetrs(
        LAPACK_ROW_MAJOR, 'N', n, nrhs, lu.data(), lda,
        ipiv.data(), x.data(), ldb);
    if (solve_info != 0) {
        std::cerr << "E07: dgetrs info=" << solve_info << '\n';
        return 1;
    }
```

```
make bin/07_lapacke_solve
./bin/07_lapacke_solve
```

Expected result

`factor_info = 0`, `solve_info = 0`,
`x = 1 2`, and a residual close to zero.

The remaining source verifies both solution components and computes $\boldsymbol{b}_{\text{original}} - A_{\text{original}}\boldsymbol{x}$. The direct compiler command is on S30 and in the README.

Why preserve the original matrix and right-hand side?

Object	Before the calls	After the calls
`A_original`	Original A	Still A; never passed for overwrite.
`lu`	Copy of A	Packed L and U.
`b_original`	Original $\boldsymbol{b}$	Still $\boldsymbol{b}$.
`x`	Copy of $\boldsymbol{b}$	Computed solution $\widehat{\boldsymbol{x}}$.

A plausible-looking but incorrect residual

Computing `lu*x` treats packed factors as the original matrix. Comparing with overwritten `x` treats the solution as the right-hand side. Neither checks the problem you intended to solve.

Correct check: $\boldsymbol{r} = \boldsymbol{b}_{\text{original}} - A_{\text{original}}\widehat{\boldsymbol{x}}$.

Residuals | Quantify how well the equations hold

Define the residual using the original data:

$$\boldsymbol{r} = \boldsymbol{b} - A\widehat{\boldsymbol{x}}, \quad \|\boldsymbol{r}\|_\infty = \max_i |r_i|, \quad \|A\|_\infty = \max_i \sum_j |A_{ij}|.$$

A useful scale-aware measure is

$$\eta = \frac{\|\boldsymbol{r}\|_\infty}{\|A\|_\infty \|\widehat{\boldsymbol{x}}\|_\infty + \|\boldsymbol{b}\|_\infty}.$$

For E07, the exact result gives $\boldsymbol{r} = (0,0)^T$ and denominator $7 \cdot 2 + 11 = 25$, so $\eta = 0$ in exact arithmetic.

Residual is not forward error

A small residual means the computed vector nearly satisfies the equations. If A is ill-conditioned, that alone does not guarantee closeness to the exact solution. Handle a zero denominator separately.

E08 | Determinant parity and the residual

Excerpt – determinant after successful factorization: `08_determinant_and_residual.cpp`, lines 23–30. Full program is in `examples/`.

```
int swaps = 0;
double product = 1;
for (lapack_int i = 0; i < n; ++i) {
    if (ipiv[i] < i+1 || ipiv[i] > n) return 1;
    if (ipiv[i] != i+1) ++swaps; // Compare against one-based row i+1.
    product *= lu[i*n+i];
}
const double det = (swaps % 2 ? -1.0 : 1.0) * product;
```

```
make bin/08_determinant_and_residual
./bin/08_determinant_and_residual
```

Expected: one swap, diagonal product $+2$, corrected determinant -2, and a scaled residual near zero. E08 also solves the system and evaluates the original equations.

Tolerances: These checks use a small known system. Interpret larger or ill-conditioned systems using their scales; extra printed digits do not improve accuracy.

Read the status; then assess numerical reliability

`dgetrf` result	Interpretation
`info = 0`	Factorization completed; conditioning is not certified.
`info > 0`	A diagonal entry of U was exactly zero; do not call the solve.
`info < 0`	Argument/input error, or an interface-specific failure; inspect the message and documentation.

The solve is not a singularity detector

`dgetrs` assumes valid factors. Check its status too, but also check finite outputs and residuals. The basic `dgetrf`+`dgetrs` pair does not perform iterative refinement or estimate a condition number.

A mathematically singular problem may appear to have a tiny nonzero pivot after rounding. Numerical status does not replace a mathematical consistency check.

Why shape a polynomial?

A shaping function is chosen to pass through specified values and to have specified slopes. In a numerical approximation, these are local conditions at **collocation nodes**.

$$q(x) = c_0 + c_1 x + c_2 x^2, \qquad q'(x) = c_1 + 2c_2 x.$$

Value and slope contain different information

A value fixes a point on the curve. A derivative fixes its tangent there. Both produce equations that are linear in the coefficients, even though the curve is nonlinear in x.

A smaller teaching example

The toy $q(x) = 1 + x - x^2$ is used for file output and plotting. The assessed task uses six unknown coefficients and its own conditions.

A fifth-degree polynomial has six coefficients

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 + a_5 x^5.$$
$$f'(x) = a_1 + 2a_2 x + 3a_3 x^2 + 4a_4 x^3 + 5a_5 x^4.$$

The unknown vector is

$$\boldsymbol{a} = (a_0, a_1, a_2, a_3, a_4, a_5)^T.$$

Count coefficients, not just the highest power

The constant term a_0 is also unknown. Four function values plus two slopes give six equations for these six coefficients.

We work in the space of polynomials of degree at most five; a coefficient can in principle vanish. Uniqueness requires six independent constraints, not merely six rows.

Values and derivatives become matrix rows

Value row: $[1, t, t^2, \ldots, t^5]$,
Slope row: $[0, 1, 2t, 3t^2, 4t^3, 5t^4]$.

Unrelated quadratic toy: solve the row-building idea

For $q = c_0 + c_1 x + c_2 x^2$, prescribe $q(0) = 1$, $q'(0) = 1$, and $q(2) = -1$:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 2 & 4 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ -1 \end{pmatrix}.$$

The first two rows give $c_0 = c_1 = 1$; then $1 + 2 + 4c_2 = -1$ gives $c_2 = -1$.

For derivative rows, the constant-column entry is explicitly zero. Do not evaluate a formula containing t^{-1} at $t = 0$.

The official polynomial conditions

Node	Position	Function value	First derivative
x_0	0	$f(0) = 0$	$f'(0) = 1$
x_1	1	$f(1) = 1$	Not prescribed
x_2	2	$f(2) = 0$	Not prescribed
x_3	3	$f(3) = -1$	$f'(3) = 0$

Keep every prescribed condition

There are four nodes, but six equations. Do not add zero-slope conditions at the interior nodes: those derivatives are not specified.

The handout orders the rows as: value at x_0, slope at x_0, value at x_1, value at x_2, value at x_3, slope at x_3.

Structure of the assigned linear system

Define two row functions:

$$v(t) = [1, t, t^2, t^3, t^4, t^5], \qquad d(t) = [0, 1, 2t, 3t^2, 4t^3, 5t^4].$$

Then the handout's matrix structure is

$$\underbrace{\begin{pmatrix} v(x_0) \\ d(x_0) \\ v(x_1) \\ v(x_2) \\ v(x_3) \\ d(x_3) \end{pmatrix}}_{A:\,6\times 6} \underbrace{\begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \end{pmatrix}}_{\boldsymbol{a}} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \\ -1 \\ 0 \end{pmatrix}.$$

Each displayed v or d is a six-entry row. Implement these row types yourself and check the resulting matrix **before** calling the factorization.

T01	Implement the handout's system; use `dgetrf` followed by `dgetrs`.
T02	Use $x_0 = 0$, $x_1 = 1$, $x_2 = 2$, $x_3 = 3$ for the initial calculation.
T03	Print the original matrix and check its entries.
T04	Calculate the determinant from U, **including row-swap parity**.

A practical execution order

Construct and print A first; preserve inputs; factor; check status and determinant; then solve. The handout lists obligations, not a reason to inspect an already overwritten matrix.

Use E02 for storage, E06/E07 for the library calls, and E08 for the determinant and residual. Their toy matrices are not your project matrix.

T05–T06 | Save coefficients, then plot

T05: Find the coefficients by solving the system and write them to a file. Preserve their order $a_0, a_1, \ldots, a_5$ and enough digits to reproduce the calculation.

T06: Use your calculated coefficients to plot $f(x)$. Sample on a sufficiently fine grid across the node interval; label the axes and compare the curve with the prescribed points.

A file is evidence, not an intermediate screenshot

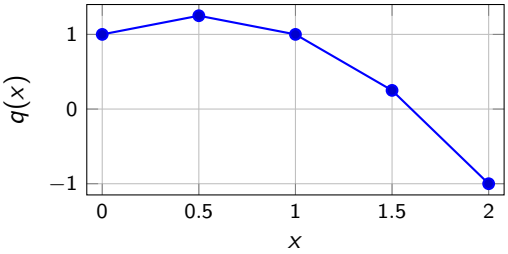
Confirm that the values plotted were read from your own program output. A hard-coded reference curve would not verify the program you wrote.

E09 teaches saving sampled toy data. Its plot helper accepts exactly five toy rows; adapt or write a separate reader for your project. File schemas and derivative checks are suggested verification aids, not additional assessed tasks.

E09 | Actual CSV data become a plot

For the toy $q(x) = 1 + x - x^2$, E09 writes:

x	$q(x)$
0	1
0.5	1.25
1	1
1.5	0.25
2	−1



Five recorded C++ samples; straight-line joins, not an exact continuous curve.

Header: `x,y`. Existing outputs are protected.

```
./bin/09_toy_polynomial_output data/toy_run01.csv
python3 plotting/plot_toy_polynomial.py data/toy_run01.csv \
  --output plots/toy_run01.png
```

At home, move only x_2 toward $x_1 = 1$:

$$x_2 = 1.1, \quad 1.01, \quad 1.001, \quad 1.0001.$$

For every case, calculate the determinant and solve the system. Predict what coincident nodes imply, then test that prediction with your program.

Two different effects

Nearby nodes carry different prescribed values. The exact polynomial can change sharply, while rounding errors can be amplified. Large coefficients are not automatically a coding mistake.

The coincident-node question

At coincidence, the conditions would require both $f(1) = 1$ and $f(1) = 0$. Explain the inconsistency, and do not treat a numerical return code as the entire mathematical argument.

A separate condition-number calculation is optional, not an added assessed requirement.

First checkpoint	Explain a value row and a derivative row; print the matrix before it is overwritten.
Second checkpoint	Preserve inputs, factor, inspect status and pivots, correct the determinant sign, solve, and evaluate the residual.
Third checkpoint	Save coefficients, generate your own samples, plot them, and check values and slopes at the prescribed nodes.

The next 70 minutes belong to your implementation

Use the finished examples as references, not as substitutes for your own matrix construction, integration, output, and interpretation. Ask for help at the first stage whose output you cannot explain.

Debug the first incorrect result

Symptom	First question to ask
Matrix entries misplaced	Is the offset `i*lda+j`, with rows and columns in the intended order?
Wrong slope row	Was the constant term set to zero and each power differentiated?
Unexpected determinant sign	Were actual swaps counted using `ipiv[i] != i+1`?
Bad solution / illegal layout	Were the factors passed unchanged, with row-major `ldb=1`?
Residual makes no sense	Are you using original A and $\boldsymbol{b}$, not packed factors or overwritten RHS?
Plot disagrees with results	Did the reader load the newest file, in the right coefficient order?

Ask for an expected *small* result before changing many parts of the program.

Validation, report evidence, and sources

In your report, distinguish the nominal problem, the prescribed node-shift experiment, and the coincident-node interpretation. Keep original numerical outputs and explain the behavior you observe.

Before finishing

Can you explain what is stored in `lu`, why the determinant needs a sign, and why a small residual is not a universal accuracy guarantee? Can you trace the plotted values back to your program?

[H] T. Chwiej, Polynomial-shaping laboratory, 12 March 2026, pp. 1–4.

[L] T. Chwiej, Direct methods lecture, especially pp. 9–13, 17–18, 22–25, 32–35.

[N] Netlib LAPACKE interface and DGETRF documentation; consult the installed header for exact declarations.

Follow the course's separately communicated submission arrangements; this package does not invent a deadline or grading scheme.

Optional E10 | Sensitivity in a 2-by-2 system

$$A_\varepsilon = \begin{pmatrix} 1 & 1 \\ 1 & 1+\varepsilon \end{pmatrix}, \quad \boldsymbol{b} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad \boldsymbol{x} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad (\varepsilon > 0).$$

Perturb only b_2 by δ. Subtracting the equations gives

$$\varepsilon\, \widetilde{x}_2 = \delta \quad \Rightarrow \quad \widetilde{\boldsymbol{x}} = \begin{pmatrix} 1 - \delta/\varepsilon \\ \delta/\varepsilon \end{pmatrix}.$$

Thus a fixed small input change can produce a large solution change, even with a tiny residual for the perturbed system.

```
make optional
./bin/10_conditioning_experiment
```

E10 uses $\delta = 2^{-40}$ and decreasing powers of two for ε. It reports requested versus stored ε and skips the solve when rounding or $\varepsilon = 0$ makes the stored matrix singular.


THANK
YOU!