VIETNAMESE - GERMAN UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

PROGRAMMING 2
TINY PROJECT TECHNICAL REPORT

Student 1: 10423099 - Dang Huu Trung Son
Student 2: 10423082 - Nguyen Dinh Nguyen
Student 3: 10423140 - Bui Huy Hoang
Student 4: 10423186 - Pham Quoc Vuong

HO CHI MINH CITY, 06/2025

Content

Chapter 1

Introduction

This report details the development of a C++ project focused on numerical linear algebra and its application to a practical 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 such as memory management, operator overloading for common algebraic operations, and methods for matrix computations like 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

Core Classes Implementation (Part 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 Features and Implementation

Key features and implementation details include:

- **Memory Management:** The class includes constructors and a destructor to handle dynamic memory allocation and deallocation for the vector's data.
  - A default constructor `Vector()` creating an empty vector.
  - A constructor `Vector(int size)` that allocates memory for a given size and initializes elements to zero.
  - A copy constructor `Vector` performing a deep copy.
  - The destructor `~Vector()` ensures that dynamically allocated memory for `mData` is freed.

  The private members are `mSize` (the size of the array) and `mData` (a pointer to a `double` storing the vector elements).
- **Operator Overloading:**
  - Assignment operator (=) is overloaded for proper deep copying of vector data and handling self-assignment and different sizes.
  - Unary operators (+) (returns a copy of the vector) and (−) (returns a new vector with negated elements) are overloaded.

```
1  class Vector {
2  private:
3      int mSize;
4      double* mData;
5
6  public:
7      Vector();
8      Vector(int size);
9      Vector(const Vector& otherVector);
10     ~Vector();
11
12     int GetSize() const;
13
14     Vector& operator=(const Vector& otherVector);
15     double& operator[](int index);
16     const double& operator[](int index) const;
17     double& operator()(int index);
18     const double& operator()(int index) const;
19
20     Vector operator+() const;
21     Vector operator-() const;
22
23     Vector operator+(const Vector& v1) const;
24     Vector operator-(const Vector& v1) const;
25     Vector operator*(double scalar) const;
26
27     friend Vector operator*(double scalar, const Vector& v);
28     friend std::ostream& operator<<(std::ostream& os, const Vector& v);
29
30     double Norm(int p = 2) const;
31 };
```

Listing 2.1: Implementation of the `Vector` class

  - Binary operators (+) for vector addition, (−) for vector subtraction, and (⋆) for scalar multiplication are overloaded. Assertions (`assert`) are used to ensure size compatibility in vector-vector operations.
  - The square bracket operator (`[]`) is overloaded for 0-based access. It provides a check that the index lies within the correct range using `assert`. Both `const` and non-`const` versions are provided.
  - The round bracket operator (`()`) is overloaded for 1-based access. It also uses `assert` for range checking. Both `const` and non-`const` versions are provided.
  - An overloaded `operator«` (friend function) allows printing vectors to an output stream.

- **Norm Calculation:** A public method `double Norm(int p = 2) const` calculates the p-norm of the vector (defaulting to Euclidean norm L2).
- **Accessor:** `int GetSize() const` returns the size of the vector.

2.2 Matrix Class

The `Matrix` class, implemented in `Matrix.h` and `Matrix.cpp`, is designed for representing and manipulating 2D matrices.

2.2.1 Features and Implementation

- **Private Members:** The class stores matrix dimensions as private integer members `mNumRows` and `mNumCols`, and the matrix data using `mData`, a `double**` (pointer to a pointer of doubles). Helper private methods `AllocateMemory()` and `DeallocateMemory()` manage the 2D dynamic array.
- **Constructors and Destructor:**
  - A default constructor `Matrix()` creating an empty matrix.
  - A constructor `Matrix(int numRows, int numCols)` that accepts the number of rows and columns, allocates memory using `AllocateMemory()`, and initializes all entries to zero.
  - An overridden copy constructor `Matrix(const Matrix& otherMatrix)` that performs a deep copy of the matrix data.
  - An overridden destructor `~Matrix()` that deallocates memory using `DeallocateMemory()`.
- **Accessors:** Public methods `GetNumberOfRows()` and `GetNumberOfColumns()` are provided.
- **Operator Overloading:**

```
1  class Matrix {
2  private:
3      int mNumRows;
4      int mNumCols;
5      double** mData;
6
7      void AllocateMemory();
8      void DeallocateMemory();
9
10 public:
11     Matrix();
12     Matrix(int numRows, int numCols);
13     Matrix(const Matrix& otherMatrix);
14     ~Matrix();
15
16     int GetNumberOfRows() const;
17     int GetNumberOfColumns() const;
18
19     Matrix& operator=(const Matrix& otherMatrix);
20     double& operator()(int i, int j);
21     const double& operator()(int i, int j) const;
22
23     Matrix operator+() const;
24     Matrix operator-() const;
25
26     Matrix operator+(const Matrix& m1) const;
27     Matrix operator-(const Matrix& m1) const;
28     Matrix operator*(const Matrix& m1) const;
29     Matrix operator*(double scalar) const;
30     Vector operator*(const Vector& v) const;
31
32     friend Matrix operator*(double scalar, const Matrix& m);
33     friend std::ostream& operator<<(std::ostream& os, const Matrix& m);
34
35     Matrix Transpose() const;
36     double Determinant() const;
37     Matrix Inverse() const;
38     Matrix PseudoInverse() const;
39
40     static void SwapRows(Matrix& A, Vector& b, int r1, int r2);
41     static void SwapRowsMatrixOnly(Matrix& A, int r1, int r2);
42 };
```

Listing 2.2: Implementation of the `Matrix` class

- Overloaded assignment operator `operator=` for deep copying.
- The round bracket operator (`()`) is overloaded for 1-based access to matrix entries. Both `const` and non-`const` versions are provided, with `assert` for index validation.
- Unary (+,−) and binary operators (+,−,⋆) are overloaded for matrix addition, subtraction, matrix-matrix multiplication, scalar-matrix multiplication, and matrix-vector multiplication (`matrix * vector`). `assert` statements ensure dimensional compatibility.
- An overloaded `operator«` (friend function) allows printing matrices to an output stream.

- **Matrix Operations:**
  - `Matrix Transpose() const`: Computes the transpose.
  - `double Determinant() const`: Computes the determinant of a square matrix using recursive cofactor expansion. Asserts matrix is square and non-empty.
  - `Matrix Inverse() const`: Computes the inverse of a square, non-singular matrix using the adjugate matrix method. Asserts matrix is square, non-empty, and determinant is non-negligible.
  - `Matrix PseudoInverse() const`: Computes the Moore-Penrose pseudo-inverse. It handles $m \geq n$ via $(A^T A)^{-1} A^T$ and $m < n$ via $A^T (A A^T)^{-1}$. Includes singularity checks for $A^T A$ or $A A^T$.
  - Static methods `SwapRows` and `SwapRowsMatrixOnly` are utility functions, likely intended for use in solvers.

2.3 LinearSystem Class

Implemented in `LinearSystem.h` and `LinearSystem.cpp`, this class is designed to solve $Ax = b$ where A is square and non-singular.

```
1  class LinearSystem {
2  protected:
3      int mSize;
4      Matrix* mpA;
5      Vector* mpb;
6      bool mOwnsPointers;
7
8  private:
9      LinearSystem();
10     LinearSystem(const LinearSystem& other);
11
12
13 public:
14     LinearSystem(Matrix& A, Vector& b, bool copyData = true);
15     virtual ~LinearSystem();
16
17     virtual Vector Solve();
18
19     Matrix GetMatrix() const;
20     Vector GetRHSVector() const;
21 };
```

Listing 2.3: Implementation of the `LinearSystem` class

2.3.1 Features and Implementation

- **Data Members (Protected):** `mSize` (integer), `mpA` (`Matrix*`), `mpb` (`Vector*`), and `mOwnsPointers` (bool).
- **Constructors and Destructor:**
  - The primary constructor `LinearSystem(Matrix& A, Vector& b, bool copyData = true)` initializes the system. `mSize` is derived from A. Asserts ensure A is square and compatible with b. `copyData` controls whether A and b are copied or pointed to.
  - A copy constructor `LinearSystem(const LinearSystem& other)` is provided for deep copying if data is owned.
  - A default constructor `LinearSystem()` is defined (initializes to empty state), but the project description suggested preventing its use if a specialized constructor is primary. The task seems to have allowed it.
  - The destructor `~LinearSystem()` deallocates `mpA` and `mpb` if `mOwnsPointers` is true.
- **Solver Method:**

- A public `virtual Vector Solve()` method implements Gaussian elimination with partial pivoting for numerical stability. It creates working copies of the matrix and RHS vector.
- It returns a `Vector` containing the solution x.
- Includes warnings for nearly singular matrices during elimination and back substitution, returning a zero vector in such cases.

- **Accessors:** `GetMatrix()` and `GetRHSVector()` return copies of the matrix A and vector b.

2.4 PosSymLinSystem Class

The `PosSymLinSystem` class (`PosSymLinSystem.h`, `PosSymLinSystem.cpp`) derives from `LinearSystem` for solving symmetric positive definite systems.

```
1  class PosSymLinSystem : public LinearSystem {
2  public:
3      PosSymLinSystem(Matrix& A, Vector& b, bool copyData = true);
4      ~PosSymLinSystem() override = default;
5
6      Vector Solve() override;
7
8  private:
9      bool IsSymmetric(const Matrix& A) const;
10 };
```

Listing 2.4: Implementation of the `PosSymLinSystem` class

2.4.1 Features and Implementation

- **Inheritance:** Publicly inherits from `LinearSystem`. Member data of `LinearSystem` is correctly `protected` for access.
- **Constructor:** `PosSymLinSystem(Matrix& A, Vector& b, bool copyData = true)` calls the base constructor. It includes an `assert` to check if A is symmetric via the private helper `IsSymmetric(const Matrix& A) const`. A positive definiteness check is not performed.
- **Overridden Solve Method:** The `virtual Vector Solve() override` method implements the Conjugate Gradient (CG) iterative algorithm.
  - It initializes x_0 (to zero vector), $r_0 = b - Ax_0$, $p_0 = r_0$.
  - Iteratively updates x, r, and p until the norm of the residual r is below a tolerance (10^{-9}) or a maximum number of iterations ($2 \times$ mSize) is reached.

- Includes a warning if CG denominator for α is near zero.
- Prints a message if convergence is not achieved.

2.5 Solving Non-Square Systems

The project required addressing solutions for under-determined or over-determined linear systems where A is not square.

- The `Matrix::PseudoInverse()` method provides the primary mechanism for this. It computes the Moore-Penrose pseudo-inverse, enabling least-squares solutions for over-determined systems or minimum-norm solutions for under-determined systems.
- For the linear regression in Part B, an over-determined system $X\beta = y$ is solved by converting it to the normal equations $X^T X\beta = X^T y$. The matrix $X^T X$ is square, and this system is then solved using `LinearSystem::Solve()`. This approach implicitly uses the concept of the pseudo-inverse, as $\beta = (X^T X)^{-1} X^T y$.
- Tikhonov regularization, though mentioned as an option in the requirements, was not explicitly implemented as a separate solver in the provided C++ code for $Ax = b$ where A is non-square. The focus was on the pseudo-inverse capability and the normal equations for the specific application.

The implemented classes provide a solid foundation for various linear algebra tasks, fulfilling the requirements of Part A.

Chapter 3

Application: CPU Performance Prediction (Part B)

This chapter describes the application of the developed C++ linear algebra library to a practical machine learning problem: the linear regression prediction of relative CPU performance, as outlined in Part B of the project requirements. The implementation is primarily contained within `main.cpp`.

3.1 Problem Description and Dataset

The objective is to predict the "published relative performance" (PRP) of computer hardware using several of its key characteristics.

- **Dataset Source:** The "Computer Hardware" dataset was obtained from the UCI Machine Learning Repository. This dataset comprises 209 instances, each described by 10 attributes (vendor name, model name, 6 predictive numerical attributes, PRP, and ERP).
- **Selected Features for the Model:** Following the project specification, 5 specific attributes are used as predictive features for the linear model:
  1. MYCT: machine cycle time in nanoseconds (integer)
  2. MMIN: minimum main memory in kilobytes (integer)
  3. MMAX: maximum main memory in kilobytes (integer)
  4. CACH: cache memory in kilobytes (integer)
  5. CHMAX: maximum channels in units (integer)

  The attributes vendor name, model name, CHMIN (minimum channels), and ERP (estimated relative performance from original article) are not used in this specific model formulation.
- **Target Variable:** The continuous variable to be predicted is PRP (published relative performance).
- **Linear Regression Model Equation:** The relationship between the features and the target is modeled by the linear equation:

$$PRP = x_1 \cdot \text{MYCT} + x_2 \cdot \text{MMIN} + x_3 \cdot \text{MMAX} + x_4 \cdot \text{CACH} + x_5 \cdot \text{CHMAX}$$

The goal is to determine the optimal values for the parameters x_1, x_2, x_3, x_4, x_5.

3.2 Methodology

The `main.cpp` file orchestrates the data loading, preprocessing, model training, and evaluation.

3.2.1 Data Handling and Preprocessing

- **Data Reading:** The program reads data from the `machine.data` file. Each line is parsed, splitting by commas. The specified feature columns (indices 2, 3, 4, 5, 7) and the target column (index 8) are extracted and converted to `double`. Error handling for invalid data conversion or incorrect column counts per line is included, with problematic lines being skipped and warnings issued.
- **Train-Test Split:** The dataset of 209 instances is divided into a training set and a testing set. 80% of the data (167 instances) is allocated for training, and the remaining 20% (42 instances) for testing. This split is performed sequentially: the first 167 records form the training set, and the subsequent 42 records constitute the testing set. These are stored in `Matrix` objects (`X_train`, `X_test`) and `Vector` objects (`y_train`, `y_test`).

3.2.2 Model Parameter Estimation

The model parameters (x_i, collectively denoted as vector β) are determined using the training data.

- **Normal Equations:** The linear regression problem $X_{train}\beta = y_{train}$ is typically over-determined. The least-squares solution for β is found by solving the normal equations:

$$(X_{train}^T X_{train})\beta = X_{train}^T y_{train}$$

  Let $A_{ls} = X_{train}^T X_{train}$ and $b_{ls} = X_{train}^T y_{train}$. The system becomes $A_{ls}\beta = b_{ls}$.
- **Solving the System:** The `main.cpp` program constructs A_{ls} (a 5×5 matrix) and b_{ls} (a 5×1 vector) using the implemented `Matrix::Transpose()`, `Matrix::operator*()`, and `Vector::operator*()` methods.
- An instance of `LinearSystem` is then created using A_{ls} and b_{ls}. The `LinearSystem::Solve()` method (which employs Gaussian elimination) is invoked to compute the parameter vector β.

3.2.3 Model Evaluation

- **Prediction:** Using the obtained parameters β, predictions are made for both the training set ($y_{pred\_train} = X_{train}\beta$) and the testing set ($y_{pred\_test} = X_{test}\beta$). These matrix-vector multiplications are handled by the overloaded operators in the `Matrix` and `Vector` classes.

- **Root Mean Square Error (RMSE):** The model's predictive accuracy is assessed using the RMSE criterion, calculated as $RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_{actual,i} - y_{pred,i})^2}$. The RMSE is computed separately for the training and testing sets to gauge both model fit and generalization ability.

3.3 Results and Discussion

```
Reading data from machine.data...
Successfully read 209 records.
Total samples: 209
Training samples: 167
Testing samples: 42
Training and testing sets created.
Formulating linear system...
Matrix A_ls (X_train^T * X_train):
[[21510945.0000, 43442804.0000, 216294780.0000, 266219.0000, 362810.0000]
 [43442804.0000, 4205880328.0000, 11350446048.0000, 24712452.0000, ↩
    11587524.0000]
 [216294780.0000, 11350446048.0000, 39827749608.0000, 73804144.0000, ↩
    41905772.0000]
 [266219.0000, 24712452.0000, 73804144.0000, 339543.0000, 120820.0000]
 [362810.0000, 11587524.0000, 41905772.0000, 120820.0000, 117195.0000]]
Vector b_ls (X_train^T * y_train):
(1444722.0000, 133693884.0000, 386488796.0000, 965668.0000, 433070.0000)
Solving linear system for model parameters...

Model Parameters (x1 to x5):
x1 (MYCT): -0.0077
x2 (MMIN): 0.0209
x3 (MMAX): 0.0022
x4 (CACH): 0.8522
x5 (CHMAX): -0.0262

Making predictions on the test set...

Root Mean Square Error (RMSE) on Testing Set: 150.8333
Root Mean Square Error (RMSE) on Training Set: 44.6323
```

Listing 3.1: C++ program output: model parameters and RMSE values

3.3.1 Determined Model Parameters

The C++ program outputs the following model parameters:

```
Model Parameters (x1 to x5):
```

```
x1 (MYCT): -0.0077
x2 (MMIN): 0.0209
x3 (MMAX): 0.0022
x4 (CACH): 0.8522
x5 (CHMAX): -0.0262
```

These coefficients quantify the estimated linear impact of each feature on PRP. For instance, CACH has the largest positive coefficient, suggesting it's a strong positive predictor within this model. MYCT and CHMAX have small negative coefficients.

3.3.2 Performance Metrics (RMSE)

The RMSE values obtained from the C++ execution are:

```
Root Mean Square Error (RMSE) on Testing Set: 150.8333
Root Mean Square Error (RMSE) on Training Set: 44.6323
```

- The training RMSE of 44.63 indicates the model's average prediction error on the data it was trained on.
- The testing RMSE of 150.83 indicates the model's average prediction error on unseen data. This value is considerably higher than the training RMSE.

3.3.3 Discussion of Results

The significant discrepancy between the training RMSE and the testing RMSE suggests that the model might be overfitting the training data. While it has learned the patterns in the training set to achieve a relatively low error, it does not generalize as effectively to new, unseen data from the testing set. The absolute value of the testing RMSE (150.83) is also quite high relative to typical PRP values (which range from single digits to over 1000 in the dataset, with a mean around 100-200), indicating that the predictions can be substantially off.

This outcome could be due to several factors:

- The inherent complexity of CPU performance prediction may not be fully captured by a simple linear model based on these five features.
- The dataset size (209 instances, 167 for training) might be small for reliably training a model that generalizes well.
- The specific features chosen, while specified by the task, might not be the most optimal set, or interactions between features might be important but are not captured by this linear model.
- The lack of regularization might contribute to overfitting.

The successful application of the custom C++ classes to this regression problem demonstrates their utility. However, for a more robust predictive model in a real-world scenario, further investigation into feature engineering, model selection, and cross-validation would be necessary.

Chapter 4

Conclusion

This project successfully addressed the requirements outlined in the project specification, encompassing both the development of a foundational C++ library for linear algebra and its application to a linear regression task.

4.1 Summary of Achievements

- **Part A: Core Linear Algebra Classes:**
  - A robust `Vector` class was implemented, featuring dynamic memory management, overloaded operators for standard vector arithmetic (including scalar operations), and both 0-based and 1-based indexing with bounds checking.
  - A comprehensive `Matrix` class was developed, providing functionalities for memory allocation, various constructors (including a deep copy constructor), operator overloading for matrix arithmetic (matrix-matrix, matrix-vector, scalar-matrix). Crucially, methods for calculating the determinant, inverse (for square matrices), and Moore-Penrose pseudo-inverse (for general matrices) were implemented, addressing the need to handle both well-posed and ill-posed systems, including non-square ones.
  - A `LinearSystem` class was created to solve $Ax = b$ for square matrices using Gaussian elimination with partial pivoting. Proper attention was given to class design, including constructors and memory management considerations.
  - A derived class, `PosSymLinSystem`, was implemented to efficiently solve symmetric positive definite systems using the Conjugate Gradient method, showcasing polymorphism by overriding the virtual `Solve` method. This class also includes a check for matrix symmetry.
- **Part B: Linear Regression Application:**
  - The implemented classes were successfully utilized to perform linear regression on the UCI Computer Hardware dataset.
  - The task involved selecting appropriate features as per the problem description, splitting the data into training (80%) and testing (20%) sets, and formulating the problem using normal

equations.

- The `LinearSystem` class was used to solve these normal equations to determine the model parameters.
- The model's performance was evaluated using Root Mean Square Error (RMSE) on both training and testing sets, providing insights into the model's fit and generalization capabilities. The results showed a notable difference between training and testing RMSE, suggesting potential overfitting or limitations of the simple linear model for this dataset.

4.2 Challenges and Learning Outcomes

The project provided substantial learning opportunities in several areas of C++ programming and numerical methods:

- **Object-Oriented Design:** Designing and implementing interconnected classes (`Vector`, `Matrix`, `LinearSystem`, `PosSymLinSystem`) with appropriate encapsulation, inheritance, and polymorphism.
- **Memory Management:** Gaining hands-on experience with dynamic memory allocation (`new`, `delete`, `delete[]`) and the importance of constructors, destructors, and copy semantics (Rule of Three/Five).
- **Operator Overloading:** Implementing intuitive interfaces for mathematical objects through operator overloading.
- **Numerical Algorithms:** Implementing core numerical linear algebra algorithms such as Gaussian elimination with pivoting, the Conjugate Gradient method, and methods for matrix determinant, inverse, and pseudo-inverse. This involved understanding their mathematical basis and considerations for numerical stability.
- **Application to Machine Learning:** Bridging the gap between a custom numerical library and a practical data analysis task, reinforcing the understanding of how linear algebra underpins machine learning techniques like linear regression.

A key challenge was ensuring numerical stability and correctness in the implemented algorithms, particularly for matrix inversion and system solving. Debugging memory management issues and ensuring correct operator behavior also required careful attention.

4.3 Potential Future Work

The current project forms a strong basis for further development. Potential enhancements could include:

- **Advanced Numerical Techniques:** Incorporating more sophisticated matrix decompositions for solving linear systems and other computations, which can offer better stability or efficiency for certain types of matrices.

- **Sparse Matrix Support:** Extending the library to handle sparse matrices efficiently using specialized storage formats and algorithms.
- **Templated Classes:** Converting `Vector` and `Matrix` to class templates to support different numerical types.
- **Enhanced Error Handling:** Replacing assertions with a more robust exception handling mechanism for runtime errors.
- **Regularization Techniques:** Explicitly implementing regularization methods like Tikhonov regularization or Ridge regression within the linear system solvers or as part of the linear regression application framework.
- **Broader Machine Learning Applications:** Extending the Part B application to include more comprehensive model evaluation, feature scaling, or comparison with other regression models.

To sum up, this project has successfully demonstrated the ability to design, implement, and apply a C++ library for fundamental linear algebra operations. The work fulfills the specified requirements and provides a valuable educational experience in software development and numerical computation.