

Numerical Methods in Physics

Project 01

Simple Numerical Operations with Vectors and Matrices

Hamzeh Khanpour

hamzeh.khanpour@fis.agh.edu.pl

view.fis.agh.edu.pl/staff/hamzeh.khanpour/

Faculty of Physics and Applied Computer Science
AGH University of Krakow

October 9, 2026

The main goal of this laboratory



More than multiplying matrices

The goal of this laboratory is **not simply to multiply matrices**. It is to understand how mathematical operations become **numerical algorithms**, how their results behave under **repeated computation**, and how to implement them **accurately and efficiently in C++**.

Scientific computing in practice

You will gain your first practical experience with **scientific computing** by combining mathematical analysis, algorithm development, numerical experimentation, data visualization, and computational performance evaluation.

To do list ...

Matrix multiplication is our starting example, not the whole lesson. We will turn a mathematical rule into C++, observe what happens when the calculation is repeated, and use recorded results and plots to explain that behaviour. We will also examine correctness and computational efficiency. Emphasize that writing a program, understanding its output, and evaluating its performance are connected parts of scientific computing.

C02 | Why this laboratory?



Two questions

How do we turn vector and matrix formulas into C++?

How do we know that the results are reliable and the comparison is fair?

Scientific workflow

Question $\longrightarrow$ method $\longrightarrow$ implementation $\longrightarrow$ verification $\longrightarrow$ interpretation.

Course sources

S01: Tomasz Chwiej, Numerical Methods in Physics course page.

S02: Project 01 handout, 3 March 2026.

S03: Lecture 1, Numerical Errors.

Technical sources

Netlib BLAS and Mark Gates' CBLAS reference; GCC 14.2 manuals; C++ working-draft clock/type sections; N. J. Higham on backward error.

How we will work today



- 0–10 min** Opening and essential readiness.
- 10–40 min** Selected mathematics and finished C++ examples.
- 40–50 min** Project items, hints and checks.
- 50–120 min** **You write and test your own project.**
- 120–135 min** Interpret results and organize the report.

A lesson, then a laboratory

Code-continuation pages are reference material. We will not lecture through every page before you begin.

Two folders, two different purposes



Keep finished examples separate from your project

NMP_Project01_Worked_Examples/: compile and explore finished demonstrations.

~/nmp/project_work/Project01/: write your own assessed program.

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

Use the Ubuntu/Linux terminal

Use the Ubuntu/Linux terminal. If you are a Windows user, from PowerShell enter `ws1 ~`. Once the prompt ends in `$`, run Linux commands directly.

Only E07 needs CBLAS. The toy plotting helper needs Python 3 and Matplotlib.

Vectors, matrices and dimensions



Vector: ordered components

$$u = \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix}$$

Three entries; indices 0, 1, 2. Order matters.

Matrix: rows and columns

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

Two rows, three columns. $A_{1,2} = 6$ uses zero-based indices.

Check the shape of the answer

$u^T v$ is a **scalar**; Ax is a **vector**; AB is a **matrix**. The input dimensions must be compatible.

From a mathematical task to an algorithm

Task: what do we want?

Given two three-component vectors, calculate one scalar: $u^T v$.

Method: which mathematical operations?

Multiply corresponding components and add all products: $\sum_{i=0}^2 u_i v_i$.

Algorithm: how do we implement the method?

Initialize one sum, visit indices 0,1,2, add each product, then check the answer.
E01 makes these steps concrete.

E01 | Scalar product by hand



$$u = (1, 2, 3)^T, \quad v = (4, 5, 6)^T, \quad u^T v = \sum_{i=0}^2 u_i v_i.$$

Index i	0	1	2
u_i	1	2	3
v_i	4	5	6
Contribution $u_i v_i$	4	10	18

$$u^T v = 1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 32$$

From formula to a loop

Initialize the accumulator once before the loop. Add every contribution; do not keep only the final product.

Predict: what changes when every component of v is zero?

E01 | Compile, run and check



The complete small program computes and checks the scalar.

```
#include <cmath>
#include <iostream>
int main() {
    const double u[3] = {1, 2, 3};
    const double v[3] = {4, 5, 6};
    double sum = 0.0;
    for (int i = 0; i < 3; ++i) {
        sum += u[i] * v[i];
    }
    std::cout << "u dot v = " << sum << '\n';
    if (!std::isfinite(sum) || std::abs(sum - 32.0) > 1e-12) {
        std::cerr << "Dot-product check failed\n";
        return 1;
    }
    return 0;
}

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

Observed output: u dot v = 32.

E02 | Euclidean norm by hand

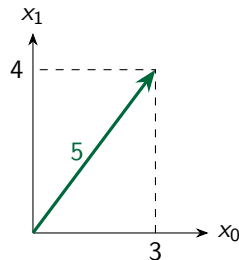


$$x = (3, 4)^T$$
$$x^T x = 3^2 + 4^2 = 25$$

$$\|x\|_2 = \sqrt{x^T x} = 5$$

Squared norm: 25.

Euclidean norm: 5.



A length cannot be negative

Changing $(3, 4)$ to $(-3, 4)$ leaves the squared terms unchanged. The zero vector has norm zero.

E02 | C++ norm and simple checks



`std::sqrt` converts the sum of squares into a length.

```
#include <cmath>
#include <iostream>
int main() {
    const double x[2] = {3, 4};
    double squared = 0.0;
    for (double value : x) {
        squared += value * value;
    }
    const double norm = std::sqrt(squared);
    std::cout << "squared norm = " << squared << '\n';
    std::cout << "norm = " << norm << '\n';
    if (!std::isfinite(norm) || std::abs(norm - 5.0) > 1e-12) {
        std::cerr << "Norm check failed\n";
        return 1;
    }
    return 0;
}

g++ -std=c++17 -Wall -Wextra -Wpedantic -O2 \
    examples/02_vector_norm.cpp -o bin/02_vector_norm
./bin/02_vector_norm
```

Observed: squared norm = 25; norm = 5.

E03 | One matrix, two storage orders



$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \quad (2 \text{ rows, } 3 \text{ columns})$$

Row-major

1	2	3	4	5	6
0	1	2	3	4	5

Column-major

1	4	2	5	3	6
0	1	2	3	4	5

The logical matrix has not been transposed

Row-major: $i \text{ lda} + j$, with $\text{lda} = 3$. Column-major: $i + j \text{ lda}$, with $\text{lda} = 2$. Small labels below cells are **flat offsets**.

E03 | Storage program, part 1



Part 1: create both representations of the same logical matrix.

```
#include <iostream>
#include <vector>
int main() {
    const int rows = 2, cols = 3;
    const int row_lda = cols, col_lda = rows;
    std::vector<double> row(rows * row_lda);
    std::vector<double> col(cols * col_lda);
    for (int i = 0; i < rows; ++i) {
        for (int j = 0; j < cols; ++j) {
            const double value = i * cols + j + 1;
            row[i * row_lda + j] = value;
            col[i + j * col_lda] = value;
        }
    }
    std::cout << "row-major:";
    for (double v : row) std::cout << ' ' << v;
    std::cout << "\ncolumn-major:";
    for (double v : col) std::cout << ' ' << v;
    std::cout << '\n';
}
```

Next: check address/value pairs. Padding is a separate optional segment.

E03 | Storage program, part 2



Part 2: verify every logical entry, not only a coincident last offset.

```
for (int i = 0; i < rows; ++i) {  
    for (int j = 0; j < cols; ++j) {  
        const int r = i * row_lda + j, c = i + j * col_lda;  
        const double expected = i * cols + j + 1;  
        std::cout << "A(" << i << ', ' << j << ")=" << row[r]  
                    << " row_offset=" << r << " col_offset=" << c << '\n';  
        if (row[r] != expected || col[c] != expected) return 1;  
    }  
}
```

row-major: 1 2 3 4 5 6

column-major: 1 4 2 5 3 6

A(0,1)=2: row_offset=1; col_offset=2

A(1,0)=4: row_offset=3; col_offset=1

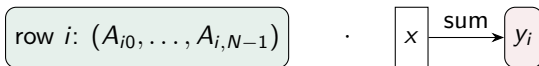
```
g++ -std=c++17 -Wall -Wextra -Wpedantic -O2 \  
    examples/03_matrix_storage.cpp -o bin/03_matrix_storage  
./bin/03_matrix_storage
```

The file continues with the optional padding segment, shown in the appendix, and returns success.

E04 | What does $y=Ax$ mean?



$$\underbrace{A}_{M \times N} \underbrace{x}_{N \times 1} = \underbrace{y}_{M \times 1}, \quad y_i = \sum_{j=0}^{N-1} A_{ij} x_j.$$



One row produces one output component

Each row has N entries to pair with the N vector components. There are M rows, so the answer has M components.

Implementation roles

i chooses the output row; j visits the contributions. Reset the sum for each new row.

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad x^{(0)} = \begin{pmatrix} 1 \\ 2 \end{pmatrix}$$

$$x^{(1)} = \begin{pmatrix} 1 \cdot 1 + 2 \cdot 2 \\ 3 \cdot 1 + 4 \cdot 2 \end{pmatrix} = \begin{pmatrix} 5 \\ 11 \end{pmatrix}$$

$$x^{(2)} = \begin{pmatrix} 1 \cdot 5 + 2 \cdot 11 \\ 3 \cdot 5 + 4 \cdot 11 \end{pmatrix} = \begin{pmatrix} 27 \\ 59 \end{pmatrix}$$

$$\|x^{(1)}\|_2 = \sqrt{146}, \quad \|x^{(2)}\|_2 = \sqrt{4210}.$$

Keep the entire old vector unchanged

Compute both new components before replacing the old vector. Otherwise a later row can mix two different states.

E04 | Matrix-vector code, part 1



Part 1: the returned vector is separate from the input.

```
#include <array>
#include <cmath>
#include <iomanip>
#include <iostream>
using Vec = std::array<double, 2>;
using Mat = std::array<double, 4>;
Vec matvec(const Mat& A, const Vec& x) {
    Vec y{}; // Separate result: do not overwrite x while reading it.
    for (int i = 0; i < 2; ++i) {
        double sum = 0.0;
        for (int j = 0; j < 2; ++j) {
            sum += A[i * 2 + j] * x[j];
        }
        y[i] = sum;
    }
    return y;
}
```

Handout clarification: p.3 defines a sum, but the pseudocode omits accumulation. Implement the defining sum.

E04 | Matrix-vector code, part 2



Part 2: apply the same operation twice and check both full answers.

```
int main() {
    const Mat A = {1, 2, 3, 4};
    const Vec x0 = {1, 2};
    const Vec x1 = matvec(A, x0);
    const Vec x2 = matvec(A, x1);
    const double n1 = std::sqrt(x1[0]*x1[0] + x1[1]*x1[1]);
    const double n2 = std::sqrt(x2[0]*x2[0] + x2[1]*x2[1]);
    std::cout << std::setprecision(12);
    std::cout << "x1 = " << x1[0] << ' ' << x1[1] << '\n';
    std::cout << "x2 = " << x2[0] << ' ' << x2[1] << '\n';
    std::cout << "norms = " << n1 << ' ' << n2 << '\n';
    if (x1 != Vec{5,11} || x2 != Vec{27,59} || x0 != Vec{1,2}) return 1;
    if (!std::isfinite(n1) || !std::isfinite(n2)) return 1;
    if (std::abs(n1 - std::sqrt(146.0)) > 1e-12 ||
        std::abs(n2 - std::sqrt(4210.0)) > 1e-12) return 1;
    return 0;
}
```

```
g++ -std=c++17 -Wall -Wextra -Wpedantic -O2 \
    examples/04_matrix_vector.cpp -o bin/04_matrix_vector
./bin/04_matrix_vector
```

Observed: $x^{(1)} = (5, 11)$; $x^{(2)} = (27, 59)$.

norms = 12.0830459736 64.88451279

$$\underbrace{A}_{M \times K} \underbrace{B}_{K \times N} = \underbrace{C}_{M \times N}, \quad C_{ij} = \sum_{\ell=0}^{K-1} A_{i\ell} B_{\ell j}.$$



Three loop roles

Choose the output row; choose the output column; accumulate the shared-index contributions. Each output element starts from a new sum.

K is the shared dimension here, not the number of repeated iterations.

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

$$C_{00} = 1 \cdot 5 + 2 \cdot 7 = 19,$$

$$C_{01} = 1 \cdot 6 + 2 \cdot 8 = 22,$$

$$C_{10} = 3 \cdot 5 + 4 \cdot 7 = 43,$$

$$C_{11} = 3 \cdot 6 + 4 \cdot 8 = 50.$$

$$AB = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

An independent check before large calculations

Your hand result checks every entry of the tiny C++ output. Reuse exactly these inputs for the CBLAS example.

E05 | A squared is not elementwise squaring



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

$$A^2 = AA = \begin{pmatrix} 1 \cdot 1 + 2 \cdot 3 & 1 \cdot 2 + 2 \cdot 4 \\ 3 \cdot 1 + 4 \cdot 3 & 3 \cdot 2 + 4 \cdot 4 \end{pmatrix} = \begin{pmatrix} 7 & 10 \\ 15 & 22 \end{pmatrix}.$$

Different operation

Elementwise squaring gives $\begin{pmatrix} 1^2 & 2^2 \\ 3^2 & 4^2 \end{pmatrix} = \begin{pmatrix} 1 & 4 \\ 9 & 16 \end{pmatrix}$, which is **not** A^2 .

E05 calculates and labels all three results: AB , AA and the elementwise contrast.

E05 | Matrix-product code, part 1



Part 1: row, column, sum; the inputs are read-only.

```
#include <array>
#include <iostream>
using Mat = std::array<double, 4>;
Mat multiply(const Mat& A, const Mat& B) {
    Mat C{};
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 2; ++j) {
            double sum = 0.0;
            for (int k = 0; k < 2; ++k) {
                sum += A[i * 2 + k] * B[k * 2 + j];
            }
            C[i * 2 + j] = sum;
        }
    }
    return C;
}
```

Reuse: pass A twice for A^2 . The returned matrix is new output storage.

E05 | Matrix-product code, part 2



Part 2: label the operations and check every entry.

```
void print(const char* label, const Mat& C) {
    std::cout << label << '\n';
    for (int i = 0; i < 2; ++i) {
        std::cout << C[i * 2] << ' ' << C[i * 2 + 1] << '\n';
    }
}

int main() {
    const Mat A = {1, 2, 3, 4}, B = {5, 6, 7, 8};
    const Mat AB = multiply(A, B);
    const Mat AA = multiply(A, A);
    Mat elementwise{};
    for (int i = 0; i < 4; ++i) elementwise[i] = A[i] * A[i];
    print("AB:", AB);
    print("A*A:", AA);
    print("Elementwise square (not A*A):", elementwise);
    if (AB != Mat{19,22,43,50} || AA != Mat{7,10,15,22}) return 1;
    if (elementwise != Mat{1,4,9,16}) return 1;
    if (A != Mat{1,2,3,4} || B != Mat{5,6,7,8}) return 1;
    return 0;
}

g++ -std=c++17 -Wall -Wextra -Wpedantic -O2 \
    examples/05_matrix_products.cpp -o bin/05_matrix_products
./bin/05_matrix_products
```

E06 | A tiny repeated-operation experiment



$$T = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}, \quad x^{(0)} = (1, 1)^T, \quad \text{three steps only.}$$

$$x^{(1)} = (2 \cdot 1 + 1 \cdot 1, 1 \cdot 1 + 2 \cdot 1)^T = (3, 3)^T.$$

Step k	Computed vector	Exact norm
1	$(3, 3)^T$	$3\sqrt{2}$
2	$(9, 9)^T$	$9\sqrt{2}$
3	$(27, 27)^T$	$27\sqrt{2}$

Why this is easy to check

Each row sums to 3, so $x^{(k)} = (3^k, 3^k)^T$. The program will calculate, then save these results; it will not write hard-coded norm values.

Toy demonstration: not the official gamma matrix or a complete project.

E06 | Calculate, then write a CSV — part 1



Part 1 of 3: all numerical helpers are in this one file.

```
#include <array>
#include <cmath>
#include <filesystem>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <limits>
#include <stdexcept>
using Vec = std::array<double, 2>;
Vec matvec(const std::array<double, 4>& T, const Vec& x) {
    Vec y{};
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 2; ++j) y[i] += T[i*2+j] * x[j];
    }
    return y;
}
```

Input stays unchanged; return the newly computed vector. No helper file is required.

E06 | Calculate, then write a CSV — part 2



Part 2 of 3: choose a fresh file and define fixed toy inputs.

```
int main(int argc, char* argv[]) {
    try {
        if (argc > 2) throw std::runtime_error("Use: program [output.csv]");
        const std::filesystem::path path =
            (argc == 2) ? argv[1] : "data/toy_run01.csv";
        if (std::filesystem::exists(path))
            throw std::runtime_error("Output exists; choose a new filename.");
        std::ofstream csv(path);
        if (!csv) throw std::runtime_error("Cannot open output; check parent.");
        const std::array<double, 4> T = {2, 1, 1, 2};
        Vec x = {1, 1};
        csv << "iteration,norm2\n";
        csv << std::setprecision(std::numeric_limits<double>::max_digits10);
        std::cout << std::setprecision(12);
```

```
mkdir -p bin data plots
```

The parent directory must exist. An existing output is refused, not silently overwritten.

E06 | Three updates and checked file output

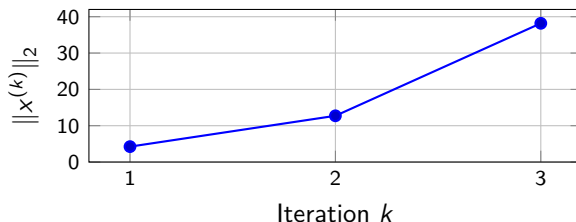


Part 3 of 3: compute first, save after the update, and check the write.

```
for (int k = 1; k <= 3; ++k) {
    const Vec next = matvec(T, x); // The whole old x is read first.
    x = next;
    const double norm = std::sqrt(x[0]*x[0] + x[1]*x[1]);
    const double expected = std::pow(3.0, k) * std::sqrt(2.0);
    if (!std::isfinite(norm) || std::abs(norm-expected) > 1e-12*expected)
        throw std::runtime_error("Toy calculation check failed.");
    csv << k << ',' << norm << '\n';
    std::cout << k << ": x=" << x[0] << ',' << x[1]
        << " norm=" << norm << '\n';
}
csv.close();
if (!csv) throw std::runtime_error("Output write/close failed.");
std::cout << "Saved: " << path.string() << '\n';
return 0;
} catch (const std::exception& e) {
    std::cerr << "Error: " << e.what() << '\n';
    return 1;
}
}
```

```
g++ -std=c++17 -Wall -Wextra -Wpedantic -O2 \
    examples/06_toy_iterations_csv.cpp -o bin/06_toy_iterations_csv
```

E06 | Plot only the data you actually created



```
./bin/06_toy_iterations_csv data/toy_run01.csv  
head -n 4 data/toy_run01.csv  
python3 plotting/plot_toy_norms.py data/toy_run01.csv \  
    --output plots/toy_run01
```

E06 only: $T = [2, 1; 1, 2]$, $x^{(0)} = (1, 1)^T$, three steps. Plot only after the producer succeeds.

BLAS is an interface to numerical kernels

Level	Operation	C interface
1	Dot product, norm	<code>cblas_ddot</code> , <code>cblas_dnrm2</code>
2	Matrix–vector product	<code>cblas_dgemv</code>
3	Matrix–matrix product	<code>cblas_dgemm</code>

BLAS and CBLAS

BLAS: Basic Linear Algebra Subprograms. **CBLAS**: the C interface, usable from C++. An installed implementation supplies the actual arithmetic routines.

The assigned library comparison

The handout explicitly requires GEMM for the matrix square. The earlier small standard-library examples teach the underlying loops; BLAS1/2 are additional available tools.

E07 | Read the small `cblas_dgemm` call



$$C \leftarrow \alpha \operatorname{op}(A) \operatorname{op}(B) + \beta C.$$

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,  
            2, 2, 2,          // M, N, K  
            1.0, A, 2, B, 2,  
            0.0, C, 2);      // beta=0 replaces old C
```

Read the parameters

Row-major storage; no transpose; $M = N = K = 2$; $\alpha = 1$; all leading dimensions are 2. With $\beta = 0$, the old output values are ignored.

Use the same known inputs as E05

$A = [1, 2; 3, 4]$, $B = [5, 6; 7, 8]$. The answer must be $AB = [19, 22; 43, 50]$, not the answer for A^2 .

E07 | Build, run and verify the library example



Part 1 of 2: a real library call with allocated, separate output.

```
#include <cblas.h>
#include <cmath>
#include <iostream>
int main() {
    const double A[4] = {1, 2, 3, 4};
    const double B[4] = {5, 6, 7, 8};
    double C[4] = {-99, -99, -99, -99};
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                2, 2, 2,          // M, N, K
                1.0, A, 2, B, 2,
                0.0, C, 2);      // beta=0 replaces old C

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

A header alone is not enough

The program must compile, link, run, and produce the expected matrix. Missing CBLAS leaves the library task pending.

E07 | Check the full CBLAS result



Part 2 of 2: inspect all four entries and reject non-finite values.

```
std::cout << "CBLAS AB:\n";
for (int i = 0; i < 2; ++i)
    std::cout << C[i*2] << ' ' << C[i*2+1] << '\n';
const double expected[4] = {19, 22, 43, 50};
for (int i = 0; i < 4; ++i) {
    if (!std::isfinite(C[i]) ||
        std::abs(C[i]-expected[i]) > 1e-12*(1+std::abs(expected[i]))) {
        std::cerr << "CBLAS entry check failed\n";
        return 1;
    }
}
std::cout << "All four entries checked\n";
return 0;
}
```

CBLAS AB:

19 22

43 50

All four entries checked

Think: why does changing only the layout flag not rearrange the stored array?

Timing: measure the calculation, not the printing

Outside the timer

Allocate and fill inputs. Afterward, check the result and print or save output.

```
const auto start = std::chrono::steady_clock::now();  
multiply(A, B, C, n);  
const auto stop = std::chrono::steady_clock::now();  
const double seconds = std::chrono::duration<double>(stop-start).count();
```

Inside the timer

Execute the selected multiplication. State exactly what the interval includes.

A clock is a tool, not a guarantee

`steady_clock` is monotonic. An extremely tiny operation may be too short to measure meaningfully; E08 uses a modest workload.

E08 | A modest timed calculation with a known answer



$$n = 96, \quad A_{ij} = i + j + 1 + t, \quad B_{ij} = 1.$$

$$C_{ij} = \sum_{\ell=0}^{n-1} (i + \ell + 1 + t)$$
$$= \boxed{n(i + 1 + t) + \frac{n(n-1)}{2}}.$$

An oracle separate from the multiplication

Every entry in one output row is equal. The program still uses a triple loop; the closed form checks its output, outside the timer.

Protocol: warm up at $t = 0$; time $t = 1, \dots, 5$. Inputs change between runs. This is not an official-size benchmark.

E08 | Timing program, part 1



Part 1 of 3: standard headers and the actual multiplication.

```
#include <algorithm>
#include <chrono>
#include <cmath>
#include <iomanip>
#include <iostream>
#include <vector>

void multiply(const std::vector<double>& A, const std::vector<double>& B,
             std::vector<double>& C, int n) {
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            double sum = 0.0;
            for (int k = 0; k < n; ++k) sum += A[i*n+k] * B[k*n+j];
            C[i*n+j] = sum;
        }
    }
}
```

The check formula is not used to produce C ; all products and sums are evaluated.

E08 | Timing program, part 2



Part 2 of 3: allocate once; fill outside the timer; start checking afterward.

```
int main() {
    const int n = 96;
    std::vector<double> A(n*n), B(n*n, 1.0), C(n*n);
    std::vector<double> times;
    std::cout << std::setprecision(12);
    for (int t = 0; t <= 5; ++t) { // t=0 is the unreported warm-up.
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j) A[i*n+j] = i+j+1+t;
        const auto start = std::chrono::steady_clock::now();
        multiply(A, B, C, n);
        const auto stop = std::chrono::steady_clock::now();
        const double seconds = std::chrono::duration<double>(stop-start).count();
        if (!std::isfinite(seconds) || seconds < 0) return 1;
        double checksum = 0.0;
        for (int i = 0; i < n; ++i) {
            const double expected = n*(i+1.0+t) + n*(n-1)/2.0;
            for (int j = 0; j < n; ++j) {
                const double value = C[i*n+j];
```

E08 | Validate, consume and report the timed result



Part 3 of 3: verify every entry and use the observed result.

```
        if (!std::isfinite(value) || std::abs(value-expected) > 1e-9)
            return 1;
        checksum += value;
    }
}

if (t > 0) {
    times.push_back(seconds);
    std::cout << "run=" << t << " seconds=" << seconds
                << " checksum=" << checksum << '\n';
}

}

std::sort(times.begin(), times.end());
std::cout << "median_seconds=" << times[2]
            << " min=" << times.front() << " max=" << times.back() << '\n';
std::cout << "All entries checked in warm-up and five timed runs\n";
return 0;
}

g++ -std=c++17 -Wall -Wextra -Wpedantic -O2 \
    examples/08_timing_small_example.cpp -o bin/08_timing_small_example
./bin/08_timing_small_example
```

E08 | Recorded timing demonstration



Run	Time / ms	Observed checksum
1	0.516140	85,819,392
2	1.838659	86,704,128
3	0.576311	87,588,864
4	0.517141	88,473,600
5	0.505945	89,358,336

Recorded output, not a prediction

Median 0.517141 ms; range 0.505945–1.838659 ms. All entries passed the independent-formula check in all six runs.

Local Linux container, GCC 14.2.0, -O2, fixed $n = 96$. These timings include only the multiplication. Your machine may differ; the spread is not a confidence interval.

Check answers before comparing speed



$$D_{ij} = |C_{ij}^{\text{manual}} - C_{ij}^{\text{BLAS}}|.$$

$$D_{ij} \leq \text{atol} + \text{rtol} \max(|C_{ij}^{\text{manual}}|, |C_{ij}^{\text{BLAS}}|).$$

A scale-aware check

Reject non-finite values first. Compare every entry, not only one corner or a total checksum. Use an absolute allowance near zero.

Two complementary sources of evidence

Small hand-worked answers check the calculation independently. Agreement between your code and a library is useful, but is not proof of exact accuracy.

For these small fixtures: E07 uses $10^{-12}(1 + |C_{ij}^{\text{exact}}|)$ as an explicitly chosen tolerance, not a universal project bound.

Finite precision: enough to know for this lab

Finite storage and rounding

Only finitely many numbers can be stored in a floating-point type. Matrix entries, products and accumulated sums can be rounded. We use `double` as the baseline, not as a guarantee of perfect answers.

Three different issues

A programming error changes the intended operation. Roundoff comes from finite arithmetic. A growing mathematical answer can be entirely correct.

What is not central here

Taylor-series and finite-difference truncation appear elsewhere in the lecture. The prescribed thirty products here define a finite task; they are not automatically a truncation error.

Epsilon describes spacing near 1 for the type; it is not a universal absolute error bar. Optional E09 illustrates representation.

Growth is not automatically a numerical failure



$$T = \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix}, \quad x^{(k)} = (3^k, 3^k)^T.$$

Growth exists in exact arithmetic

For E06, $\|x^{(k)}\|_2 = 3^k\sqrt{2}$. A rising graph is therefore expected even before rounding is considered.

Ask a numerical question

Does the computed result agree with the intended calculation within a justified tolerance? Large output alone cannot answer that.

Instructor clarification

The lecture discusses bounded output. Here we distinguish bounded iterates from algorithmic sensitivity to rounding. More precision does not repair incorrect indices.

Your project: inputs and required outputs



$$A_{ij} = \frac{1}{|i-j|^4 + \gamma}, \quad i, j = 0, \dots, 1999, \quad n = \text{lda} = 2000.$$
$$x_i^{(0)} = 1, \quad \gamma \in \{2, 3\}, \quad x^{(k+1)} = Ax^{(k)}.$$

- P01** Construct the matrix in one-dimensional storage.
- P02** Perform thirty unnormalized updates for each gamma.
- P03** Save one norm per update; make two separate plots.
- P04** Write your own matrix-square calculation $C = AA$.
- P05** Compare with CBLAS and assess execution times.

The examples are preparation

You now construct, integrate and test the assigned calculation yourself. No supplied toy executable runs this complete project.

P01 | Construct and store your matrix



What you implement

Construct every entry of the official matrix and store it in one flat array. Choose row-major or column-major and use that convention consistently.

Hints from E03

Packed row-major uses $i \text{ lda} + j$; column-major uses $i + j \text{ lda}$. At the official size, $\text{lda} = n = 2000$.

Avoid arithmetic traps

Use a signed-safe difference or convert before subtracting. A fourth power of a large integer can overflow; $\wedge$ is not C++ exponentiation.

Checkpoint before multiplication

Check selected entries, $A_{ii} = 1/\gamma$, and $A_{ij} = A_{ji}$. Explain your memory mapping. Positive entries alone do not prove positive definiteness.

P02 | Implement thirty unnormalized updates



$$x^{(k+1)} = Ax^{(k)}, \quad k = 0, \dots, 29.$$

What you implement

For each gamma, begin again with **all ones**. Use the complete old vector to compute the new vector, then calculate its norm.

Transfer from E02, E04 and E06

Use E02 for norms, E04 for matrix-vector products and E06 for updates. Adapt these building blocks in your own program.

Do not change the experiment

Do not normalize the vector and do not form A^{30} as a separate dense matrix. Neither is needed for the assigned recurrence.

Checkpoint: validate tiny cases, count all thirty updates, and check finite output.

What you produce

For each gamma, save the post-update iteration number and the Euclidean norm. There must be thirty records, indexed 1 through 30, plus any clearly identified header.

File and plot conventions

Use separate files and separate plots. A header such as `iteration,norm2` is a suggested convention, not a literal handout requirement. Match your plotting reader to your own schema.

Checkpoint before plotting

Confirm successful execution, actual file existence, correct column names, thirty rows and finite values. Label the axes, gamma, size and starting vector.

E06 is only a three-step demonstration. Its helper intentionally checks three records; adapt or write your project plotting script for thirty.

P04 | Implement the matrix square



$$C = A^2 = A A, \quad C_{ij} = \sum_{\ell=0}^{n-1} A_{i\ell} A_{\ell j}.$$

What you implement

Use your own row-column-sum triple loop. Reset each accumulator, keep the inputs unchanged, and store the result in a separate matrix.

Test small before scaling

E05 supplies hand-known checks for the general product and the square. Repeat those checks with your implementation before using the official matrix.

Plan the cost

Matrix-vector: $O(n^2)$ work; matrix-matrix: $O(n^3)$. With eight-byte doubles, one official matrix occupies 32 MB (decimal). Time depends on the machine.

What you implement

Adapt the E07 `cblas_dgemm` call to your own $A A$. Match the layout and dimensions; use no transpose, $\alpha = 1$, $\beta = 0$, and the actual leading dimensions.

Compare correctness before speed

Use the same stored inputs and precision. Check all output entries with a stated tolerance; only then interpret your timings.

Record the measurement

Use E08 to define the timing interval. Document size, gamma, compiler flags, library provider, repetitions and applicable thread settings. Your elapsed times must come from your runs.

Both gamma values are explicitly required for the vector experiment; the handout does not separately demand a two-gamma benchmark campaign.

Now write your own project



Independent implementation and testing: 70 minutes

Write your own project in its separate folder. The instructor will circulate, ask questions, and help diagnose difficulties.

Working checkpoints

First: tiny operations and indexing pass.

Next: build your matrix and implement the updates.

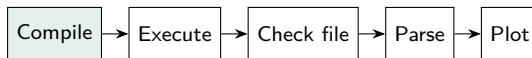
Then: save your data, compare the matrix square, and record evidence.

Ask with evidence

Show the active machine, working directory, exact command, first error or unexpected value, and the small case you expected to pass.

Use the complete examples as learning references. No provided command replaces your integrated implementation.

Debug the first failing stage



```
pwd
ls examples
# Immediately after a command, inspect its exit status:
echo $?
```

Follow the first failure

A missing header is not a wrong norm. A compiled but unfinished algorithm cannot create valid results. A missing CSV is not repaired by changing the graph label.

For your toy run: inspect `data/toy_run01.csv` only after E06 succeeds. On managed machines, report missing software instead of installing it.

Interpret results and prepare the report



Activity ZIP

Your source code and required numerical data. Include a short README with working compile/run commands. Test the archive before submission.

Separate PDF report

Method (about a page); task (about half a page); labelled results with short comments; conclusions (about half a page).

Interpret your own evidence

Show two gamma norm plots and discuss your manual/library timings. Explain observations and limitations, rather than pasting a terminal screenshot.

Published report deadline: within 7 days of the last laboratory; late reports lose 20 points per subsequent week. Confirm the actual UPeL assignment and deadline.

Exit check: explain your calculation

Representation and calculation

What does the leading dimension measure? Why is each row sum reset? Why must the old vector remain available during a multiplication?

Checking and interpretation

What independent small answer did you use? Why is equal output norm not enough to prove equal arrays? Does growing output alone prove a numerical failure?

Explain one design choice and one test

State what your timing interval includes. Identify any remaining task or environment blocker honestly. Keep your conclusions connected to your own calculations.

Optional E03 | Leading dimensions with padding



Optional: continuation of the same complete storage file.

```
// Optional extension: physical row length 5, logical row length 3.
const int lda = 5;
std::vector<double> padded(rows * lda, -99.0);
for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < cols; ++j) {
        padded[i * lda + j] = row[i * row_lda + j];
    }
}

std::cout << "padded row-major:";
for (double v : padded) std::cout << ' ' << v;
std::cout << '\n';
for (int i = 0; i < rows; ++i) {
    for (int j = 0; j < lda; ++j) {
        const double expected = (j < cols) ? i * cols + j + 1 : -99;
        if (padded[i * lda + j] != expected) return 1;
    }
}

std::cout << "All logical entries and padding checked\n";
return 0;
}
```

Check: six logical entries, four padding cells. $A_{1,0} = 4$ is at offset 5.

Optional E09 | Representation and printed precision



Two different stored starting values

0.1f is first rounded to float. Promoting that stored value to double does not undo the original rounding. 0.1 initializes a double directly.

Observed on this test platform:

float promoted to double = 0.10000000149011612

double initialized directly = 0.10000000000000001

float: radix=2, digits=24

double: radix=2, digits=53

Printed precision is not stored accuracy

More printed digits reveal the stored representation; they do not create more accurate information. E09 queries the actual type properties rather than assuming them.

Optional E09 | Compile the precision demonstration



A finished optional example; no fast-math assumptions.

```
#include <cmath>
#include <iomanip>
#include <iostream>
#include <limits>
int main() {
    const float f = 0.1f;
    const double from_float = static_cast<double>(f);
    const double d = 0.1;
    std::cout << std::setprecision(std::numeric_limits<double>::max_digits10);
    std::cout << "float promoted to double = " << from_float << '\n';
    std::cout << "double initialized directly = " << d << '\n';
    std::cout << "float radix=" << std::numeric_limits<float>::radix
        << " digits=" << std::numeric_limits<float>::digits
        << " epsilon=" << std::numeric_limits<float>::epsilon() << '\n';
    std::cout << "double radix=" << std::numeric_limits<double>::radix
        << " digits=" << std::numeric_limits<double>::digits
        << " epsilon=" << std::numeric_limits<double>::epsilon() << '\n';
    if (!std::isfinite(from_float) || !std::isfinite(d)) return 1;
    return 0;
}

g++ -std=c++17 -Wall -Wextra -Wpedantic -O2 \
    examples/09_precision_example.cpp -o bin/09_precision_example
./bin/09_precision_example
```

Optional reference | Terms and sources



Dot product	Sum of paired component products.
Euclidean norm	Nonnegative square root of the dot product with itself.
Leading dimension	Stored row/column stride, measured in elements.
CBLAS	C interface to Basic Linear Algebra Subprograms.
Roundoff	Difference caused by finite-precision arithmetic.

Source route

Course and rules; Project handout; Numerical Errors.

BLAS; CBLAS arguments; GCC linking; clock; type properties; WSL paths.

Detailed eigenvalue and backward-error theory are further reading, not prerequisites for this lab.

