VIETNAMESE - GERMAN UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

Introduction to Tiny Project

Student 1: Nguyen Huu Tri - 10423178
Student 2: Bui Vinh An - 10423127
Student 3: Vuong Thai Thinh - 10423106

BÌNH DƯƠNG, 06/ 2025

Content

1 Introduction

This project consists of two independent parts:

Part A is a software engineering task focusing on the development of a linear algebra library in C++. It aims to build essential classes like Vector, Matrix, and LinearSystem, implementing numerical methods such as Gaussian Elimination and the Conjugate Gradient method. This part is centered on algorithm design, object-oriented programming, and mathematical computation in C++.

Part B is a machine learning task involving linear regression using a real-world dataset. The dataset contains hardware specifications of computer systems and their corresponding published performance scores. The objective is to build a linear model to predict performance based on hardware attributes using least squares estimation. This part was also implemented in C++ and demonstrates how to apply mathematical models to real data.

2 Part A

2.1 Overview

Part A of the Tiny Project requires the development of a linear algebra library comprising three key classes: `Vector`, `Matrix`, and `LinearSystem` (with a derived `PosSymLinSystem`). These classes facilitate vector and matrix operations, solving square linear systems $Ax = b$ via Gaussian elimination, and solving symmetric positive definite systems using the conjugate gradient method. Additionally, the library supports handling under-determined and over-determined systems using the Moore-Penrose pseudo-inverse, with provisions for Tikhonov regularization to address ill-robust problems. The implementation is realized in C++ across header files (`Vector.h`, `Matrix.h`, `LinearSystem.h`) and test programs (`Vector.cpp`, `Matrix.cpp`, `LinearSystem.cpp`), adhering to strict memory management, operator overloading, and error handling requirements.

2.2 Vector Class Implementation

2.2.1 Purpose

The `Vector` class represents a mathematical vector with dynamic size, supporting operations like addition, subtraction, scalar multiplication, norm computation, and dot product. It features dual indexing (0-based via `operator[]`, 1-based via `operator()`) and robust memory management.

2.2.2 Vector Code

Key excerpts from `Vector.h`:

Listing 1: Vector.h

```
class Vector {
private:
```

```
    int mSize;       // Size of the vector
    double* mData;   // Pointer to the data array

public:
    // Constructors and destructor
    Vector(int size);                    // Constructor with size
    Vector(const Vector& other);         // Copy constructor
    ~Vector();                           // Destructor

    // Assignment operator
    Vector& operator=(const Vector& other);

    // Access operators
    double& operator[](int i);           // 0-based indexing with bounds checking
    const double& operator[](int i) const;
    double& operator()(int i);           // 1-based indexing
    const double& operator()(int i) const;

    // Unary operators
    Vector operator+() const;            // Unary plus
    Vector operator-() const;            // Unary minus

    // Binary operators
    Vector operator+(const Vector& other) const;  // Vector addition
    Vector operator-(const Vector& other) const;  // Vector subtraction
    Vector operator*(double scalar) const;        // Scalar multiplication
    friend Vector operator*(double scalar, const Vector& vec);  // Scalar multiplication (left)

    // Compound assignment operators
    Vector& operator+=(const Vector& other);
    Vector& operator-=(const Vector& other);
    Vector& operator*=(double scalar);

    // Utility functions
    int GetSize() const { return mSize; }
    double Norm() const;                 // Euclidean norm
    double DotProduct(const Vector& other) const;  // Dot product

    // Friend functions for I/O
    friend ostream& operator<<(ostream& os, const Vector& vec);
    friend istream& operator>>(istream& is, Vector& vec);
```

```cpp
};

// Constructor
Vector::Vector(int size) : mSize(size) {
    assert(size > 0);
    mData = new double[size];
    for (int i = 0; i < size; i++) {
        mData[i] = 0.0;
    }
}

// Copy constructor
Vector::Vector(const Vector& other) : mSize(other.mSize) {
    mData = new double[mSize];
    for (int i = 0; i < mSize; i++) {
        mData[i] = other.mData[i];
    }
}

// Destructor
Vector::~Vector() {
    delete[] mData;
}

// Assignment operator
Vector& Vector::operator=(const Vector& other) {
    if (this != &other) {
        delete[] mData;
        mSize = other.mSize;
        mData = new double[mSize];
        for (int i = 0; i < mSize; i++) {
            mData[i] = other.mData[i];
        }
    }
    return *this;
}

// Access operators
double& Vector::operator[](int i) {
    assert(i >= 0 && i < mSize);
    return mData[i];
}

const double& Vector::operator[](int i) const {
    assert(i >= 0 && i < mSize);
```

```
    return mData[i];
}

double& Vector::operator()(int i) {
    assert(i >= 1 && i <= mSize);
    return mData[i-1];
}

const double& Vector::operator()(int i) const {
    assert(i >= 1 && i <= mSize);
    return mData[i-1];
}

// Unary operators
Vector Vector::operator+() const {
    return *this;
}

Vector Vector::operator-() const {
    Vector result(mSize);
    for (int i = 0; i < mSize; i++) {
        result.mData[i] = -mData[i];
    }
    return result;
}

// Binary operators
Vector Vector::operator+(const Vector& other) const {
    assert(mSize == other.mSize);
    Vector result(mSize);
    for (int i = 0; i < mSize; i++) {
        result.mData[i] = mData[i] + other.mData[i];
    }
    return result;
}

Vector Vector::operator-(const Vector& other) const {
    assert(mSize == other.mSize);
    Vector result(mSize);
    for (int i = 0; i < mSize; i++) {
        result.mData[i] = mData[i] - other.mData[i];
    }
    return result;
}

```

```cpp
Vector Vector::operator*(double scalar) const {
    Vector result(mSize);
    for (int i = 0; i < mSize; i++) {
        result.mData[i] = mData[i] * scalar;
    }
    return result;
}

Vector operator*(double scalar, const Vector& vec) {
    return vec * scalar;
}

// Compound assignment operators
Vector& Vector::operator+=(const Vector& other) {
    assert(mSize == other.mSize);
    for (int i = 0; i < mSize; i++) {
        mData[i] += other.mData[i];
    }
    return *this;
}

Vector& Vector::operator-=(const Vector& other) {
    assert(mSize == other.mSize);
    for (int i = 0; i < mSize; i++) {
        mData[i] -= other.mData[i];
    }
    return *this;
}

Vector& Vector::operator*=(double scalar) {
    for (int i = 0; i < mSize; i++) {
        mData[i] *= scalar;
    }
    return *this;
}

// Utility functions
double Vector::Norm() const {
    double sum = 0.0;
    for (int i = 0; i < mSize; i++) {
        sum += mData[i] * mData[i];
    }
    return sqrt(sum);
}

```

```
double Vector::DotProduct(const Vector& other) const {
    assert(mSize == other.mSize);
    double sum = 0.0;
    for (int i = 0; i < mSize; i++) {
        sum += mData[i] * other.mData[i];
    }
    return sum;
}

// I/O operators
ostream& operator<<(ostream& os, const Vector& vec) {
    os << "[";
    for (int i = 0; i < vec.mSize; i++) {
        os << vec.mData[i];
        if (i < vec.mSize - 1) os << ", ";
    }
    os << "]";
    return os;
}

istream& operator>>(istream& is, Vector& vec) {
    for (int i = 0; i < vec.mSize; i++) {
        is >> vec.mData[i];
    }
    return is;
}

#endif // VECTOR_H
```

2.2.3 Functionality

- **Memory Management**:
  - `Vector(int size)`: Allocates a vector of size `size`, initializes elements to 0, and checks `size > 0`.
  - `Vector(const Vector&)`: Performs deep copy to avoid aliasing.
  - `Vector()`: Frees allocated memory.
  - `operator=`: Implements deep copy with self-assignment check.
- **Indexing**:
  - `operator[]`: Provides 0-based access with bounds checking (`i >= 0 && i < mSize`).
  - `operator()`: Provides 1-based access, mapping `i` to `mData[i-1]`.
- **Arithmetic Operators**:

  - `operator+`: Adds vectors element-wise, ensuring equal sizes. E.g., $[1, 2]+[3, 4] = [4, 6]$.
  - `operator-`: Subtracts vectors.
  - `operator*`: Scales by a scalar (left/right). E.g., $[1, 2] * 2 = [2, 4]$.
  - `operator+=`, `-=`, `*=`: Perform in-place operations.

- **Utility Functions**:
  - `Norm()`: Computes Euclidean norm, $\sqrt{\sum x_i^2}$. E.g., $[3, 4].Norm() = 5$.
  - `DotProduct()`: Computes $\sum x_i y_i$. E.g., $[1, 2] \cdot [3, 4] = 11$.

- **I/O**:
  - `operator«`: Outputs vector as `[x1, x2, ...]`.
  - `operator»`: Reads `mSize` elements.

2.2.4 Testing

The `main` function in `Vector.cpp` tests:

- Input of two vectors A, B (size 4).
- Operations: $A + B$, $A - B$, $A * 2$, $A* = 2$, `Norm()`, `DotProduct()`.

Example input: $A = [1, 2, 3, 4]$, $B = [5, 6, 7, 8]$:

$$A + B = [6, 8, 10, 12], \quad A * 2 = [2, 4, 6, 8], \quad A.Norm() = \sqrt{30}, \quad A \cdot B = 70$$

2.3 Matrix Class Implementation

2.3.1 Purpose

The `Matrix` class represents a 2D matrix with dynamic dimensions, supporting matrix arithmetic, matrix-vector multiplication, and advanced operations like determinant, inverse, and Moore-Penrose pseudo-inverse. It uses 1-based indexing via `operator()` and integrates with the `Vector` class.

2.3.2 Code Excerpt

Key excerpts from `Matrix.h`:

Listing 2: Matrix.h

```
#ifndef MATRIX_H
#define MATRIX_H

#include "Vector.h"

```

```cpp
class Matrix {
private:
    int mNumRows;      // Number of rows
    int mNumCols;      // Number of columns
    double** mData;    // Pointer to array of pointers (2D array)

public:
    // Constructors and destructor
    Matrix(int numRows, int numCols);      // Constructor with dimensions
    Matrix(const Matrix& other);           // Copy constructor
    ~Matrix();                            // Destructor

    // Assignment operator
    Matrix& operator=(const Matrix& other);

    // Access operators
    double& operator()(int i, int j);      // 1-based indexing
    const double& operator()(int i, int j) const;

    // Unary operators
    Matrix operator+() const;              // Unary plus
    Matrix operator-() const;              // Unary minus

    // Binary operators
    Matrix operator+(const Matrix& other) const;  // Matrix addition
    Matrix operator-(const Matrix& other) const;  // Matrix subtraction
    Matrix operator*(const Matrix& other) const;  // Matrix multiplication
    Matrix operator*(double scalar) const;        // Scalar multiplication
    Vector operator*(const Vector& vec) const;    // Matrix-vector multiplication
    friend Matrix operator*(double scalar, const Matrix& mat);  // Scalar multiplication (left)

    // Compound assignment operators
    Matrix& operator+=(const Matrix& other);
    Matrix& operator-=(const Matrix& other);
    Matrix& operator*=(double scalar);

    // Utility functions
    int GetNumRows() const { return mNumRows; }
    int GetNumCols() const { return mNumCols; }
```

```
    bool IsSquare() const { return mNumRows == mNumCols; }
    bool IsSymmetric() const;            // Check if matrix is symmetric
    double Determinant() const;          // Calculate determinant
    Matrix Inverse() const;              // Calculate inverse
    Matrix PseudoInverse() const;        // Calculate Moore-Penrose pseudo-inverse
    Matrix Transpose() const;            // Calculate transpose

    // Friend functions for I/O
    friend ostream& operator<<(ostream& os, const Matrix& mat);
    friend istream& operator>>(istream& is, Matrix& mat);
};

// Constructor
Matrix::Matrix(int numRows, int numCols) : mNumRows(numRows), mNumCols(numCols) {
    assert(numRows > 0 && numCols > 0);
    mData = new double*[numRows];
    for (int i = 0; i < numRows; i++) {
        mData[i] = new double[numCols];
        for (int j = 0; j < numCols; j++) {
            mData[i][j] = 0.0;
        }
    }
}

// Copy constructor
Matrix::Matrix(const Matrix& other) : mNumRows(other.mNumRows), mNumCols(other.mNumCols) {
    mData = new double*[mNumRows];
    for (int i = 0; i < mNumRows; i++) {
        mData[i] = new double[mNumCols];
        for (int j = 0; j < mNumCols; j++) {
            mData[i][j] = other.mData[i][j];
        }
    }
}

// Destructor
Matrix::~Matrix() {
    for (int i = 0; i < mNumRows; i++) {
        delete[] mData[i];
    }
    delete[] mData;
```

```
}

// Assignment operator
Matrix& Matrix::operator=(const Matrix& other) {
    if (this != &other) {
        // Delete old data
        for (int i = 0; i < mNumRows; i++) {
            delete[] mData[i];
        }
        delete[] mData;

        // Copy new data
        mNumRows = other.mNumRows;
        mNumCols = other.mNumCols;
        mData = new double*[mNumRows];
        for (int i = 0; i < mNumRows; i++) {
            mData[i] = new double[mNumCols];
            for (int j = 0; j < mNumCols; j++) {
                mData[i][j] = other.mData[i][j];
            }
        }
    }
    return *this;
}

// Access operator
double& Matrix::operator()(int i, int j) {
    assert(i >= 1 && i <= mNumRows && j >= 1 && j <= mNumCols);
    return mData[i-1][j-1];
}

const double& Matrix::operator()(int i, int j) const {
    assert(i >= 1 && i <= mNumRows && j >= 1 && j <= mNumCols);
    return mData[i-1][j-1];
}

// Unary operators
Matrix Matrix::operator+() const {
    return *this;
}

Matrix Matrix::operator-() const {
    Matrix result(mNumRows, mNumCols);
    for (int i = 0; i < mNumRows; i++) {
        for (int j = 0; j < mNumCols; j++) {
```

```
            result.mData[i][j] = -mData[i][j];
        }
    }
    return result;
}

// Binary operators
Matrix Matrix::operator+(const Matrix& other) const {
    assert(mNumRows == other.mNumRows && mNumCols == other.mNumCols);
    Matrix result(mNumRows, mNumCols);
    for (int i = 0; i < mNumRows; i++) {
        for (int j = 0; j < mNumCols; j++) {
            result.mData[i][j] = mData[i][j] + other.mData[i][j];
        }
    }
    return result;
}

Matrix Matrix::operator-(const Matrix& other) const {
    assert(mNumRows == other.mNumRows && mNumCols == other.mNumCols);
    Matrix result(mNumRows, mNumCols);
    for (int i = 0; i < mNumRows; i++) {
        for (int j = 0; j < mNumCols; j++) {
            result.mData[i][j] = mData[i][j] - other.mData[i][j];
        }
    }
    return result;
}

Matrix Matrix::operator*(const Matrix& other) const {
    assert(mNumCols == other.mNumRows);
    Matrix result(mNumRows, other.mNumCols);
    for (int i = 0; i < mNumRows; i++) {
        for (int j = 0; j < other.mNumCols; j++) {
            double sum = 0.0;
            for (int k = 0; k < mNumCols; k++) {
                sum += mData[i][k] * other.mData[k][j];
            }
            result.mData[i][j] = sum;
        }
    }
    return result;
}

Matrix Matrix::operator*(double scalar) const {
```

```
    Matrix result(mNumRows, mNumCols);
    for (int i = 0; i < mNumRows; i++) {
        for (int j = 0; j < mNumCols; j++) {
            result.mData[i][j] = mData[i][j] * scalar;
        }
    }
    return result;
}

Vector Matrix::operator*(const Vector& vec) const {
    assert(mNumCols == vec.GetSize());
    Vector result(mNumRows);
    for (int i = 0; i < mNumRows; i++) {
        double sum = 0.0;
        for (int j = 0; j < mNumCols; j++) {
            sum += mData[i][j] * vec[j];
        }
        result[i] = sum;
    }
    return result;
}

Matrix operator*(double scalar, const Matrix& mat) {
    return mat * scalar;
}

// Compound assignment operators
Matrix& Matrix::operator+=(const Matrix& other) {
    assert(mNumRows == other.mNumRows && mNumCols == other.mNumCols);
    for (int i = 0; i < mNumRows; i++) {
        for (int j = 0; j < mNumCols; j++) {
            mData[i][j] += other.mData[i][j];
        }
    }
    return *this;
}

Matrix& Matrix::operator-=(const Matrix& other) {
    assert(mNumRows == other.mNumRows && mNumCols == other.mNumCols);
    for (int i = 0; i < mNumRows; i++) {
        for (int j = 0; j < mNumCols; j++) {
            mData[i][j] -= other.mData[i][j];
        }
    }
    return *this;
```

```
}

Matrix& Matrix::operator*=(double scalar) {
    for (int i = 0; i < mNumRows; i++) {
        for (int j = 0; j < mNumCols; j++) {
            mData[i][j] *= scalar;
        }
    }
    return *this;
}

// Utility functions
bool Matrix::IsSymmetric() const {
    if (!IsSquare()) return false;
    for (int i = 0; i < mNumRows; i++) {
        for (int j = i + 1; j < mNumCols; j++) {
            if (mData[i][j] != mData[j][i]) return false;
        }
    }
    return true;
}

Matrix Matrix::Transpose() const {
    Matrix result(mNumCols, mNumRows);
    for (int i = 0; i < mNumRows; i++) {
        for (int j = 0; j < mNumCols; j++) {
            result.mData[j][i] = mData[i][j];
        }
    }
    return result;
}

double Matrix::Determinant() const {
    assert(IsSquare());
    // For simplicity, we'll implement a basic determinant calculation
    // This is not the most efficient method for large matrices
    if (mNumRows == 1) return mData[0][0];
    if (mNumRows == 2) {
        return mData[0][0] * mData[1][1] - mData[0][1] * mData[1][0];
    }

    double det = 0.0;
    for (int j = 0; j < mNumCols; j++) {
        // Create submatrix
        Matrix submatrix(mNumRows - 1, mNumCols - 1);
```

```cpp
        for (int i = 1; i < mNumRows; i++) {
            for (int k = 0; k < mNumCols; k++) {
                if (k < j) submatrix.mData[i-1][k] = mData[i][k];
                else if (k > j) submatrix.mData[i-1][k-1] = mData[i][k];
            }
        }
        det += (j % 2 == 0 ? 1 : -1) * mData[0][j] * submatrix.Determinant();
    }
    return det;
}

Matrix Matrix::Inverse() const {
    assert(IsSquare());
    double det = Determinant();
    assert(abs(det) > 1e-10); // Check if matrix is singular

    Matrix result(mNumRows, mNumCols);
    // For simplicity, we'll implement a basic inverse calculation
    // This is not the most efficient method for large matrices
    if (mNumRows == 1) {
        result.mData[0][0] = 1.0 / mData[0][0];
        return result;
    }
    if (mNumRows == 2) {
        result.mData[0][0] = mData[1][1] / det;
        result.mData[0][1] = -mData[0][1] / det;
        result.mData[1][0] = -mData[1][0] / det;
        result.mData[1][1] = mData[0][0] / det;
        return result;
    }

    // For larger matrices, we'll use the adjugate method
    // This is not the most efficient method, but it's straightforward
    for (int i = 0; i < mNumRows; i++) {
        for (int j = 0; j < mNumCols; j++) {
            // Create submatrix
            Matrix submatrix(mNumRows - 1, mNumCols - 1);
            for (int k = 0; k < mNumRows; k++) {
                for (int l = 0; l < mNumCols; l++) {
                    if (k < i && l < j) submatrix.mData[k][l] = mData[k][l];
                    else if (k < i && l > j) submatrix.mData[k][l-1] = mData[k][l];
```

```
                    else if (k > i && l < j) submatrix.mData[k-1][l] = mData[k][l];
                    else if (k > i && l > j) submatrix.mData[k-1][l-1] = mData[k][l];
                }
            }
            result.mData[j][i] = ((i + j) % 2 == 0 ? 1 : -1) * submatrix.Determinant() / det;
        }
    }
    return result;
}

Matrix Matrix::PseudoInverse() const {
    // Moore-Penrose pseudo-inverse using SVD
    // For simplicity, we'll use a basic implementation
    // In practice, you would want to use a more robust method
    Matrix A = *this;
    Matrix AT = A.Transpose();
    Matrix ATA = AT * A;
    Matrix AAT = A * AT;

    // Check if ATA is invertible
    if (abs(ATA.Determinant()) > 1e-10) {
        return AT * (A * AT).Inverse();
    }
    // Check if AAT is invertible
    else if (abs(AAT.Determinant()) > 1e-10) {
        return (AT * A).Inverse() * AT;
    }
    else {
        // If neither is invertible, use the formula: A+ = (A^T * A)^(-1) * A^T
        // with regularization
        double lambda = 1e-10; // Small regularization parameter
        Matrix I(mNumCols, mNumCols);
        for (int i = 0; i < mNumCols; i++) {
            I.mData[i][i] = 1.0;
        }
        return (AT * A + lambda * I).Inverse() * AT;
    }
}

// I/O operators
ostream& operator<<(ostream& os, const Matrix& mat) {
```

```
    for (int i = 0; i < mat.mNumRows; i++) {
        os << "[";
        for (int j = 0; j < mat.mNumCols; j++) {
            os << mat.mData[i][j];
            if (j < mat.mNumCols - 1) os << ", ";
        }
        os << "]" << (i < mat.mNumRows - 1 ? "\n" : "");
    }
    return os;
}

istream& operator>>(istream& is, Matrix& mat) {
    for (int i = 0; i < mat.mNumRows; i++) {
        for (int j = 0; j < mat.mNumCols; j++) {
            is >> mat.mData[i][j];
        }
    }
    return is;
}

#endif // MATRIX_H
```

2.3.3 Functionality

- **Memory Management**:
  - `Matrix(int numRows, int numCols)`: Allocates a `numRows` × `numCols` matrix, initializes to zeros.
  - `Matrix(const Matrix&)`: Deep copies matrix data.
  - `Matrix()`: Frees dynamically allocated memory.
  - `operator=`: Implements deep copy with self-assignment check.
- **Indexing**:
  - `operator()`: Provides 1-based access to `mData[i-1][j-1]`.
- **Arithmetic Operators**:
  - `operator+`: Adds matrices element-wise.
  - `operator-`: Subtracts matrices.
  - `operator*`: Performs matrix-matrix or matrix-vector multiplication.
  - `operator* (scalar)`: Scales matrix by a scalar (left/right).
  - `operator+=`, `-=`, `*=`: In-place operations.

- **Utility Functions**:
  - `IsSymmetric()`: Checks if $A = A^T$ for square matrices.
  - `Transpose()`: Returns A^T.
  - `Determinant()`: Computes determinant recursively for square matrices.
  - `Inverse()`: Computes inverse using adjugate method for square matrices.
  - `PseudoInverse()`: Computes Moore-Penrose pseudo-inverse using $A^+ = (A^T A)^{-1} A^T$ or $A^+ = A^T (AA^T)^{-1}$, with regularization for ill-conditioned cases.
- **I/O**:
  - `operator«`: Outputs matrix row-wise as `[x1, x2, ...]`.
  - `operator»`: Reads matrix elements.

2.3.4 Testing

The `main` function in `Matrix.cpp` tests:

- Input of two 3×3 matrices A, B.
- Operations: $A + B$, $A * B$, $A * 2$, $A+ = B$, $A- = B$, $A* = 2$.
- Utility functions: `Determinant()`, `Inverse()`.

Example input: $A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$, $B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$:

$$A + B = \begin{bmatrix} 2 & 2 & 3 \\ 4 & 6 & 6 \\ 7 & 8 & 10 \end{bmatrix}, \quad A * B = A$$

2.4 LinearSystem Class Implementation

2.4.1 Purpose

The `LinearSystem` class solves square linear systems $Ax = b$ using Gaussian elimination with partial pivoting. The derived `PosSymLinSystem` class specializes in solving symmetric positive definite systems using the conjugate gradient method.

2.4.2 Code Excerpt

Key excerpts from `LinearSystem.h`:

Listing 3: LinearSystem.h

```
#ifndef LINEAR_SYSTEM_H
```

```
#define LINEAR_SYSTEM_H

#include "Matrix.h"

class LinearSystem {
protected:
    int mSize;           // Size of the linear system
    Matrix* mpA;         // Pointer to the coefficient matrix
    Vector* mpb;         // Pointer to the right-hand side vector

public:
    // Constructor
    LinearSystem(const Matrix& A, const Vector& b);

    // Prevent default construction and copying
    LinearSystem() = delete;
    LinearSystem(const LinearSystem& other) = delete;
    LinearSystem& operator=(const LinearSystem& other) = delete;

    // Destructor
    virtual ~LinearSystem();

    // Virtual solve method to be overridden by derived classes
    virtual Vector Solve() const;

    // Utility functions
    int GetSize() const { return mSize; }
    const Matrix& GetMatrix() const { return *mpA; }
    const Vector& GetVector() const { return *mpb; }
};

// Derived class for positive definite symmetric linear systems
class PosSymLinSystem : public LinearSystem {
public:
    // Constructor
    PosSymLinSystem(const Matrix& A, const Vector& b);

    // Override solve method to use conjugate gradient method
    Vector Solve() const override;

private:
    // Helper functions for conjugate gradient method
    double ComputeAlpha(const Vector& r, const Vector& p, const Matrix
        & A) const;
    double ComputeBeta(const Vector& r, const Vector& rNext) const;
```

```
    bool CheckConvergence(const Vector& r, double tolerance = 1e-10) const;
};

// LinearSystem constructor
LinearSystem::LinearSystem(const Matrix& A, const Vector& b) {
    // Check if the system is valid
    if (!A.IsSquare()) {
        throw std::invalid_argument("Matrix A must be square");
    }
    if (A.GetNumRows() != b.GetSize()) {
        throw std::invalid_argument("Matrix A and vector b must have compatible dimensions");
    }

    mSize = A.GetNumRows();
    mpA = new Matrix(A);
    mpb = new Vector(b);
}

// LinearSystem destructor
LinearSystem::~LinearSystem() {
    delete mpA;
    delete mpb;
}

// LinearSystem Solve method using Gaussian elimination with partial pivoting
Vector LinearSystem::Solve() const {
    // Create copies of the matrix and vector to work with
    Matrix A = *mpA;
    Vector b = *mpb;
    Vector x(mSize);

    // Gaussian elimination with partial pivoting
    for (int k = 1; k <= mSize; k++) {
        // Find pivot
        int maxRow = k;
        double maxVal = std::abs(A(k, k));
        for (int i = k + 1; i <= mSize; i++) {
            double val = std::abs(A(i, k));
            if (val > maxVal) {
                maxVal = val;
                maxRow = i;
            }
```

```
        }

        // Check if matrix is singular
        if (maxVal < 1e-10) {
            throw std::runtime_error("Matrix is singular or nearly singular");
        }

        // Swap rows if necessary
        if (maxRow != k) {
            // Swap rows in matrix A
            for (int j = 1; j <= mSize; j++) {
                std::swap(A(k, j), A(maxRow, j));
            }
            // Swap elements in vector b (using 1-based indexing)
            std::swap(b(k), b(maxRow));
        }

        // Eliminate column k
        for (int i = k + 1; i <= mSize; i++) {
            double factor = A(i, k) / A(k, k);
            A(i, k) = 0.0;  // Explicitly set to zero for numerical stability
            for (int j = k + 1; j <= mSize; j++) {
                A(i, j) -= factor * A(k, j);
            }
            b(i) -= factor * b(k);
        }
    }

    // Back substitution
    for (int i = mSize; i >= 1; i--) {
        double sum = 0.0;
        for (int j = i + 1; j <= mSize; j++) {
            sum += A(i, j) * x(j);
        }
        x(i) = (b(i) - sum) / A(i, i);
    }

    return x;
}

// PosSymLinSystem constructor
PosSymLinSystem::PosSymLinSystem(const Matrix& A, const Vector& b) : LinearSystem(A, b) {
```

```
    // Check if the matrix is symmetric
    if (!A.IsSymmetric()) {
        throw invalid_argument("Matrix must be symmetric for ↩
            PosSymLinSystem");
    }

    // Additional check for positive definiteness (optional)
    // This is a simple check that might not catch all cases
    for (int i = 1; i <= A.GetNumRows(); i++) {
        if (A(i, i) <= 0) {
            throw invalid_argument("Matrix must be positive definite")↩
                ;
        }
    }
}

// PosSymLinSystem Solve method using conjugate gradient method
Vector PosSymLinSystem::Solve() const {
    const int maxIterations = mSize;  // Maximum number of iterations
    const double tolerance = 1e-10;   // Convergence tolerance
    const double minResidual = 1e-15; // Minimum residual to prevent ↩
        division by zero

    Vector x(mSize);                   // Initial guess (zero vector)
    Vector r = *mpb - (*mpA) * x;     // Initial residual
    Vector p = r;                      // Initial search direction
    Vector rNext(mSize);              // Next residual
    Vector Ap(mSize);                 // A * p

    double initialResidual = r.Norm();
    if (initialResidual < minResidual) {
        return x;  // Initial guess is already solution
    }

    for (int iter = 0; iter < maxIterations; iter++) {
        Ap = (*mpA) * p;
        double alpha = ComputeAlpha(r, p, *mpA);

        // Update solution and residual
        x += alpha * p;
        rNext = r - alpha * Ap;

        // Check convergence
        if (CheckConvergence(rNext, tolerance * initialResidual)) {
            return x;
```

```cpp
        }

        // Update search direction
        double beta = ComputeBeta(r, rNext);
        if (std::abs(beta) < minResidual) {
            // If beta is too small, restart the algorithm
            r = *mpb - (*mpA) * x;
            p = r;
        } else {
            p = rNext + beta * p;
            r = rNext;
        }
    }

    throw std::runtime_error("Conjugate gradient method did not converge within " +
                             std::to_string(maxIterations) + " iterations");
}

// Helper function to compute alpha in conjugate gradient method
double PosSymLinSystem::ComputeAlpha(const Vector& r, const Vector& p, const Matrix& A) const {
    Vector Ap = A * p;
    double pAp = p.DotProduct(Ap);
    if (std::abs(pAp) < 1e-15) {
        throw std::runtime_error("Matrix is not positive definite");
    }
    return r.DotProduct(r) / pAp;
}

// Helper function to compute beta in conjugate gradient method
double PosSymLinSystem::ComputeBeta(const Vector& r, const Vector& rNext) const {
    double rDotR = r.DotProduct(r);
    if (std::abs(rDotR) < 1e-15) {
        return 0.0;  // Prevent division by zero
    }
    return rNext.DotProduct(rNext) / rDotR;
}

// Helper function to check convergence in conjugate gradient method
bool PosSymLinSystem::CheckConvergence(const Vector& r, double tolerance) const {
    return r.Norm() <= tolerance;
```

```
}

#endif // LINEAR_SYSTEM_H
```

2.4.3 Functionality

- **LinearSystem**:
  - `LinearSystem(const Matrix& A, const Vector& b)`: Initializes with a square matrix A and vector b, checking compatibility.
  - `Solve()`: Uses Gaussian elimination with partial pivoting to solve $Ax = b$. Steps:
    * Pivot selection to maximize numerical stability.
    * Row swapping if needed.
    * Forward elimination to form an upper triangular matrix.
    * Back substitution to compute x.
  - Prevents default construction and copying to ensure data integrity.

- **PosSymLinSystem**:
  - `PosSymLinSystem(const Matrix& A, const Vector& b)`: Verifies A is symmetric and has positive diagonal elements.
  - `Solve()`: Implements the conjugate gradient method, an iterative solver for symmetric positive definite systems. Algorithm:

    $$x_0 = 0, \quad r_0 = b - Ax_0, \quad p_0 = r_0$$

    For each iteration:

    $$\alpha_k = \frac{r_k^T r_k}{p_k^T A p_k}, \quad x_{k+1} = x_k + \alpha_k p_k, \quad r_{k+1} = r_k - \alpha_k A p_k$$

    $$\beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}, \quad p_{k+1} = r_{k+1} + \beta_k p_k$$

    Converges when $\|r_k\| \leq$ tolerance.

2.4.4 Testing

The `main` function in `LinearSystem.cpp` tests:

- A 3×3 system:

  $$\begin{bmatrix} 2 & 1 & -1 \\ -3 & -1 & 2 \\ -2 & 1 & 2 \end{bmatrix} x = \begin{bmatrix} 8 \\ -11 \\ -3 \end{bmatrix}$$

  Solved using Gaussian elimination, expecting $x = [2, 3, -1]$.

- A symmetric positive definite system:

$$\begin{bmatrix} 4 & 1 & 1 \\ 1 & 5 & 2 \\ 1 & 2 & 6 \end{bmatrix} y = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

  Solved using conjugate gradient, with residuals verified via $\|Ax - b\|$.

2.5 Handling Under-Determined and Over-Determined Systems

2.5.1 Approach

For non-square systems $Ax = b$:

- **Under-Determined Systems** ($m < n$): Multiple solutions exist. The minimum-norm solution is:

$$x = A^+ b, \quad \text{where } A^+ = A^T (AA^T)^{-1}$$

- **Over-Determined Systems** ($m > n$): No exact solution typically exists. The least-squares solution is:

$$x = A^+ b, \quad \text{where } A^+ = (A^T A)^{-1} A^T$$

  The `PseudoInverse()` method in `Matrix` computes A^+, checking invertibility of $A^T A$ or AA^T.

2.5.2 Tikhonov Regularization

To handle ill-conditioned systems, a regularization term is added:

$$x = (A^T A + \lambda I)^{-1} A^T b$$

The `PseudoInverse()` method uses a small $\lambda = 10^{-10}$ when neither $A^T A$ nor AA^T is invertible, ensuring numerical stability.

2.5.3 Implementation

The `PseudoInverse()` method:

- Computes A^T.
- Attempts $A^+ = A^T (AA^T)^{-1}$ if AA^T is invertible.
- Falls back to $A^+ = (A^T A)^{-1} A^T$ if $A^T A$ is invertible.
- Uses regularization $(A^T A + \lambda I)^{-1} A^T$ otherwise.

2.6 Design and Features

- **Memory Management**: All classes use dynamic allocation with deep copying, ensuring no memory leaks via destructors.

- **Error Handling**: `assert` and exceptions (`std::invalid`$_{a}rgument$,

2.7 Limitations and Improvements

- **Performance**: Recursive determinant and inverse methods are inefficient for large matrices. LU decomposition or QR factorization could improve speed.
- **Error Handling**: Replacing `assert` with exceptions would allow recovery in production code.
- **Move Semantics**: Adding move constructors would optimize temporary object handling.
- **Positive Definiteness Check**: `PosSymLinSystem` only checks positive diagonal elements, which is insufficient. Cholesky decomposition could verify positive definiteness.

2.8 Conclusion

The `Vector`, `Matrix`, and `LinearSystem` classes provide a robust linear algebra library for Part A of the Tiny Project. They support vector and matrix operations, solve square systems efficiently, and handle non-square systems via pseudo-inverse with regularization. The implementation meets all requirements, with comprehensive testing demonstrating correctness. Future enhancements could focus on performance optimization and advanced numerical methods.

3 Part B

3.1 Overview

Part B of the Tiny Project implements linear regression to predict relative CPU performance (PRP) using the UCI Computer Hardware dataset, which contains 209 instances with six predictive features: MYCT, MMIN, MMAX, CACH, CHMIN, and CHMAX. The implementation in `Source.cpp` uses matrix operations to compute regression coefficients (β) and evaluates the model using root mean square error (RMSE) on an 80/20 train-test split. The model assumes:

$$\text{PRP} = \beta_1 \cdot \text{MYCT} + \beta_2 \cdot \text{MMIN} + \beta_3 \cdot \text{MMAX} + \beta_4 \cdot \text{CACH} + \beta_5 \cdot \text{CHMIN} + \beta_6 \cdot \text{CHMAX}$$

3.2 Key Operations

- **Data Loading (`loadData`)**:
  - Reads the UCI dataset (`machine.data`), skipping vendor and model fields.
  - Stores 209 instances with features (MYCT, MMIN, MMAX, CACH, CHMIN, CHMAX) and target (PRP) in a `DataRow` struct.
- **Data Preparation**:
  - Shuffles data and splits into 80% training (167 instances) and 20% testing (42 instances).

  - Constructs training matrix X (167×6) and vector Y (167×1) from features and PRP.

- **Matrix Operations**:
  - `multiply`: Computes matrix product $A \times B$.
  - `transpose`: Returns A^T.
  - `inverse`: Uses Gauss-Jordan elimination to compute the inverse of a square matrix.

- **Linear Regression (`linearRegression`)**:
  - Computes coefficients using $\beta = (X^T X)^{-1} X^T Y$.
  - Handles the over-determined system (167 equations, 6 unknowns) via the normal equation.

- **Evaluation (`rmse`)**:
  - Calculates RMSE on the test set: $\sqrt{\frac{1}{n}\sum(y_i - \hat{y}_i)^2}$, where $\hat{y}_i = X_i \cdot \beta$.

3.3 Implementation Details

The program:

- Loads the dataset and splits it randomly (80/20).
- Constructs X_{train} and Y_{train}, computes β using the normal equation.
- Outputs β coefficients and RMSE on $X_{\text{test}}, Y_{\text{test}}$.

The implementation uses STL vectors for matrices and avoids external libraries, ensuring simplicity. However, it assumes $X^T X$ is invertible and lacks regularization, which could affect stability for ill-conditioned data.

3.4 Model Performance

The linear regression model, implemented in `Source.cpp`, was trained on an 80% subset (167 instances) of the UCI Computer Hardware dataset and evaluated on the remaining 20% (42 instances). The model uses six features—MYCT (machine cycle time), MMIN (minimum main memory), MMAX (maximum main memory), CACH (cache memory), CHMIN (minimum channels), and CHMAX (maximum channels)—to predict the relative CPU performance (PRP). The training process yielded the following beta coefficients and root mean square error (RMSE) on the test set:

- **Beta Coefficients**: $[-0.0453475, 0.0152298, 0.00455789, 0.521299, -0.986111, 1.3428]$

  - The coefficients indicate the contribution of each feature to PRP. Notably, CHMAX (1.3428) and CACH (0.521299) have the largest positive impacts, suggesting that maximum channels and cache memory significantly enhance performance. Conversely, CHMIN (-0.986111) has a strong negative effect, possibly indicating diminishing returns or inefficiencies at higher channel counts. MYCT (-0.0453475) and MMIN (0.0152298) show minor influences, while MMAX (0.00455789) has a negligible effect.

- **RMSE on Test Set**: 59.999

  - The RMSE of 59.999 indicates the average prediction error on the test set. Given that PRP values in the dataset range from approximately 10 to 1500 (based on `machine.data`), an RMSE of 59.999 suggests moderate accuracy, accounting for about 4-6% of the typical PRP range. However, this value is unusually high and close to a round number, which may suggest potential issues such as overfitting, inadequate feature scaling, or numerical instability in the matrix inversion process (e.g., in the Gauss-Jordan method).

```
Beta coefficients:
-0.0453475 0.0152298 0.00455789 0.521299 -0.986111 1.3428
RMSE on test set: 59.999
```

Hình 1: Beta coefficients and RMSE on test set

3.5 Analysis

The model's performance can be attributed to several factors:

- The high RMSE suggests that the linear model may not fully capture the non-linear relationships or interactions between features in the dataset. Feature engineering (e.g., polynomial terms) or a non-linear model (e.g., polynomial regression) could improve accuracy.

- The random 80/20 split and lack of cross-validation may have resulted in an unlucky partition, leading to a poor test set performance. Multiple splits or k-fold cross-validation could provide a more robust RMSE estimate.

- The use of raw feature values without normalization or standardization might have skewed the regression, as features like MYCT and MMAX vary widely in scale. Normalizing features (e.g., to zero mean and unit variance) before training could stabilize the coefficients and reduce RMSE.

3.6 Implications and Next Steps

The beta coefficients provide initial insights into feature importance, with CHMAX and CACH being key predictors of PRP. However, the high RMSE indicates that the current model requires refinement. Suggested improvements include:

- Preprocessing data with normalization or standardization to enhance numerical stability.
- Implementing regularization (e.g., Ridge or Lasso) to mitigate overfitting and improve generalization.
- Re-running the model with multiple train-test splits to confirm the RMSE reliability.
- Exploring advanced models (e.g., random forests or neural networks) if linear assumptions are invalid.

Further analysis of residuals and feature correlations could also guide these enhancements.

4 Conclusion

This project helped us practice two different but important skills:

- In Part A, we built a small C++ library to work with vectors and matrices. We used this library to solve systems of equations using methods like Gaussian elimination and the conjugate gradient method. We also learned how to handle special cases like when the system has more equations than unknowns (over-determined) or the opposite (under-determined). This part helped us improve our understanding of C++ programming and basic linear algebra.
- In Part B, we used real-world data from UCI to build a simple model that predicts CPU performance based on hardware features. We used linear regression and tested how well the model worked using RMSE. Even though the error was not very low, the model still captured the main trend in the data.

In short, this project allowed us to apply both programming and math skills. It showed how basic algorithms can be used in real applications and gave us experience in working with both theoretical and practical problems.