# Lab 03

## Cache alignement - Eigen - Factories, variadic templates and traits

Paolo Joseph Baioni

October 17, 2025

# Cache alignement

## Introduction to exercise 1

- The implemented matrix class is organized as **column-major**, *i.e.* $A(i,j) =$ `data[i + j * rows()]`, conversion from 1d to 2d indexing is performed by the utility method `sub2ind`.
- Access to elements is implemented both in `const` and non-`const` versions, by overloading `operator()`.
- Data is private, *getter methods* expose what is needed to the user, both `const` and non-`const` versions are provided.
- Naive implementation of matrix-matrix multiplication is slow because it has low *data locality*, simply transposing the left matrix factor improves performance significantly[1].
- The `#include <ctime>` header provides timing utilities, `tic()` and `toc(x)` macros start and stop the timer.

[1]See *M. Kowarschik, C. Weiß. (2002). Lecture Notes in Computer Science. 213-232. DOI: 10.1007/3-540-36574-5_10* for further details.

# Cache alignement

## Exercise 1.1

Starting from the provided implementation of the class for dense matrices (and column vectors represented as 1-column matrices) based on `std::vector`, implement the following methods:

- transpose: $A = A^T$.
- `operator*`: matrix-matrix and matrix-vector multiplication.

# Cache alignement

## Exercise 1.2

- Transpose the first factor in matrix multiplication before performing the product.
- Compare the execution speed with respect to the previous implementation.

# Cache alignement

## Exercise 1.2 - Details

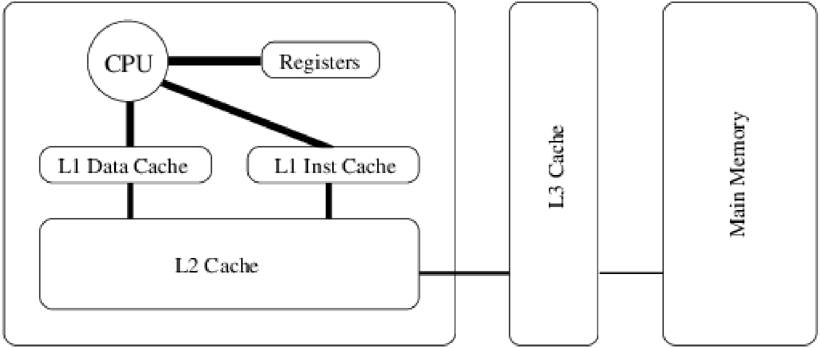


Figure: Typical memory layout of a computer.

# Cache alignement

## Exercise 1.2 - Details

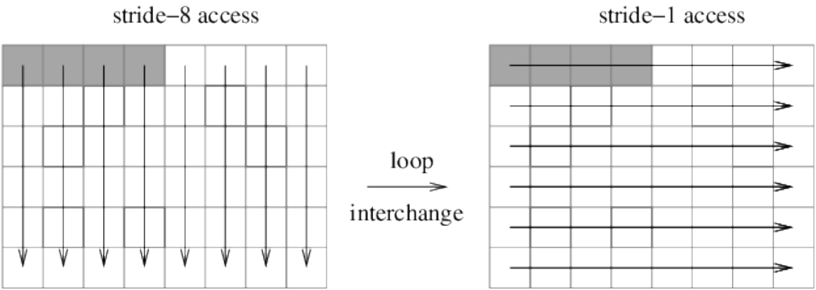


Figure: Example with a row-major matrix.

# Cache alignement

## Exercise 1.2 - Details

```
double sum;
double a[n, n];
// Original loop nest:
for j = 1 to n do
    for i = 1 to n do
        sum+ = a[i, j];
    end for
end for
```

```
double sum;
double a[n, n];
// Interchanged loop nest:
for i = 1 to n do
    for j = 1 to n do
        sum+ = a[i, j];
    end for
end for
```

Figure: Example with a row-major matrix.

# Eigen

## Exercise 1.3

- Include the `Eigen/Dense` header.
- Use the `Eigen::Map` template class to wrap the matrix data and interpret it as `Eigen::MatrixXd`.
- Compare the execution speed with respect to the previous implementations.

# Factories, variadic templates and traits

## Exercise 2 - Newton solver

This example (an extended version of `Examples/src/NewtonSolver`) is about a set of tools that implement generic Newton or quasi-Newton methods to determine the zero of scalar non-linear equations, as well as vector systems using the `Eigen` library.

The code structure is the following:

- `NewtonTraits` contains the definition of the types used by the main classes, to guarantee uniformity.
- `JacobianBase` is a base class which implements the action of a *quasi-Jacobian*: the user may choose among `FullJacobian` where the actual Jacobian must be specified by the user, and `DiscreteJacobian`, that approximates the Jacobian via finite differences.
- `JacobianFactory` instantiates a concrete derived class of `JacobianBase` family on the fly.
- `Newton` applies the Newton method, given the non-linear system and a `JacobianBase`.
- `NewtonOptions` and `NewtonResults` bind the input options and the output results.

# Factories, variadic templates and traits

## Exercise 2 - Newton solver

Consider the problem

$$\mathbf{f}(x, y) = \begin{bmatrix} (x-1)^2 + 0.1(y-5)^2 \\ 1.5 - x - 0.1y \end{bmatrix} = \mathbf{0}.$$

Starting from the provided solution sketch:

1. Implement the `NewtonTraits` class defining common types for homogeneity.
2. Implement the `FullJacobian` class (inheriting from `JacobianBase`) which, provided the full Jacobian matrix, solves the linear system using a direct solver with $LU$ factorization.
3. `DiscreteJacobian` (inheriting from `JacobianBase`) which approximates the system Jacobian using finite differences and solves the linear system using a direct solver with $LU$ factorization.
4. Implement a `JacobianFactory` method, returning an istance of `FullJacobian` or `DiscreteJacobian` depending on a parameter chosen by the user.
5. Solve the problem above using both the full and the discrete approach.