VIETNAMESE - GERMAN UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Programming 2 - Tiny Project Report

Student 1: Dao Hoang Minh Triet - 10423179

HO CHI MINH CITY, 06/ 2025

Content

1 Part A Linear System Implementation

1.1 Introduction

We are creating a C++ library that can represent and solve systems of linear equations of the form Ax = b, where A is a matrix of coefficients and b is a vector of constants. The solution x is the vector of unknowns. The system may be square (number of equations equals number of unknowns), overdetermined, or underdetermined. Additionally, the project requires handling special cases such as symmetric positive definite matrices.

1.2 Method

1.2.1 Class Design Over

The system is organized into four main classes to support numerical linear algebra and system solving in a structured and extensible way.

- The design separates low-level data representation (vectors and matrices) from high-level solving strategies. This improves modularity and enables future extensions with minimal code duplication.
- `Vector` and `Matrix` classes provide fundamental data structures with dynamic memory management, arithmetic operations, and transformation utilities. These are used as building blocks for all higher-level computations.
- The `LinearSystem` class encapsulates a system of equations $\mathbf{Ax} = \mathbf{b}$ and provides a virtual `Solve()` interface. It determines whether to use matrix inversion or pseudo-inverse based on the system structure.
- `PosSymLinSystem` extends `LinearSystem` and introduces an efficient solver based on the Conjugate Gradient method, specialized for symmetric positive definite matrices.
- This architecture promotes clean abstraction, reusability, and efficient specialization. Each class maintains clear responsibilities, and polymorphism is used to allow specialized behavior without altering the base solver logic.

1.2.2 Vector Class

Role: Represents a 1-dimensional dynamic array of real numbers, supporting core linear algebra operations and memory management.

Responsibilities:

- Manages its own dynamic memory for storing elements.
- Provides arithmetic operations: element-wise addition, subtraction, and scalar multiplication/division.

- Supports indexed access via both bracket and function-style operators, including const and non-const overloads.
- Implements deep copy semantics through copy constructor and assignment.
- Exposes a method to retrieve the size of the vector.

1.2.3 Matrix Class

Role: Encapsulates a 2D dynamic matrix with full support for matrix algebra, transformations, and structural property checks.

Responsibilities:

- Allocates and deallocates a 2D dynamic array.
- Implements basic operations: matrix addition, subtraction, scalar multiplication/division, and matrix multiplication.
- Supports matrix-vector multiplication.
- Provides element access methods and utilities for matrix shape queries.
- Offers transformation functions such as transpose and inverse.
- Computes determinant and pseudo-inverse.
- Checks structural properties like symmetry.

1.2.4 LinearSystem Class

Role: Defines a general linear system of the form $\mathbf{Ax} = \mathbf{b}$ and provides a standard interface for solving it.

Responsibilities:

- Stores the coefficient matrix $\mathbf{A}$ and the right-hand side vector $\mathbf{b}$.
- Provides a virtual method `Solve()` for computing the solution vector $\mathbf{x}$.
- Applies inverse or pseudo-inverse depending on whether $\mathbf{A}$ is square.
- Serves as a base class for specialized linear solvers.

1.2.5 PosSymLinSystem Class

Role: A subclass of `LinearSystem` optimized for symmetric positive definite systems.

Responsibilities:

- Validates that the input matrix is symmetric and positive definite upon construction.
- Overrides the `Solve()` method to use the Conjugate Gradient method for efficient iterative solution.

1.3 Implementation

Listing 1: Vector.h code

```
class Vector {
    private:
        int mSize;
        double* mData;
    public:
        Vector(int size);
        Vector(const Vector& other);
        ~Vector();

        Vector& operator=(const Vector& other);
        Vector operator+(const Vector& other) const;
        Vector operator-(const Vector& other) const;
        Vector operator*(double scalar) const;
        Vector operator*(const Vector& other) const;
        Vector operator/(double scalar) const;
        double& operator[](int index);
        double operator()(int index) const;
        double& operator()(int index);
        int size() const;
};
```

Listing 2: Matrix.h code

```
#include "Vector.h"
class Matrix {
    private:
        int mNumRows;
        int mNumCols;
        double** mData;

        void allocateMemory();
        void freeMemory();


    public:
        Matrix(int rows, int cols);
        Matrix(const Matrix& other);
        ~Matrix();

        Matrix& operator=(const Matrix& other);
        Matrix operator+(const Matrix& other) const;
```

```
        Matrix operator-(const Matrix& other) const;
        Matrix operator*(const Matrix& other) const;
        Vector operator*(Vector& other) const;
        Matrix operator*(double scalar) const;
        Matrix operator/(double scalar) const;
        double& operator()(int row, int col);
        double operator()(int row, int col) const;
        int rows() const;
        int cols() const;

        double Determinant() const;
        Matrix Transpose() const;
        Matrix Inverse() const;
        Matrix PseudoInverse() const;
        bool IsSymmetric() const;
};
```

Listing 3: LinearSystem.h code

```
#include "Matrix.h"
#include "Vector.h"

class LinearSystem {
protected:
    int mSize;
    Matrix* mpA = nullptr;
    Vector* mpb = nullptr;

public:
    LinearSystem(Matrix& A, Vector& b);
    virtual ~LinearSystem();
    virtual Vector Solve() const;
};

class PosSymLinSystem : public LinearSystem {
public:
    PosSymLinSystem(Matrix& A, Vector& b);
    Vector Solve() const override;
};
```

1.4 Demo Result

1.4.1 Demo

The `main()` function in `LinearSystem.cpp` demonstrates:

- Solving a square system using matrix inversion.
- Solving a non-square system using the pseudo-inverse.
- Solving a symmetric positive definite system using the Conjugate Gradient method.

Listing 4: main() function code in LinearSystem.cpp

```
int main() {
    // Example usage of LinearSystem with a square matrix
    cout << "Solving a square linear system..." << endl;
    // Define a square matrix and a vector
    Matrix A(3, 3);
    Vector b(3);

    A(1,1) = 4; A(1,2) = 1; A(1,3) = 10;
    A(2,1) = 12; A(2,2) = 3; A(2,3) = 0;
    A(3,1) = 2; A(3,2) = 7; A(3,3) = 5;

    b(1) = 7;
    b(2) = 8;
    b(3) = 9;
    // Create an instance of LinearSystem
    LinearSystem system(A, b);
    try {
        // Solve the system
        Vector solution = system.Solve();
        // Output the solution
        for (int i = 1; i <= solution.size(); ++i) {
            cout << "x[" << i << "] = " << solution(i) << std::endl;
        }
    } catch (const std::exception& e) {
        cout << "Error: " << e.what() << std::endl;
    }


    // Example usage of LinearSystem with a non-square matrix
    cout << "Solving a not square linear system..." << endl;
    // Define a non-square matrix and a vector
    Matrix A2(3, 2);
    Vector b2(3);
```

```
34      A2(1,1) = 1; A2(1,2) = 2;
35      A2(2,1) = 3; A2(2,2) = 4;
36      A2(3,1) = 5; A2(3,2) = 6;
37      b2(1) = 7;
38      b2(2) = 8;
39      b2(3) = 9;
40      // Create an instance of LinearSystem
41      // Note: This will use the pseudo-inverse method since A is not ↩
            square
42      LinearSystem system2(A2, b2);
43      try {
44          // Solve the system
45          Vector solution2 = system2.Solve();
46          // Output the solution
47          for (int i = 1; i <= solution2.size(); ++i) {
48              cout << "x[" << i << "] = " << solution2(i) << std::endl;
49          }
50      } catch (const std::exception& e) {
51          cout << "Error: " << e.what() << std::endl;
52      }
53
54      // Example usage of PosSymLinSystem
55      cout << "Solving a positive definite symmetric linear system..." ↩
            << endl;
56      // Define a positive definite symmetric matrix and a vector
57      Matrix A3(3, 3);
58      Vector b3(3);
59      A3(1,1) = 4; A3(1,2) = 1; A3(1,3) = 2;
60      A3(2,1) = 1; A3(2,2) = 3; A3(2,3) = 0;
61      A3(3,1) = 2; A3(3,2) = 0; A3(3,3) = 5;
62      b3(1) = 7;
63      b3(2) = 8;
64      b3(3) = 9;
65      // Create an instance of PosSymLinSystem
66      PosSymLinSystem posSystem(A3, b3);
67      try {
68          // Solve the system
69          Vector posSolution = posSystem.Solve();
70          // Output the solution
71          for (int i = 1; i <= posSolution.size(); ++i) {
72              cout << "x[" << i << "] = " << posSolution(i) << std::endl↩
                    ;
73          }
74      } catch (const std::exception& e) {
75          cout << "Error: " << e.what() << std::endl;
```

```
76      }
77
78      return 0;
79  }
```

1.4.2 Sample Output

The program prints the solution vector for each system, showing that the implementation works for different cases.

Listing 5: Output of the above main() function

```
1  Solving a square linear system...
2  x[1] = 0.455128
3  x[2] = 0.846154
4  x[3] = 0.433333
5  Solving a not square linear system...
6  x[1] = -6
7  x[2] = 6.5
8  Solving a positive definite symmetric linear system...
9  x[1] = 0.255814
10 x[2] = 2.5814
11 x[3] = 1.69767
```

1.5 Conclusion

The class architecture demonstrates a well-structured object-oriented design tailored for linear system solving. By separating vector and matrix operations from solver logic, the implementation promotes code reusability, abstraction, and maintainability. The use of inheritance and polymorphism enables specialized solvers—like the Conjugate Gradient method—to integrate seamlessly without altering the base structure. Overall, the design balances computational efficiency with extensibility, making it suitable for expanding to more advanced numerical methods in future work.

2 Part B: Linear Regression

2.1 Introduction

We are implementing a linear regression model to predict computer system performance based on hardware features. The goal is to fit a linear model to a dataset and evaluate its predictive accuracy.

2.2 Dataset

The dataset used is the **Relative CPU Performance Data**, originally compiled by Phillip Ein-Dor and Jacob Feldmesser and donated by David W. Aha. It contains measurements and characteristics of various computer systems, with the goal of predicting relative CPU performance.

- **Key Details**
  - **Number of Instances:** 209 computer systems.
  - **Number of Attributes:** 10 per instance:
    - 6 predictive attributes (hardware features)
    - 2 non-predictive attributes (vendor and model name)
    - 1 goal field (published relative performance, PRP)
    - 1 linear regression estimate (ERP)
- **Attributes**
  1. **vendor name** (categorical): 30 possible values (e.g., ibm, hp, dec, etc.)
  2. **model name** (categorical): many unique values
  3. **MYCT**: machine cycle time in nanoseconds (integer)
  4. **MMIN**: minimum main memory in kilobytes (integer)
  5. **MMAX**: maximum main memory in kilobytes (integer)
  6. **CACH**: cache memory in kilobytes (integer)
  7. **CHMIN**: minimum number of channels (integer)
  8. **CHMAX**: maximum number of channels (integer)
  9. **PRP**: published relative performance (integer, target variable)
  10. **ERP**: estimated relative performance from the original article (integer)
- **Statistics**

  Table 1 presents the distribution of PRP values across the dataset, while Table 2 provides descriptive statistics and correlations for each attribute. As shown in Table 1, the majority of instances fall within the 21–100 range, revealing a strong class imbalance that could affect model performance. Table 2 highlights that `MMAX` (maximum main memory) has the

highest positive correlation with PRP ($r = 0.8630$), suggesting it is a strong predictor. In contrast, `MYCT` (cycle time) is negatively correlated with PRP ($r = -0.3071$), implying that shorter cycle times are generally associated with better performance. These statistical insights are valuable for guiding feature selection and improving the effectiveness of the regression model.

PRP Value Range	Instances
0–20	31
21–100	121
101–200	27
201–300	13
301–400	7
401–500	4
501–600	2
Above 600	4

Table 1: Class Distribution of PRP Values

Attr.	Min	Max	Mean	SD	Corr.
MYCT	17	1500	203.8	260.3	-0.3071
MMIN	64	32000	2868.0	3878.7	0.7949
MMAX	64	64000	11796.1	11726.6	0.8630
CACH	0	256	25.2	40.6	0.6626
CHMIN	0	52	4.7	6.8	0.6089
CHMAX	0	176	18.2	26.0	0.6052
PRP	6	1150	105.6	160.8	1.0000
ERP	15	1238	99.3	154.8	0.9665

Table 2: Summary Statistics of Attributes

2.3 Method

2.3.1 Data Loading (`readData`)

Purpose: Read numeric features from `machine.txt`, normalize them, and add to the model.

Process:

- Extracts columns 2 to 8 (7 values: 6 features + 1 ground truth).
- Stores each sample as an integer vector of length 7.
- Computes feature-wise min and max for normalization.
- Applies min-max normalization on the first 6 features:

$$x_i' = \frac{x_i - \min(x_i)}{\max(x_i) - \min(x_i)}$$

- Appends the normalized feature vector to the model.

2.3.2 Train/Test Split (`splitData`)

Purpose: Randomly partition the dataset into training and test sets.

Process:

- Randomly shuffles all data samples.
- Splits the first 80% into training set and the rest into test set.

2.3.3 Training (`train`)

Purpose: Optimize weights using mini-batch gradient descent.

Process:

- Weights are initialized with random values between 0 and 399.
- For each epoch:
  - Shuffle the training set.
  - Iterate in batches of size B (default: 5).
  - For each batch:
    - Predict output: $\hat{y} = \sum_{i=1}^{6} w_i x_i + b$
    - Compute error: $e = \hat{y} - y$
    - Accumulate gradients:

$$\nabla w_i \mathrel{+}= e \cdot x_i, \quad \nabla b \mathrel{+}= e$$

    - Update parameters:

$$w_i \mathrel{-}= \eta \cdot \frac{\nabla w_i}{B}, \quad b \mathrel{-}= \eta \cdot \frac{\nabla b}{B}$$

- Every 1000 epochs and at the final epoch, print RMSE on both train and test sets.

2.3.4 Loss Function (`computeRMSE`)

Purpose: Quantify prediction accuracy using Root Mean Squared Error.

$$\text{RMSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2}$$

2.3.5 Prediction (`predict`)

Purpose: Compute model output for a single sample.

$$\hat{y} = b + \sum_{i=1}^{6} w_i x_i$$

2.3.6 Inference (`infer`)

Purpose: Generate predictions on the test set and return them paired with ground truth.

Process:

- For each sample in the test set:
  - Predict output $\hat{y}$.
  - Store the pair $(\hat{y}, y_{\text{true}})$ for evaluation.

2.4 Experimental Result

2.4.1 Loss Summary

The training and testing RMSE values demonstrate the convergence behavior of the model during training (see Figure 1). Initially, both errors are high, with a training RMSE of 122.561 and a test RMSE of 199.868 at epoch 0. As training progresses, the RMSE decreases steadily.

By epoch 1000, the training RMSE drops below 60, while the test RMSE improves to approximately 101. After 2000 epochs, improvements become more gradual. The training loss continues to decline slightly, reaching 52.4018 by epoch 4999. In contrast, the test RMSE stabilizes around 93–94, showing minor fluctuations.

This pattern suggests that the model learns effectively from the training set and generalizes reasonably well to unseen data. However, the flattening of the test RMSE curve and its slight increase toward the end may indicate mild overfitting as training continues. Overall, the mini-batch gradient descent approach successfully reduces error, with test RMSE improving by over 100 points from the initial epoch.

2.4.2 Demo result

This is the predictions and ground truth of the first 10 samples in the test set (see Table 3).

2.5 Conclusion

The linear regression model trained using mini-batch gradient descent effectively reduced prediction error over time, as reflected in the declining RMSE values. The consistent decrease in training loss and the stable test loss suggest that the model learned general patterns from the data without significant overfitting. The modular implementation supports reproducible inference and allows further optimization, such as adaptive learning rates or regularization techniques, to improve generalization on unseen data.

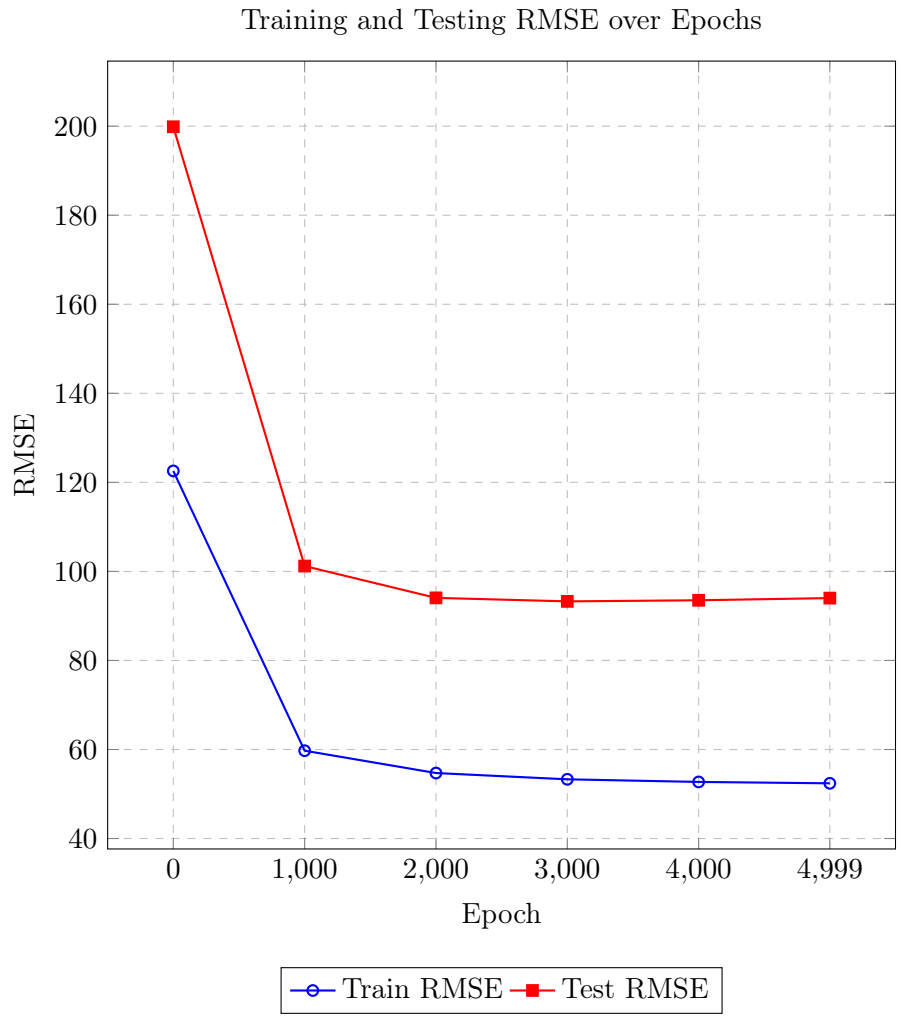

Figure 1: RMSE values of training and testing sets over training epochs

3 Source Code

The full implementation, including all class definitions, training logic, data processing routines, and result visualization, is available in the following GitHub repository:

– `https://github.com/daotriet05/Prog2_TinyProject`

This repository contains the complete source code for both Part A (object-oriented system design for linear system solving) and Part B (linear regression model with mini-batch gradient descent), along with input data, documentation, and sample outputs for reproducibility.

Sample	Predicted Value	Ground Truth
1	22.8797	42
2	418.723	405
3	20.3469	36
4	33.8709	50
5	106.677	66
6	119.315	113
7	442.144	367
8	76.7584	42
9	516.562	307
10	240.775	173

Table 3: Sample Inference Results (First 10 Samples)