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 **2**

2 Part A **2**

2.1 Overview . . . 2
2.2 Vector Class Implementation . . . 2
2.2.1 **Purpose** . . . 2
2.2.2 **Vector Code** . . . 2
2.2.3 **Functionality** . . . 7
2.2.4 **Testing** . . . 8
2.3 Matrix Class Implementation . . . 8
2.3.1 **Purpose** . . . 8
2.3.2 **Code Excerpt** . . . 8
2.3.3 **Functionality** . . . 17
2.3.4 **Testing** . . . 18
2.4 LinearSystem Class Implementation . . . 18
2.4.1 **Purpose** . . . 18
2.4.2 **Code Excerpt** . . . 18
2.4.3 **Functionality** . . . 24
2.4.4 **Testing** . . . 24
2.5 Handling Under-Determined and Over-Determined Systems . . . 25
2.5.1 **Approach** . . . 25
2.5.2 **Tikhonov Regularization** . . . 25
2.5.3 **Implementation** . . . 25
2.6 Design and Features . . . 25
2.7 Limitations and Improvements . . . 26
2.8 Conclusion . . . 26

3 Part B **26**

3.1 Overview . . . 26
3.2 Key Operations . . . 26
3.3 Implementation Details . . . 27
3.4 Model Performance . . . 27
3.5 Analysis . . . 28
3.6 Implications and Next Steps . . . 29

4 Conclusion **29**

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

```
1 class Vector {
2 private:
```

```
3       int mSize;      // Size of the vector
4       double* mData;  // Pointer to the data array
5
6   public:
7       // Constructors and destructor
8       Vector(int size);                    // Constructor with size
9       Vector(const Vector& other);         // Copy constructor
10      ~Vector();                           // Destructor
11
12      // Assignment operator
13      Vector& operator=(const Vector& other);
14
15      // Access operators
16      double& operator[](int i);           // 0-based indexing with ↩
            bounds checking
17      const double& operator[](int i) const;
18      double& operator()(int i);           // 1-based indexing
19      const double& operator()(int i) const;
20
21      // Unary operators
22      Vector operator+() const;            // Unary plus
23      Vector operator-() const;            // Unary minus
24
25      // Binary operators
26      Vector operator+(const Vector& other) const;  // Vector addition
27      Vector operator-(const Vector& other) const;  // Vector ↩
            subtraction
28      Vector operator*(double scalar) const;        // Scalar ↩
            multiplication
29      friend Vector operator*(double scalar, const Vector& vec);  // ↩
            Scalar multiplication (left)
30
31      // Compound assignment operators
32      Vector& operator+=(const Vector& other);
33      Vector& operator-=(const Vector& other);
34      Vector& operator*=(double scalar);
35
36      // Utility functions
37      int GetSize() const { return mSize; }
38      double Norm() const;                 // Euclidean norm
39      double DotProduct(const Vector& other) const;  // Dot product
40
41      // Friend functions for I/O
42      friend ostream& operator<<(ostream& os, const Vector& vec);
43      friend istream& operator>>(istream& is, Vector& vec);
```

```
};

// 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);
```

```
89      return mData[i];
90  }
91
92  double& Vector::operator()(int i) {
93      assert(i >= 1 && i <= mSize);
94      return mData[i-1];
95  }
96
97  const double& Vector::operator()(int i) const {
98      assert(i >= 1 && i <= mSize);
99      return mData[i-1];
100 }
101
102 // Unary operators
103 Vector Vector::operator+() const {
104     return *this;
105 }
106
107 Vector Vector::operator-() const {
108     Vector result(mSize);
109     for (int i = 0; i < mSize; i++) {
110         result.mData[i] = -mData[i];
111     }
112     return result;
113 }
114
115 // Binary operators
116 Vector Vector::operator+(const Vector& other) const {
117     assert(mSize == other.mSize);
118     Vector result(mSize);
119     for (int i = 0; i < mSize; i++) {
120         result.mData[i] = mData[i] + other.mData[i];
121     }
122     return result;
123 }
124
125 Vector Vector::operator-(const Vector& other) const {
126     assert(mSize == other.mSize);
127     Vector result(mSize);
128     for (int i = 0; i < mSize; i++) {
129         result.mData[i] = mData[i] - other.mData[i];
130     }
131     return result;
132 }
133
```

```
134 Vector Vector::operator*(double scalar) const {
135     Vector result(mSize);
136     for (int i = 0; i < mSize; i++) {
137         result.mData[i] = mData[i] * scalar;
138     }
139     return result;
140 }
141 
142 Vector operator*(double scalar, const Vector& vec) {
143     return vec * scalar;
144 }
145 
146 // Compound assignment operators
147 Vector& Vector::operator+=(const Vector& other) {
148     assert(mSize == other.mSize);
149     for (int i = 0; i < mSize; i++) {
150         mData[i] += other.mData[i];
151     }
152     return *this;
153 }
154 
155 Vector& Vector::operator-=(const Vector& other) {
156     assert(mSize == other.mSize);
157     for (int i = 0; i < mSize; i++) {
158         mData[i] -= other.mData[i];
159     }
160     return *this;
161 }
162 
163 Vector& Vector::operator*=(double scalar) {
164     for (int i = 0; i < mSize; i++) {
165         mData[i] *= scalar;
166     }
167     return *this;
168 }
169 
170 // Utility functions
171 double Vector::Norm() const {
172     double sum = 0.0;
173     for (int i = 0; i < mSize; i++) {
174         sum += mData[i] * mData[i];
175     }
176     return sqrt(sum);
177 }
178 
```

```
179 double Vector::DotProduct(const Vector& other) const {
180     assert(mSize == other.mSize);
181     double sum = 0.0;
182     for (int i = 0; i < mSize; i++) {
183         sum += mData[i] * other.mData[i];
184     }
185     return sum;
186 }
187
188 // I/O operators
189 ostream& operator<<(ostream& os, const Vector& vec) {
190     os << "[";
191     for (int i = 0; i < vec.mSize; i++) {
192         os << vec.mData[i];
193         if (i < vec.mSize - 1) os << ", ";
194     }
195     os << "]";
196     return os;
197 }
198
199 istream& operator>>(istream& is, Vector& vec) {
200     for (int i = 0; i < vec.mSize; i++) {
201         is >> vec.mData[i];
202     }
203     return is;
204 }
205
206 #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

```
1 #ifndef MATRIX_H
2 #define MATRIX_H
3
4 #include "Vector.h"
5
```

```
6  class Matrix {
7  private:
8      int mNumRows;      // Number of rows
9      int mNumCols;      // Number of columns
10     double** mData;    // Pointer to array of pointers (2D array)
11
12 public:
13     // Constructors and destructor
14     Matrix(int numRows, int numCols);      // Constructor with ↩
           dimensions
15     Matrix(const Matrix& other);           // Copy constructor
16     ~Matrix();                            // Destructor
17
18     // Assignment operator
19     Matrix& operator=(const Matrix& other);
20
21     // Access operators
22     double& operator()(int i, int j);      // 1-based indexing
23     const double& operator()(int i, int j) const;
24
25     // Unary operators
26     Matrix operator+() const;              // Unary plus
27     Matrix operator-() const;              // Unary minus
28
29     // Binary operators
30     Matrix operator+(const Matrix& other) const;  // Matrix addition
31     Matrix operator-(const Matrix& other) const;  // Matrix ↩
           subtraction
32     Matrix operator*(const Matrix& other) const;  // Matrix ↩
           multiplication
33     Matrix operator*(double scalar) const;        // Scalar ↩
           multiplication
34     Vector operator*(const Vector& vec) const;    // Matrix-vector ↩
           multiplication
35     friend Matrix operator*(double scalar, const Matrix& mat);  // ↩
           Scalar multiplication (left)
36
37     // Compound assignment operators
38     Matrix& operator+=(const Matrix& other);
39     Matrix& operator-=(const Matrix& other);
40     Matrix& operator*=(double scalar);
41
42     // Utility functions
43     int GetNumRows() const { return mNumRows; }
44     int GetNumCols() const { return mNumCols; }
```

```
45      bool IsSquare() const { return mNumRows == mNumCols; }
46      bool IsSymmetric() const;              // Check if matrix is ↩
            symmetric
47      double Determinant() const;            // Calculate determinant
48      Matrix Inverse() const;                // Calculate inverse
49      Matrix PseudoInverse() const;          // Calculate Moore-Penrose ↩
            pseudo-inverse
50      Matrix Transpose() const;              // Calculate transpose
51
52      // Friend functions for I/O
53      friend ostream& operator<<(ostream& os, const Matrix& mat);
54      friend istream& operator>>(istream& is, Matrix& mat);
55  };
56
57  // Constructor
58  Matrix::Matrix(int numRows, int numCols) : mNumRows(numRows), mNumCols↩
        (numCols) {
59      assert(numRows > 0 && numCols > 0);
60      mData = new double*[numRows];
61      for (int i = 0; i < numRows; i++) {
62          mData[i] = new double[numCols];
63          for (int j = 0; j < numCols; j++) {
64              mData[i][j] = 0.0;
65          }
66      }
67  }
68
69  // Copy constructor
70  Matrix::Matrix(const Matrix& other) : mNumRows(other.mNumRows), ↩
        mNumCols(other.mNumCols) {
71      mData = new double*[mNumRows];
72      for (int i = 0; i < mNumRows; i++) {
73          mData[i] = new double[mNumCols];
74          for (int j = 0; j < mNumCols; j++) {
75              mData[i][j] = other.mData[i][j];
76          }
77      }
78  }
79
80  // Destructor
81  Matrix::~Matrix() {
82      for (int i = 0; i < mNumRows; i++) {
83          delete[] mData[i];
84      }
85      delete[] mData;
```

```
86  }
87
88  // Assignment operator
89  Matrix& Matrix::operator=(const Matrix& other) {
90      if (this != &other) {
91          // Delete old data
92          for (int i = 0; i < mNumRows; i++) {
93              delete[] mData[i];
94          }
95          delete[] mData;
96
97          // Copy new data
98          mNumRows = other.mNumRows;
99          mNumCols = other.mNumCols;
100         mData = new double*[mNumRows];
101         for (int i = 0; i < mNumRows; i++) {
102             mData[i] = new double[mNumCols];
103             for (int j = 0; j < mNumCols; j++) {
104                 mData[i][j] = other.mData[i][j];
105             }
106         }
107     }
108     return *this;
109 }
110
111 // Access operator
112 double& Matrix::operator()(int i, int j) {
113     assert(i >= 1 && i <= mNumRows && j >= 1 && j <= mNumCols);
114     return mData[i-1][j-1];
115 }
116
117 const double& Matrix::operator()(int i, int j) const {
118     assert(i >= 1 && i <= mNumRows && j >= 1 && j <= mNumCols);
119     return mData[i-1][j-1];
120 }
121
122 // Unary operators
123 Matrix Matrix::operator+() const {
124     return *this;
125 }
126
127 Matrix Matrix::operator-() const {
128     Matrix result(mNumRows, mNumCols);
129     for (int i = 0; i < mNumRows; i++) {
130         for (int j = 0; j < mNumCols; j++) {
```

```
131                 result.mData[i][j] = -mData[i][j];
132             }
133         }
134         return result;
135     }
136 
137     // Binary operators
138     Matrix Matrix::operator+(const Matrix& other) const {
139         assert(mNumRows == other.mNumRows && mNumCols == other.mNumCols);
140         Matrix result(mNumRows, mNumCols);
141         for (int i = 0; i < mNumRows; i++) {
142             for (int j = 0; j < mNumCols; j++) {
143                 result.mData[i][j] = mData[i][j] + other.mData[i][j];
144             }
145         }
146         return result;
147     }
148 
149     Matrix Matrix::operator-(const Matrix& other) const {
150         assert(mNumRows == other.mNumRows && mNumCols == other.mNumCols);
151         Matrix result(mNumRows, mNumCols);
152         for (int i = 0; i < mNumRows; i++) {
153             for (int j = 0; j < mNumCols; j++) {
154                 result.mData[i][j] = mData[i][j] - other.mData[i][j];
155             }
156         }
157         return result;
158     }
159 
160     Matrix Matrix::operator*(const Matrix& other) const {
161         assert(mNumCols == other.mNumRows);
162         Matrix result(mNumRows, other.mNumCols);
163         for (int i = 0; i < mNumRows; i++) {
164             for (int j = 0; j < other.mNumCols; j++) {
165                 double sum = 0.0;
166                 for (int k = 0; k < mNumCols; k++) {
167                     sum += mData[i][k] * other.mData[k][j];
168                 }
169                 result.mData[i][j] = sum;
170             }
171         }
172         return result;
173     }
174 
175     Matrix Matrix::operator*(double scalar) const {
```

```
176     Matrix result(mNumRows, mNumCols);
177     for (int i = 0; i < mNumRows; i++) {
178         for (int j = 0; j < mNumCols; j++) {
179             result.mData[i][j] = mData[i][j] * scalar;
180         }
181     }
182     return result;
183 }
184 
185 Vector Matrix::operator*(const Vector& vec) const {
186     assert(mNumCols == vec.GetSize());
187     Vector result(mNumRows);
188     for (int i = 0; i < mNumRows; i++) {
189         double sum = 0.0;
190         for (int j = 0; j < mNumCols; j++) {
191             sum += mData[i][j] * vec[j];
192         }
193         result[i] = sum;
194     }
195     return result;
196 }
197 
198 Matrix operator*(double scalar, const Matrix& mat) {
199     return mat * scalar;
200 }
201 
202 // Compound assignment operators
203 Matrix& Matrix::operator+=(const Matrix& other) {
204     assert(mNumRows == other.mNumRows && mNumCols == other.mNumCols);
205     for (int i = 0; i < mNumRows; i++) {
206         for (int j = 0; j < mNumCols; j++) {
207             mData[i][j] += other.mData[i][j];
208         }
209     }
210     return *this;
211 }
212 
213 Matrix& Matrix::operator-=(const Matrix& other) {
214     assert(mNumRows == other.mNumRows && mNumCols == other.mNumCols);
215     for (int i = 0; i < mNumRows; i++) {
216         for (int j = 0; j < mNumCols; j++) {
217             mData[i][j] -= other.mData[i][j];
218         }
219     }
220     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);
```

```
266             for (int i = 1; i < mNumRows; i++) {
267                 for (int k = 0; k < mNumCols; k++) {
268                     if (k < j) submatrix.mData[i-1][k] = mData[i][k];
269                     else if (k > j) submatrix.mData[i-1][k-1] = mData[i][k↩
                            ];
270                 }
271             }
272             det += (j % 2 == 0 ? 1 : -1) * mData[0][j] * submatrix.↩
                    Determinant();
273         }
274         return det;
275     }
276 
277     Matrix Matrix::Inverse() const {
278         assert(IsSquare());
279         double det = Determinant();
280         assert(abs(det) > 1e-10); // Check if matrix is singular
281 
282         Matrix result(mNumRows, mNumCols);
283         // For simplicity, we'll implement a basic inverse calculation
284         // This is not the most efficient method for large matrices
285         if (mNumRows == 1) {
286             result.mData[0][0] = 1.0 / mData[0][0];
287             return result;
288         }
289         if (mNumRows == 2) {
290             result.mData[0][0] = mData[1][1] / det;
291             result.mData[0][1] = -mData[0][1] / det;
292             result.mData[1][0] = -mData[1][0] / det;
293             result.mData[1][1] = mData[0][0] / det;
294             return result;
295         }
296 
297         // For larger matrices, we'll use the adjugate method
298         // This is not the most efficient method, but it's straightforward
299         for (int i = 0; i < mNumRows; i++) {
300             for (int j = 0; j < mNumCols; j++) {
301                 // Create submatrix
302                 Matrix submatrix(mNumRows - 1, mNumCols - 1);
303                 for (int k = 0; k < mNumRows; k++) {
304                     for (int l = 0; l < mNumCols; l++) {
305                         if (k < i && l < j) submatrix.mData[k][l] = mData[↩
                                k][l];
306                         else if (k < i && l > j) submatrix.mData[k][l-1] =↩
                                 mData[k][l];
```

```
307                     else if (k > i && l < j) submatrix.mData[k-1][l] =↩
                            mData[k][l];
308                     else if (k > i && l > j) submatrix.mData[k-1][l-1]↩
                            = mData[k][l];
309                 }
310             }
311             result.mData[j][i] = ((i + j) % 2 == 0 ? 1 : -1) * ↩
                    submatrix.Determinant() / det;
312         }
313     }
314     return result;
315 }
316
317 Matrix Matrix::PseudoInverse() const {
318     // Moore-Penrose pseudo-inverse using SVD
319     // For simplicity, we'll use a basic implementation
320     // In practice, you would want to use a more robust method
321     Matrix A = *this;
322     Matrix AT = A.Transpose();
323     Matrix ATA = AT * A;
324     Matrix AAT = A * AT;
325
326     // Check if ATA is invertible
327     if (abs(ATA.Determinant()) > 1e-10) {
328         return AT * (A * AT).Inverse();
329     }
330     // Check if AAT is invertible
331     else if (abs(AAT.Determinant()) > 1e-10) {
332         return (AT * A).Inverse() * AT;
333     }
334     else {
335         // If neither is invertible, use the formula: A+ = (A^T * A)↩
                ^(-1) * A^T
336         // with regularization
337         double lambda = 1e-10; // Small regularization parameter
338         Matrix I(mNumCols, mNumCols);
339         for (int i = 0; i < mNumCols; i++) {
340             I.mData[i][i] = 1.0;
341         }
342         return (AT * A + lambda * I).Inverse() * AT;
343     }
344 }
345
346 // I/O operators
347 ostream& operator<<(ostream& os, const Matrix& mat) {
```

```
348     for (int i = 0; i < mat.mNumRows; i++) {
349         os << "[";
350         for (int j = 0; j < mat.mNumCols; j++) {
351             os << mat.mData[i][j];
352             if (j < mat.mNumCols - 1) os << ", ";
353         }
354         os << "]" << (i < mat.mNumRows - 1 ? "\n" : "");
355     }
356     return os;
357 }
358
359 istream& operator>>(istream& is, Matrix& mat) {
360     for (int i = 0; i < mat.mNumRows; i++) {
361         for (int j = 0; j < mat.mNumCols; j++) {
362             is >> mat.mData[i][j];
363         }
364     }
365     return is;
366 }
367
368 #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 $A^+ = A^T (A A^T)^{-1}$, 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

```
1 #ifndef LINEAR_SYSTEM_H
```

```
2  #define LINEAR_SYSTEM_H
3
4  #include "Matrix.h"
5
6  class LinearSystem {
7  protected:
8      int mSize;           // Size of the linear system
9      Matrix* mpA;         // Pointer to the coefficient matrix
10     Vector* mpb;         // Pointer to the right-hand side vector
11
12 public:
13     // Constructor
14     LinearSystem(const Matrix& A, const Vector& b);
15
16     // Prevent default construction and copying
17     LinearSystem() = delete;
18     LinearSystem(const LinearSystem& other) = delete;
19     LinearSystem& operator=(const LinearSystem& other) = delete;
20
21     // Destructor
22     virtual ~LinearSystem();
23
24     // Virtual solve method to be overridden by derived classes
25     virtual Vector Solve() const;
26
27     // Utility functions
28     int GetSize() const { return mSize; }
29     const Matrix& GetMatrix() const { return *mpA; }
30     const Vector& GetVector() const { return *mpb; }
31 };
32
33 // Derived class for positive definite symmetric linear systems
34 class PosSymLinSystem : public LinearSystem {
35 public:
36     // Constructor
37     PosSymLinSystem(const Matrix& A, const Vector& b);
38
39     // Override solve method to use conjugate gradient method
40     Vector Solve() const override;
41
42 private:
43     // Helper functions for conjugate gradient method
44     double ComputeAlpha(const Vector& r, const Vector& p, const Matrix↩
        & A) const;
45     double ComputeBeta(const Vector& r, const Vector& rNext) const;
```

```
46      bool CheckConvergence(const Vector& r, double tolerance = 1e-10) ↩
            const;
47  };
48
49  // LinearSystem constructor
50  LinearSystem::LinearSystem(const Matrix& A, const Vector& b) {
51      // Check if the system is valid
52      if (!A.IsSquare()) {
53          throw std::invalid_argument("Matrix A must be square");
54      }
55      if (A.GetNumRows() != b.GetSize()) {
56          throw std::invalid_argument("Matrix A and vector b must have ↩
                compatible dimensions");
57      }
58
59      mSize = A.GetNumRows();
60      mpA = new Matrix(A);
61      mpb = new Vector(b);
62  }
63
64  // LinearSystem destructor
65  LinearSystem::~LinearSystem() {
66      delete mpA;
67      delete mpb;
68  }
69
70  // LinearSystem Solve method using Gaussian elimination with partial ↩
        pivoting
71  Vector LinearSystem::Solve() const {
72      // Create copies of the matrix and vector to work with
73      Matrix A = *mpA;
74      Vector b = *mpb;
75      Vector x(mSize);
76
77      // Gaussian elimination with partial pivoting
78      for (int k = 1; k <= mSize; k++) {
79          // Find pivot
80          int maxRow = k;
81          double maxVal = std::abs(A(k, k));
82          for (int i = k + 1; i <= mSize; i++) {
83              double val = std::abs(A(i, k));
84              if (val > maxVal) {
85                  maxVal = val;
86                  maxRow = i;
87              }
```

```
88          }
89
90          // Check if matrix is singular
91          if (maxVal < 1e-10) {
92              throw std::runtime_error("Matrix is singular or nearly ↩
                    singular");
93          }
94
95          // Swap rows if necessary
96          if (maxRow != k) {
97              // Swap rows in matrix A
98              for (int j = 1; j <= mSize; j++) {
99                  std::swap(A(k, j), A(maxRow, j));
100             }
101             // Swap elements in vector b (using 1-based indexing)
102             std::swap(b(k), b(maxRow));
103         }
104
105         // Eliminate column k
106         for (int i = k + 1; i <= mSize; i++) {
107             double factor = A(i, k) / A(k, k);
108             A(i, k) = 0.0;  // Explicitly set to zero for numerical ↩
                    stability
109             for (int j = k + 1; j <= mSize; j++) {
110                 A(i, j) -= factor * A(k, j);
111             }
112             b(i) -= factor * b(k);
113         }
114     }
115
116     // Back substitution
117     for (int i = mSize; i >= 1; i--) {
118         double sum = 0.0;
119         for (int j = i + 1; j <= mSize; j++) {
120             sum += A(i, j) * x(j);
121         }
122         x(i) = (b(i) - sum) / A(i, i);
123     }
124
125     return x;
126 }
127
128 // PosSymLinSystem constructor
129 PosSymLinSystem::PosSymLinSystem(const Matrix& A, const Vector& b) : ↩
      LinearSystem(A, b) {
```

```
130     // Check if the matrix is symmetric
131     if (!A.IsSymmetric()) {
132         throw invalid_argument("Matrix must be symmetric for PosSymLinSystem");
133     }
134 
135     // Additional check for positive definiteness (optional)
136     // This is a simple check that might not catch all cases
137     for (int i = 1; i <= A.GetNumRows(); i++) {
138         if (A(i, i) <= 0) {
139             throw invalid_argument("Matrix must be positive definite");
140         }
141     }
142 }
143 
144 // PosSymLinSystem Solve method using conjugate gradient method
145 Vector PosSymLinSystem::Solve() const {
146     const int maxIterations = mSize;  // Maximum number of iterations
147     const double tolerance = 1e-10;   // Convergence tolerance
148     const double minResidual = 1e-15; // Minimum residual to prevent division by zero
149 
150     Vector x(mSize);                   // Initial guess (zero vector)
151     Vector r = *mpb - (*mpA) * x;     // Initial residual
152     Vector p = r;                      // Initial search direction
153     Vector rNext(mSize);              // Next residual
154     Vector Ap(mSize);                 // A * p
155 
156     double initialResidual = r.Norm();
157     if (initialResidual < minResidual) {
158         return x;  // Initial guess is already solution
159     }
160 
161     for (int iter = 0; iter < maxIterations; iter++) {
162         Ap = (*mpA) * p;
163         double alpha = ComputeAlpha(r, p, *mpA);
164 
165         // Update solution and residual
166         x += alpha * p;
167         rNext = r - alpha * Ap;
168 
169         // Check convergence
170         if (CheckConvergence(rNext, tolerance * initialResidual)) {
171             return x;
```

```
172         }
173
174         // Update search direction
175         double beta = ComputeBeta(r, rNext);
176         if (std::abs(beta) < minResidual) {
177             // If beta is too small, restart the algorithm
178             r = *mpb - (*mpA) * x;
179             p = r;
180         } else {
181             p = rNext + beta * p;
182             r = rNext;
183         }
184     }
185
186     throw std::runtime_error("Conjugate gradient method did not converge within " +
187                              std::to_string(maxIterations) + " iterations");
188 }
189
190 // Helper function to compute alpha in conjugate gradient method
191 double PosSymLinSystem::ComputeAlpha(const Vector& r, const Vector& p, const Matrix& A) const {
192     Vector Ap = A * p;
193     double pAp = p.DotProduct(Ap);
194     if (std::abs(pAp) < 1e-15) {
195         throw std::runtime_error("Matrix is not positive definite");
196     }
197     return r.DotProduct(r) / pAp;
198 }
199
200 // Helper function to compute beta in conjugate gradient method
201 double PosSymLinSystem::ComputeBeta(const Vector& r, const Vector& rNext) const {
202     double rDotR = r.DotProduct(r);
203     if (std::abs(rDotR) < 1e-15) {
204         return 0.0;  // Prevent division by zero
205     }
206     return rNext.DotProduct(rNext) / rDotR;
207 }
208
209 // Helper function to check convergence in conjugate gradient method
210 bool PosSymLinSystem::CheckConvergence(const Vector& r, double tolerance) const {
211     return r.Norm() <= tolerance;
```

```
212 }
213
214 #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. The project code and documentation are available in the GitHub repository at https://github.com/VTT-Icebear/Tiny$project$