Project Title: *Linear Algebra Mini-Library in C++*

Overview

In this project, you will design and implement a small linear algebra library from scratch in C++. The library should handle fundamental operations on vectors and matrices, such as addition, subtraction, multiplication, and selected decompositions (e.g., LU or QR). You will also include basic statistical functions (mean, variance, standard deviation) that operate on the data structures you create. The goal is to demonstrate both a solid understanding of object-oriented design in modern C++ and core numeric techniques.

Learning Objectives

1. **Reinforce OOP and design principles** by creating a coherent library interface (headers, classes, methods).

2. **Practice with modern C++** features such as smart pointers, move semantics, const-correctness, and RAII.

3. **Gain familiarity with numerical methods** for linear algebra (matrix and vector operations, decompositions, and basic stats).

4. **Develop robust test cases** to ensure correctness and reliability of your library.

5. **Focus on performance considerations** (e.g., memory layout, cache efficiency, complexity analysis).

Requirements

1. Core Data Structures

1. **Vector Class**
   - Implement a `Vector` class to store real numbers (e.g., `double`).
   - **Constructor(s)**: default, parameterized (e.g., `Vector(size_t size)`, plus possibly a constructor that takes `std::initializer_list<double>` or a `std::vector<double>`).
   - **Element Access**: Provide both `operator[]` (with bounds checking in debug mode if desired) and perhaps a function `at(size_t index)`.
   - **Size**: A function returning the vector's size (`size()`).
2. **Matrix Class**
   - Implement a `Matrix` class to store real numbers in a 2D structure.
   - **Constructor(s)**: default, parameterized (e.g., `Matrix(size_t rows, size_t cols)`, plus optional advanced constructors).
   - **Element Access**: Provide `operator()(size_t row, size_t col)` (again, you may optionally include bounds checking in debug mode).
   - **Rows/Cols**: Functions to return the number of rows (`rows()`) and columns (`cols()`).

2. Basic Operations

1. **Vector-Vector**
   - **Addition/Subtraction**: `v1 + v2`, `v1 - v2` (must be same size).
   - **Dot Product**: `dot(v1, v2)` returns a scalar.
   - **Scalar Multiplication**: `v * scalar` and `scalar * v`.
2. **Matrix-Matrix**
   - **Addition/Subtraction**: `m1 + m2`, `m1 - m2` (must have same dimensions).
   - **Multiplication**: `m1 * m2` (dimensions must be compatible, resulting in a new `Matrix`).
3. **Matrix-Vector Multiplication**
   - Implement `m * v` which returns a new `Vector`.
4. **Utility Functions (optional but recommended)**
   - **Transpose**: A function to return the transpose of a matrix.
   - **Identity Matrix**: A function or static method to create an identity matrix of given size.

3. Decompositions (Choose at least one)

1. **LU Decomposition**
   - Write a function that takes a `Matrix` and returns its L (lower triangular) and U (upper triangular) factors.
   - Handle the case where the matrix is singular (you may return an error code or throw an exception).
2. **QR Decomposition**
   - Implement the Gram-Schmidt process or Householder transformations.
   - Return two matrices: Q (orthonormal) and R (upper triangular).
3. **Cholesky Decomposition** (if your matrix is guaranteed positive definite).

4. Basic Statistics

Implement **at least two** of the following on either a `Vector` or a row/column of a `Matrix`:

- **Mean**
- **Variance**
- **Standard Deviation**
- **Covariance** (matrix form if you like)

Example signature for a vector function:

```
double mean(const Vector& v);
double variance(const Vector& v);
```

5. Error Handling and Validation

- **Index Out of Bounds**: For debug builds, you may throw an exception (`std::out_of_range`) if an index is invalid.
- **Dimension Mismatch**: For operations requiring dimension checks, throw an exception or handle gracefully with an error message.

6. Testing

Create a set of **unit tests** that systematically validate your library. For instance:

- **Small Vectors/Matrices**
  - Check addition, subtraction, dot product.
- **Random Vectors/Matrices**
  - Generate random data and check if your results match a known reference or approximate known properties.
- **Decomposition Tests**
  - For LU, verify that `L * U` reconstructs the original matrix (within a small numerical epsilon).
  - For QR, verify that `Q * R` reconstructs the original matrix and that Q is orthonormal (`Q^T Q = I`).

You may use a testing framework like **Google Test** or **Catch2**, or you can write simple `assert` statements.

7. Performance (Optional for Extra Credit)

1. **Compare performance**
   - Measure how long it takes to multiply two matrices using your library vs. a naive approach.
   - Optionally compare to a well-known library (e.g., **Eigen**, **Blaze**, or **Armadillo**) in a small benchmark.
2. **Optimizations**
   - Investigate row-major vs. column-major storage.
   - Explore loop unrolling, blocking, or vectorization with compiler intrinsics (advanced).

Deliverables

1. **Source Code**
   - Organized into `.h` (header) and `.cpp` (implementation) files.
   - A CMake or Makefile to build the project.
   - Clean, well-commented code.
2. **Test Files**
   - A separate test executable or test suite showcasing the correctness of your library.
3. **Documentation**
   - A short `README.md` explaining how to build and run tests.
   - Brief explanations of each function's purpose and usage (in-code comments and/or Doxygen).
4. **(Optional) Performance Report**
   - If you explore optimizations, include a short write-up (1–2 pages) with benchmark results.

Evaluation Rubric

- **Correctness (40%)**
  - Does your library correctly implement the specified operations?

  - Do decompositions (if implemented) accurately reconstruct matrices?
- **Code Quality (30%)**
  - Proper use of modern C++ (smart pointers, RAII, const correctness, etc.).
  - Organization, readability, and maintainability.
- **Testing & Verification (20%)**
  - Coverage of core functions with tests.
  - Clear pass/fail criteria.
- **Documentation & Build (10%)**
  - Ease of compilation and usage instructions.
  - Quality of comments and/or generated documentation.
- **Extra Credit**
  - Performance optimizations or advanced features (additional decompositions, concurrency, etc.).

Submission Guidelines

1. **Git Repository**: Submit a link to a public or private repository containing all required files (source, tests, build scripts).

2. **ZIP Archive** (if required by your course platform): Include all source files, a `README.md`, and any documentation.

3. **Testing Demonstration**: Provide instructions or a script that compiles the library and runs the tests.

Good luck, and have fun! By completing this assignment, you'll deepen your knowledge of C++ design, numeric methods, and best practices for building reliable, high-performance libraries.