Basic macros Compatibility with language.def Hooks Defining babelensure Macros for setting language files up Shorhands
Language attributes Macros for saving definitions Short tags Hyphens Multiencoding strings Macros related to glyphs Bidi layout Input engine specific macros
Redefinitions for bidi layout Bidi footnotes Hyphen rules for 'canadian' set to \l@english
(\language0). ReportedHyphen rules for 'australian' set to \l@ukenglish
(\language23). ReportedHyphen rules for 'newzealand' set to \l@ukenglish
(\language23). Reported
english

Undefined language '' in aux.

Reported

english

VIETNAMESE - GERMAN UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Programming 2 - Tiny Project

Link to my Project's Github Repository

Lecturer:	Dr. Huynh Trung Hieu
Student:	Tran Trong Nhan - 10423160

Ben Cat, Binh Duong City, 05/ 2025

Content

1 Part A **2**
1.1 Introduction . . . 2
1.2 Classes . . . 2
1.2.1 Vector Class . . . 2
1.2.2 Matrix Class . . . 2
1.2.3 LinearSystem Class . . . 3
1.2.4 PosSysLinSystem Class . . . 3
1.3 Implementation . . . 3
1.4 Demo Result . . . 5
1.5 Conclusion . . . 6

2 Part B **7**
2.1 Introduction . . . 7
2.2 Dataset . . . 7
2.3 Regression Model . . . 7
2.4 Implementation . . . 8
2.4.1 Data Loading . . . 8
2.5 Implementation . . . 8
2.5.1 Data Loading (`readData`) . . . 8
2.5.2 Analysis of Dataset . . . 8
2.5.3 Splitting the Dataset (`splitData`) . . . 8
2.5.4 Model Training . . . 8
2.5.5 Evaluation . . . 9
2.5.6 Inference . . . 9
2.6 Result . . . 9
2.7 Conclusion . . . 10

3 Conclusion **10**

1 Part A

1.1 Introduction

In Part A, we developed foundational classes for numerical computation: **Vector, Matrix,** and **LinearSystem.** These classes provide the core operations required for implementing numerical linear algebra solutions.

1.2 Classes

1.2.1 Vector Class

Purpose: Represents a one-dimensional array (vector) with support for arithmetic operations and memory management.
Key Methods:

- `Vector(int size)`: Constructor to initialize the vector with zero values.
- `Vector(const Vector& other)`: Copy constructor implementing deep copy.
- `Vector()`: Destructor that deallocates memory.
- `operator+`, `operator-`, `operator*`: Overloaded operators for vector addition, subtraction, scalar multiplication, and dot product.
- `operator[]` (0-based) and `operator()` (1-based): Indexed access to vector entries.
- `size()`: Returns the size of the vector.

1.2.2 Matrix Class

Purpose: Encapsulates 2D matrices with operator overloading for mathematical operations.
Key Methods:

- `Matrix(int rows, int cols)`: Constructor that initialzies a matrix with given dimensions.
- `Matrix(const Matrix other)`: Deep copy constructor.
- `Matrix()`: Destructor for safe memory management.
- `operator+`, `operator-`, `operator*`: Handles matrix arithmetic and matrix-vector multiplication.
- `operator()` (1-based): Accesses elements in matrix.
- `numRows()` and `numCols()`: Return matrix dimensions.

1.2.3 LinearSystem Class

Purpose: Represents and solves systems $Ax = b$ using Gaussian elimination.
Key Methods:

- `LinearSystem(const Matrix& A, const Vector& b)`: Constructor that validates and copies input matrix and vector.
- `Solve()`: Implements Gaussian elimination with partial pivoting and back-substitution.

1.2.4 PosSysLinSystem Class

Purpose: Inherits from `LinearSystem` and specializes in solving SPD systems.
Key Methods:

- `PosSymLinSystem(const Matrix& A, const Vector& b)`: Validates if matrix is symmetric.
- `Solve()`: Uses Conjugate Gradient method to iteratively compute the solution.

1.3 Implementation

Listing 1: Vector.h

```
#ifndef VECTOR_H
#define VECTOR_H

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;
    double operator*(const Vector& other) const; // Dot product

    double& operator[](int index);           // Zero-based indexing
    double operator()(int index) const;      // One-based indexing

    int size() const;
    void print() const;
};

#endif
```

Listing 2: Matrix.h

```
#ifndef MATRIX_H
#define MATRIX_H

#include "Vector.h"

class Matrix {
private:
    int mNumRows, mNumCols;
    double** mData;

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*(const Vector& vec) const;
    Matrix operator*(double scalar) const;

    double& operator()(int i, int j);       // 1-based
    double operator()(int i, int j) const;
    int numRows() const;
    int numCols() const;

    double determinant() const;
    Matrix inverse() const;
    Matrix pseudoInverse() const;

    void print() const;
};

#endif
```

Listing 3: LinearSystem.h

```cpp
#ifndef LINEARSYSTEM_H
#define LINEARSYSTEM_H

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

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

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

    virtual Vector Solve(); // Gaussian elimination
};

class PosSymLinSystem : public LinearSystem {
public:
    PosSymLinSystem(const Matrix& A, const Vector& b);
    Vector Solve() override; // Conjugate Gradient method
};

#endif
```

1.4 Demo Result

Listing 4: main.cpp

```cpp
#include <iostream>
#include "Vector.h"
#include "Matrix.h"
#include "LinearSystem.h"

using namespace std;

int main() {
    // Test 1: Solve Ax = b using Gaussian Elimination
    cout << "=== LinearSystem: Gaussian Elimination ===" << endl;
    Matrix A(3, 3);
    A(1,1) = 2; A(1,2) = -1; A(1,3) = 0;
    A(2,1) = -1; A(2,2) = 2; A(2,3) = -1;
    A(3,1) = 0; A(3,2) = -1; A(3,3) = 2;

    Vector b(3);
    b[0] = 1; b[1] = 0; b[2] = 1;

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

    cout << "Solution x:" << endl;
    x.print();

    // Test 2: Solve symmetric system using Conjugate Gradient
    cout << "\n=== PosSymLinSystem: Conjugate Gradient ===" << endl;
    Matrix S(3, 3);
    S(1,1) = 4; S(1,2) = 1; S(1,3) = 1;
    S(2,1) = 1; S(2,2) = 3; S(2,3) = 0;
    S(3,1) = 1; S(3,2) = 0; S(3,3) = 2;

    Vector b2(3);
    b2[0] = 1; b2[1] = 2; b2[2] = 3;

    PosSymLinSystem psys(S, b2);
    Vector x2 = psys.Solve();

    cout << "Solution x:" << endl;
    x2.print();

    return 0;
}
```

1.5 Conclusion

In Part A, we successfully designed and implemented a modular C++ system for solving linear systems of equations using object-oriented principles. The `Vector` and `Matrix` classes provided robust data structures with memory management and arithmetic capabilities. The `LinearSystem` class enabled the solution of square systems using Gaussian elimination, while the `PosSymLinSystem` class specialized in handling symmetric positive definite matrices via the Conjugate Gradient method. The implementation emphasizes clean abstraction, code reuse, and extensibility, laying a solid foundation for integrating more advanced numerical methods in the future.

2 Part B

2.1 Introduction

In Part B, we developed a linear regression model using C++ to predict the relative CPU performance (PRP) of computer systems based on hardware specifications. The model assumes a linear relationship between the target variable PRP and six predictive features from the dataset.

2.2 Dataset

The dataset is obtained from the UCI Machine Learning Repository and contains 209 instances. Each instance has 10 attributes, including:

- 2 non-predictive: vendor name, model name
- 6 predictive: MYCT, MMIN, MMAX, CACH, CHMIN, CHMAX
- 1 target: PRP (published relative performance)
- 1 unused: ERP (estimated relative performance)

Attributes:

- **vendor name** (categorical): 30 possible values (e.g., ibm, hp, dec, etc.)
- **model name** (categorical): many unique values
- **MYCT**: machine cycle time in nanoseconds (integer)
- **MMIN**: minimum main memory in kilobytes (integer)
- **MMAX**: maximum main memory in kilobytes (integer)
- **CHMIN**: minimum number of channels (integer)
- **CHMAX**: maximum number of channels (integer)
- **PRP**: published relative performance (integer, target variable)
- **ERP**: estimated relative performance from the original article (integer)

For the regression model, only the six numeric predictive features and the target PRP are used. All features are normalized using min-max scaling.

2.3 Regression Model

The regression model assumes the form:

$$\text{PRP} = x_1 \cdot \text{MYCT} + x_2 \cdot \text{MMIN} + x_3 \cdot \text{MMAX} + x_4 \cdot \text{CACH} + x_5 \cdot \text{CHMIN} + x_6 \cdot \text{CHMAX} + b$$

Where $x_1, ..., x_6$ are the model weights and b is the bias.

2.4 Implementation

2.4.1 Data Loading

2.5 Implementation

2.5.1 Data Loading (`readData`)

The `readData()` function loads data from `machine.data`, extracting features 3–8 (indices 2–7 in 0-based indexing) and the target PRP (index 8). Min-max normalization is applied to the six features. The target is left as raw.

2.5.2 Analysis of Dataset

The dataset contains **209 instances** of computer systems with numerical features describing their hardware and a performance metric called PRP (published relative performance).

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

2.5.3 Splitting the Dataset (`splitData`)

The dataset is shuffled randomly and split into: 80% for training, and 20% for testing.

2.5.4 Model Training

The model uses mini-batch gradient descent with the following characteristics:

- Random initialization of weights and bias
- Gradients are computed from each mini-batch
- Parameters updated using a fixed learning rate
- RMSE printed periodically

2.5.5 Evaluation

Root Mean Square Error (RMSE) is used as the evaluation metric:

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

It is calculated on both training and testing sets after each epoch.

2.5.6 Inference

The `infer()` function returns a list of predictions versus ground truth values from the test set for error inspection.

2.6 Result

Sample training progress:

```
Epoch 0 | Train RMSE: 127.789 | Test RMSE: 186.355
Epoch 500 | Train RMSE: 70.2716 | Test RMSE: 113.624
Epoch 1000 | Train RMSE: 59.06 | Test RMSE: 98.9325
...
Epoch 4999 | Train RMSE: 52.2805 | Test RMSE: 91.0225
...
Final weights: 66.1917 390.755 335.198 243.662 90.0289 131.095
```

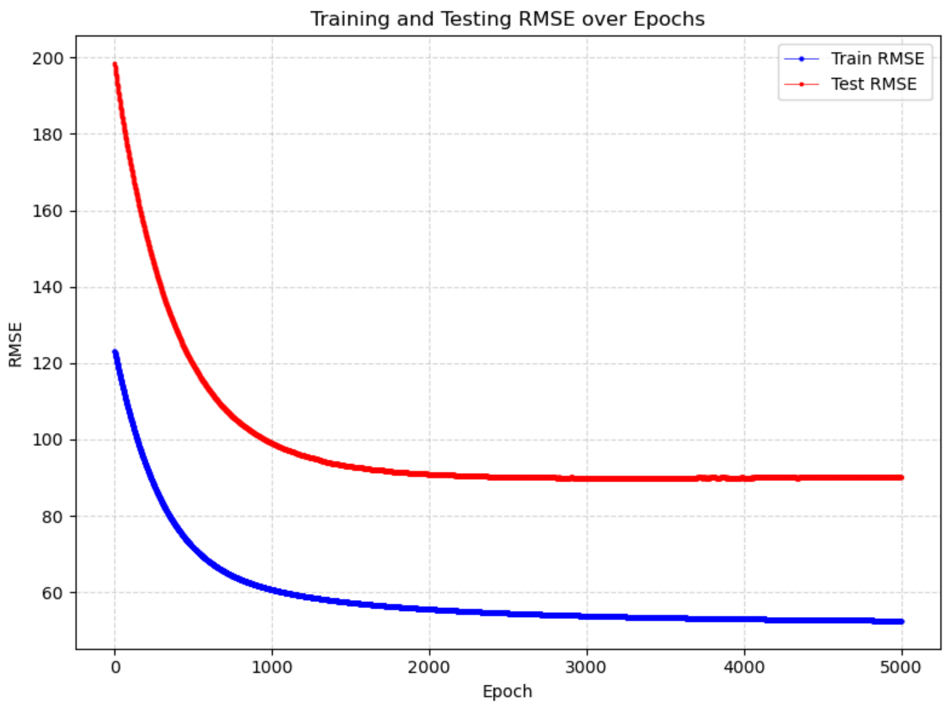

Figure 1: Training and Testing RMSE over Epochs

2.7 Conclusion

In Part B, we implemented a linear regression model to predict the published relative performance (PRP) of computer systems based on six hardware-related features. The model was trained using gradient descent with batch updates and evaluated using Root Mean Square Error (RMSE) on both training and testing datasets.

Our analysis showed that features such as maximum memory (MMAX), minimum memory (MMIN), and cache size (CACH) have strong positive correlations with system performance, while machine cycle time (MYCT) correlates negatively, as expected. After normalization and training, the model achieved a decreasing RMSE over time, indicating convergence and learning.

Although the linear regression model captures the general trends in the dataset, the residual variance suggests that additional non-linear factors or interaction terms might be required for higher accuracy. Nonetheless, the results confirm that a basic linear model is a reasonable baseline for estimating relative CPU performance using hardware attributes.

3 Conclusion

This project demonstrates how low-level vector/ matrix operations (*Part A*) can be composed to built a fully functional linear regression model (*Part B*).
Implementing and training the model from scratch with gradient descent teaches critical understanding of:

- Data normalization
- Numerical stability
- RMSE-based evaluation
- Algorithmic implementation of linear algebra concepts

The final result is complete in C++ pipeline from raw data to model training and evaluation with support for visualization using external tools.
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/youngazier/programming2-tiny-project