VIETNAMESE-GERMAN UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

PROGRAMMING 2

THE TINY PROJECT TECHNICAL REPORT

Instructor - Prof. Huỳnh Trung Hiếu

Student 1 - Võ Minh Duy 10423028
Student 2 - Đỗ Minh Triết 10423114
Student 3 - Phạm Nguyên Phương 10423094

HO CHI MINH CITY, 06/2025

Table of Contents

Chapter 1

Introduction

This report outlines the development of a C++ project centered on numerical linear algebra and its application to a real-world machine learning problem. The project involved two main parts.

The first part centered on the creation of fundamental C++ classes for handling vectors and matrices. These classes were built with essential functionalities, including memory management, operator overloading for common algebraic operations, and methods for matrix computations such as determinant, inverse, and pseudo-inverse. Based on these foundational classes, a `LinearSystem` class was developed to solve systems of linear equations ($Ax = b$) using Gaussian elimination with pivoting. Additionally, a specialized derived class, `PosSymLinSystem`, was implemented to handle positive definite symmetric linear systems using the Conjugate Gradient method. The project also addressed solving under-determined or over-determined linear systems.

The second part of the project applied these custom-built classes to predict relative CPU performance using the Computer Hardware dataset from the UCI Machine Learning Repository. This entailed setting up a linear regression model, partitioning the dataset into training and testing sets, employing the implemented linear system solvers to determine model parameters, and evaluating the model's performance with the Root Mean Square Error (RMSE).

This report is structured as follows:

- Chapter 2 provides a detailed overview of the design and implementation of the `Vector`, `Matrix`, `LinearSystem`, and `PosSymLinSystem` classes.
- Chapter 3 describes the application of these classes to the linear regression task, covering data processing, model training, and evaluation of results.
- Chapter 4 concludes the report with a summary of the work accomplished and potential future enhancements.

The C++ implementation uses standard libraries and follows object-oriented programming principles to create a reusable and robust set of tools for linear algebra computations.

Chapter 2

Implementation (Section A)

This chapter details the design and implementation of the core C++ classes for vector and matrix operations, as well as classes for solving linear systems, as required by Part A of the project specification.

2.1 Vector Class

The `Vector` class was developed to represent mathematical vectors and perform common vector operations. Its implementation in `Vector.h` and `Vector.cpp` addresses the specified requirements.

2.1.1 Key Features

Key features of the `Vector` class include:

- **Manage Memory** Used dynamic memory allocation to store vector elements, ensuring efficient memory usage.
  - Creating an empty vector with a default constructor, `Vector()`.
  - Initializing a vector with a specified size using the constructor `Vector(int size)`.
  - Perform a deep copy with the constructor `Vector`.
  - Free the allocated memory in the destructor of `Vector()`.

The private members are `mSize` (the size of the array) and `mData` (a pointer to a `double` storing the vector elements).

- **Overload Operators**
  - The assignment operator (=) is overloaded to ensure deep copying of vector contents, correctly handling cases of self-assignment and vectors of differing sizes.
  - Unary operators (+) and (-) are overloaded: (+) returns a duplicate of the vector, while (-) produces a new vector with all elements negated.
  - Binary operators (+) and (-) are overloaded for vector addition and subtraction, and (*) for multiplication by a scalar. Assertions (assert) are used to verify that vectors involved in addition or subtraction have matching sizes.

- The square bracket operator ([]) is overloaded to provide 0-based element access, with an assert used to check index validity. Both const and non-const versions are implemented.
- The round bracket operator (()) is overloaded for 1-based element access, also using assert for bounds checking. Both const and non-const versions are available.
- The output stream operator (operator«) is overloaded as a friend function to enable easy printing of vector contents.

- **Norm Calculation:** The public method double Norm(int p = 2) const computes the p-norm of the vector, with the default being the Euclidean (L2) norm.
- **Accessor:** The method `int GetSize() const` provides the current size of the vector.

2.2 Matrix Class

The `Matrix` class, defined in `Matrix.h` and `Matrix.cpp`, is used for handling and operating on two-dimensional matrices.

2.2.1 Features and Implementation

- **Private Members:** The class maintains the number of rows and columns as private integers (`mNumRows, mNumCols`), and stores matrix data in a dynamically allocated 2D array (`mData`, a `double**`). Memory management is handled by private methods `AllocateMemory()` and `DeallocateMemory()`.
- **Constructors and Destructors:**
  - The default constructor `Matrix()` creates an empty matrix.
  - The constructor `Matrix(int numRows, int numCols)` allocates memory for the specified dimensions and initializes all elements to zero.
  - The copy constructor `Matrix(const Matrix& otherMatrix)` performs a deep copy of another matrix.
  - The destructor `~Matrix()` releases all allocated memory.
- **Accessors:** Methods `GetNumberOfRows()` and `GetNumberOfColumns()` provide access to matrix dimensions.
- **Operator Overloading:**
  - The assignment operator (`operator=`) is overloaded for deep copying.

- The round bracket operator (()) is overloaded for 1-based element access, with both const and non-const versions and bounds checking via `assert`.
- Unary (`+`, `-`) and binary (`+`, `-`, `*`) operators are overloaded for matrix arithmetic, including addition, subtraction, multiplication (matrix-matrix, scalar-matrix, and matrix-vector). Assertions ensure valid dimensions.
- The output stream operator (`operator«`) is overloaded for easy printing.

- **Matrix Operations:**
  - `Matrix Transpose() const`: Returns the transpose of the matrix.
  - `Double Determinant() const`: Calculates the determinant recursively using cofactor expansion, with assertions for square, non-empty matrices.
  - `Matrix Inverse() const`: Computes the inverse of a square, non-singular matrix using the adjugate method, with checks for squareness and non-zero determinant.
  - `Matrix PseudoInverse() const`: Calculates the Moore-Penrose pseudo-inverse, handling both $m \geq n$ and $m < n$ cases, and checks for singularity in the relevant matrices.
  - Static utility methods `SwapRows` and `SwapRowsMatrixOnly` are provided for row operations, useful in solvers.

2.3 LinearSystem Class

The `LinearSystem` class (`LinearSystem.h`, `LinearSystem.cpp`) is designed to solve equations of the form $Ax = b$, where A is a square, non-singular matrix.

2.3.1 Features and Implementation

- **Protected Data Members:** Includes `mSize` (size of the system), `mpA` (pointer to a `Matrix`), `mpb` (pointer to a `Vector`), and `mOwnsPointers` (ownership flag).
- **Constructors and Destructors:**
  - The main constructor `LinearSystem(Matrix& A, Vector& b, bool copyData = true)` initializes the system, checks that A is square and matches b, and manages data ownership.
  - A copy constructor is provided for deep copying when needed.

- A default constructor initializes an empty system, though the project suggested restricting its use.
- The destructor frees memory if the object owns its data.

- **Solver Method:**
  - The public `virtual Vector Solve()` method uses Gaussian elimination with partial pivoting for stability, working on copies of *A* and *b*.
  - Returns the solution vector, and issues warnings or returns a zero vector if the matrix is nearly singular.
- **Accessors:** Methods `GetMatrix()` and `GetRHSVector()` return copies of *A* and b.

2.4 PosSymLinSystem Class

The `PosSymLinSystem` class (`PosSymLinSystem.h`, `PosSymLinSystem.cpp`) extends `LinearSystem` to handle symmetric positive definite systems.

2.4.1 Features and Implementation

- **Inheritance:** Inherits publicly from `LinearSystem`, with protected access to base members.
- **Constructor:** The constructor checks that *A* is symmetric using a private helper function. Positive definiteness is not explicitly checked.
- **Overridden Solve Method:** The `Solve()` method implements the Conjugate Gradient (CG) algorithm:
  - Initializes x_0 as the zero vector, computes $r_0 = b - Ax_0$, and sets $p_0 = r_0$.
  - Iterates until the residual norm is below a set tolerance or a maximum number of iterations is reached.
  - Warns if the denominator for α is close to zero and notifies if convergence fails.

2.4 Solving Non-Square Systems

The project also required handling linear systems where *A* is not square.

- The `Matrix::PseudoInverse()` method is the main tool for this, providing the Moore-Penrose pseudo-inverse for least-squares (over-determined) or minimum-norm (under-determined) solutions.

- For linear regression in Part B, the over-determined system $X\beta = y$ is solved by forming the normal equations $X^TX\beta = X^Ty$, making the system square and solvable with `LinearSystem::Solve()`. This is equivalent to using the pseudo-inverse:

  $\beta = (X^T X)^{-1} X^T y$.

- Although Tikhonov regularization was mentioned in the requirements, it was not implemented as a separate solver. The focus was on the pseudo-inverse and normal equations for the project's needs.

These classes together provide a comprehensive framework for linear algebra operations, meeting the requirements of Part A.

Chapter 3

Application (Section B)

This regression model is self-provided from the previous part of the project.

3.1 Usage of other classes

3.1.1. Vector

- Holds feature vectors and prediction targets (Vector b, y_true, y_pred_new, y_pred_old).
- Stores model coefficients (x_old, x_new, x_best).
- Computes RMSE by element-wise diff and dot-product.

```
static double computeRMSE(const Vector& T, const Vector& P) {
    if (T.size() != P.size())
        throw invalid_argument("RMSE size mismatch");
    double s=0;
    for (size_t i=0;i<T.size();++i) {
        double d = T[i] - P[i];
        s += d*d;
    }
    return my_sqrt(s / T.size());
}
```

3.1.2. Matrix

- Builds the design matrix A (dimensions n_train × 6) from CPURecord features.
- Multiplies with coefficient vectors to compute predictions (via Matrix::operator*(Vector)).
- In the rectangular case (lambda=0, non-square), LinearSystem falls back on pseudo-inverse under the hood.

```
Matrix A(n_train, 6);
for (int i = 0; i < n_train; ++i) {
    auto& r = all[i];
    A(i+1,1)=r.MYCT; A(i+1,2)=r.MMIN;
    A(i+1,3)=r.MMAX; A(i+1,4)=r.CACH;
    A(i+1,5)=r.CHMIN; A(i+1,6)=r.CHMAX;
}
```

3.1.3. LinearSystem

- Instantiates LinearSystem sys(A, b) each epoch (default λ=0).
- Calls sys.Solve() to get new coefficients x_new.
- Comparison of x_new vs. previous x_old on test split drives model selection.

```
LinearSystem sys(A,b);
Vector x_new = sys.Solve();
```

3.1.4.PosSymLinSystem

- When $\lambda > 0$, main.cpp forms the normal equations $(A^TA + \lambda I)\, x = A^Tb$ and invokes:

```
PosSymLinSystem psys(ATA, rhs);
Vector x_new = psys.Solve();
```

- This avoids a full matrix inverse, improving efficiency for high-dimensional or strongly regularized problems.

3.1.5. Overall Data-Processing Pipeline in main.cpp

- **Data Loading & Shuffling:**

- loadData() reads raw CSV into CPURecord[].
- shuffleData() randomizes the order before each epoch.

```
CPURecord* all = loadData("data/machine.data", N);
shuffleData(all, N);
```

- **Train/Test Split**
  - 80/20 split on the shuffled array.

```
int n_train = int(0.8 * N), n_test = N - n_train;
```

- **Model Training & Evaluation***
  - Build Matrix A and Vector b for training features/targets.

```
Matrix A(n_train, 6);
Vector b(n_train);
// loop to fill A and b shown above in 3.1.2
```

- Solve for x_new via LinearSystem.

```
LinearSystem sys(A,b);
Vector x_new = sys.Solve();
```

- Evaluate both x_new and x_old on the test set (using Matrix×Vector or a manual loop).

```
for (int i = 0; i < n_test; ++i) {
    // build prediction for y_pred_new[i] and y_pred_old[i]
}
double rmse_new = computeRMSE(y_true, y_pred_new);
double rmse_old = computeRMSE(y_true, y_pred_old);
```

- Compute RMSE via computeRMSE().

- **Model Selection & Persistence**

- Choose the coefficient vector with lower RMSE.

```
Vector x_best = (rmse_new <= rmse_old ? x_new : x_old);
```

- Write the chosen vector to model.txt.

```
std::ofstream outM(modelFile);
for (size_t k = 0; k < x_best.size(); ++k)
    outM << x_best[k] << "\n";
```

- Append a timestamped RMSE entry to rmse.log.

```
std::ofstream logFile(outDir + "/rmse.log", std::ios::app);
logFile << timestamp << " [epoch " << ep << "] RMSE = " << rmse_best <<
"\n";
```

- Finally, renumber epochs with renumberEpochsInLog().

```
renumberEpochsInLog(outDir + "/rmse.log");
```

3.2 Remarks

The three core linear-algebra modules (Vector, Matrix, LinearSystem) provide all the data structures and algorithms that drive the regression workflow in main.cpp. By decoupling storage, arithmetic, and solver logic, main.cpp remains focused on I/O, data splits, epoch management, and logging, while relying on well-encapsulated classes for all numerical routines.

Chapter 4

Conclusion

This project met the objectives set out in the specification, covering both the creation of a basic C++ linear algebra library and its use in a linear regression scenario.

4.1 Summary of Accomplishments

- **Part A: Core Linear Algebra Classes:**
  - Developed a robust `Vector` class with dynamic memory handling, overloaded operators for standard arithmetic (including scalar operations), and both 0-based and 1-based indexing with bounds checking.
  - Built a comprehensive `Matrix` class supporting memory management, multiple constructors (including deep copy), operator overloading for various matrix operations, and methods for determinant, inverse (for square matrices), and Moore-Penrose pseudo-inverse (for general matrices). This enables handling both regular and irregular systems, including nonsquare matrices.
  - Implemented a `LinearSystem` class to solve $Ax = b$ for square matrices using Gaussian elimination with partial pivoting, with careful attention to class design and memory management.
  - Created a derived `PosSymLinSystem` class for efficiently solving symmetric positive definite systems via the Conjugate Gradient method, demonstrating polymorphism by overriding the virtual `Solve` method and including symmetry checks.
- **Part B: Linear Regression Application:**
  - Applied the developed classes to perform linear regression on the UCI Computer Hardware dataset.

- Selected relevant features, split the data into training (80%) and testing (20%) sets, and formulated the regression problem using normal equations.
- Used the `LinearSystem` class to solve for model parameters.
- Evaluated model performance using Root Mean Square Error (RMSE) on both training and testing sets, highlighting differences that may indicate overfitting or the limitations of a simple linear model for this dataset.

4.2 Challenges and Lessons Learned

This project offered valuable experience in C++ programming and numerical methods:

- **Object-Oriented Design:** Gained experience designing interconnected classes (`Vector`, `Matrix`, `LinearSystem`, `PosSymLinSystem`) with proper encapsulation, inheritance, and polymorphism.
- **Memory Management:** Practiced dynamic memory allocation (`new`, `delete`, `delete[]`) and learned the importance of constructors, destructors, and copy semantics (Rule of Three/Five).
- **Operator Overloading:** Developed intuitive interfaces for mathematical objects through operator overloading.
- **Numerical Algorithms:** Implemented key numerical linear algebra algorithms such as Gaussian elimination with pivoting, the Conjugate Gradient method, and methods for determinant, inverse, and pseudo-inverse, while considering their mathematical foundations and numerical stability.
- **Machine Learning Application:** Connected a custom numerical library to a practical data analysis task, reinforcing the role of linear algebra in machine learning techniques like linear regression.

A significant challenge was ensuring numerical stability and correctness, especially for matrix inversion and solving systems. Debugging memory management and operator behavior also required careful attention.

4.3 Opportunities for Future Work

This project provides a solid foundation for further enhancements. Possible future directions include:

- **Advanced Numerical Methods:** Adding more sophisticated matrix decompositions for improved stability or efficiency.
- **Sparse Matrix Support:** Extending the library to efficiently handle sparse matrices with specialized storage and algorithms.
- **Templated Classes:** Making `Vector` and `Matrix` class templates to support various numerical types.
- **Improved Error Handling:** Replacing assertions with a more robust exception handling system.
- **Regularization Methods:** Implementing techniques like Tikhonov regularization or Ridge regression within the solvers or regression framework. PROGRAMMING 2 TECHNICAL REPORT Page 8 Vietnamese-German University Department of Computer Science
- **Expanded Machine Learning Applications:** Enhancing the application with more thorough model evaluation, feature scaling, or comparisons with other regression models.

In summary, this project demonstrates the ability to design, implement, and apply a C++ library for essential linear algebra operations. It fulfills the requirements and offers meaningful experiences in both software development and numerical computation.

References

- **Source code:** https://github.com/sleepysphere/TheTinyProject This repository holds all the implementation of the project, including all scripts and documentation.
- **`run.bat`:** A .bat script used for executing `main.cpp` C++ code