My Project

Generated by Doxygen 1.9.8

Chapter 1

Class Index

1.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 3

Class Documentation

3.1 Linear_Solvers Class Reference

Class containing static methods for solving linear systems.

`#include <matrix_math.hpp>`

3.1.1 Detailed Description

Class containing static methods for solving linear systems.

using various methods including LU and QR decomposition

The documentation for this class was generated from the following file:

- matrix_math.hpp

3.2 LUResult Struct Reference

Result of LU decomposition of a matrix.

`#include <matrix_math.hpp>`

Public Attributes

- std::vector< std::vector< double > > **L**
  Lower triangular matrix.
- std::vector< std::vector< double > > **U**
  Upper triangular matrix.
- std::vector< int > **P**
  Permutation vector.

3.2.1 Detailed Description

Result of LU decomposition of a matrix.

Stores the lower triangular (L), upper triangular (U) matrices, and the permutation vector (P) from the decomposition

The documentation for this struct was generated from the following file:

- matrix_math.hpp

3.3 LUResult_to_pass Struct Reference

Collaboration diagram for LUResult_to_pass:

Public Attributes

- Matrix **L**
- Matrix **U**
- std::vector< int > **P**

The documentation for this struct was generated from the following file:

- matrix_math.hpp

3.4 Matrix Class Reference

Matrix class implementation for mathematical operations.

`#include <matrix_math.hpp>`

Public Member Functions

- Matrix (int rs=1, int cs=1)
  Constructor for Matrix.
- Matrix (const Matrix &other)
  Copy constructor for Matrix.
- Matrix & operator= (const Matrix &other)
  Assignment operator for Matrix.
- Matrix operator+ (const Matrix &other) const
  Matrix addition operator.
- Matrix operator- (const Matrix &other) const
  Matrix subtraction operator.
- LUResult L_U () const
  Computes the LU Decomposition of the matrix.
- QRresult QR_fact () const
  Computes the QR Decomposition of the matrix.
- Matrix operator* (const Matrix &other) const
  Matrix multiplication operator.
- My_Vec multiply (const My_Vec &x) const
  Multiplies the matrix with a vector.
- Matrix Transpose () const
  Transposes the matrix.
- Matrix Scalar_Mul (double k) const
  Scalar multiplication of the matrix.

Static Public Member Functions

- static My_Vec UV (int i, int L)

  Creates a unit vector with a 1 at position i.
- static Matrix Embed (const Matrix &Householder, const Matrix &A)

  Embeds a Householder matrix into a larger identity matrix.
- static Matrix Outer_Product (const My_Vec &u, const My_Vec &v)

  Static method to compute the outer product of two vectors.
- static Matrix eye (int a)

  Static method to create an identity matrix of size a.
- static Matrix Ones (int a, int b)

  Static method to create a matrix of ones.
- static Matrix Zeros (int a, int b)

  Static method to create a matrix of zeros.

Public Attributes

- int **rows**

  Number of rows in the matrix.
- int **cols**

  Number of columns in the matrix.
- std::vector< std::vector< double > > **MyMAT**

  Internal matrix storage.

3.4.1 Detailed Description

Matrix class implementation for mathematical operations.

Provides basic matrix operations including addition, subtraction, multiplication, and various matrix factorizations (LU, QR)

3.4.2 Constructor & Destructor Documentation

3.4.2.1 Matrix() [1/2]

```
Matrix::Matrix (
            int rs = 1,
            int cs = 1 )  [inline]
```

Constructor for Matrix.

Parameters

rs	Number of rows (defaults to 1)
cs	Number of columns (defaults to 1)

3.4.2.2 Matrix() [2/2]

```
Matrix::Matrix (
            const Matrix & other )  [inline]
```

Copy constructor for Matrix.

Parameters

other	The matrix to copy

3.4.3 Member Function Documentation

3.4.3.1 Embed()

```
static Matrix Matrix::Embed (
            const Matrix & Householder,
            const Matrix & A )  [inline], [static]
```

Embeds a Householder matrix into a larger identity matrix.

Parameters

Householder	The Householder matrix to embed
A	The original matrix

Returns

The embedded matrix

3.4.3.2 eye()

```
static Matrix Matrix::eye (
            int a )  [inline], [static]
```

Static method to create an identity matrix of size a.

Parameters

a	Size of the matrix (number of rows and columns)

Returns

Identity matrix of size a

3.4.3.3 L_U()

```
LUResult Matrix::L_U ( ) const  [inline]
```

Computes the LU Decomposition of the matrix.

Returns

LUResult structure containing L, U matrices and P vector

Exceptions

std::invalid_argument	if the matrix is not square
std::runtime_error	if the matrix is singular or nearly singular

3.4.3.4 multiply()

```
My_Vec Matrix::multiply (
            const My_Vec & x ) const  [inline]
```

Multiplies the matrix with a vector.

Parameters

x	The vector to multiply with

Returns

Resulting vector after multiplication

Exceptions

std::invalid_argument	if dimensions are not compatible

3.4.3.5 Ones()

```
static Matrix Matrix::Ones (
            int a,
            int b )  [inline], [static]
```

Static method to create a matrix of ones.

Parameters

a	Number of rows
b	Number of columns

Returns

Matrix of size a x b with all elements set to 1

3.4.3.6 operator∗()

```
Matrix Matrix::operator* (
            const Matrix & other ) const  [inline]
```

Matrix multiplication operator.

Parameters

other	The matrix to multiply with

Returns

Resulting matrix after multiplication

Exceptions

std::invalid_argument	if dimensions are not compatible

3.4.3.7 operator+()

```
Matrix Matrix::operator+ (
            const Matrix & other ) const  [inline]
```

Matrix addition operator.

Parameters

other	The matrix to add

Returns

Resulting matrix after addition

Exceptions

std::invalid_argument	if matrices have different dimensions

3.4.3.8 operator-()

```
Matrix Matrix::operator- (
            const Matrix & other ) const  [inline]
```

Matrix subtraction operator.

Parameters

other	The matrix to subtract

Returns

Resulting matrix after subtraction

Exceptions

std::invalid_argument	if matrices have different dimensions

3.4.3.9 operator=()

```
Matrix & Matrix::operator= (
            const Matrix & other )  [inline]
```

Assignment operator for Matrix.

Parameters

other	The matrix to assign

Returns

Reference to this matrix

3.4.3.10 Outer_Product()

```
static Matrix Matrix::Outer_Product (
            const My_Vec & u,
            const My_Vec & v )  [inline], [static]
```

Static method to compute the outer product of two vectors.

Parameters

u	First vector
v	Second vector

Returns

Matrix resulting from the outer product

3.4.3.11 QR_fact()

```
QRresult Matrix::QR_fact ( ) const  [inline]
```

Computes the QR Decomposition of the matrix.

Returns

QRresult structure containing Q, R matrices

3.4.3.12 Scalar_Mul()

```
Matrix Matrix::Scalar_Mul (
            double k ) const  [inline]
```

Scalar multiplication of the matrix.

Parameters

k	Scalar value

Returns

New matrix resulting from scalar multiplication

3.4.3.13 Transpose()

```
Matrix Matrix::Transpose ( ) const  [inline]
```

Transposes the matrix.

Returns

New matrix that is the transpose of this matrix

3.4.3.14 UV()

```
static My_Vec Matrix::UV (
            int i,
            int L )  [inline], [static]
```

Creates a unit vector with a 1 at position i.

Parameters

i	Position of the 1 in the unit vector
L	Total length of the vector

Returns

Unit vector with 1 at position i

3.4.3.15 Zeros()

```
static Matrix Matrix::Zeros (
            int a,
            int b )  [inline], [static]
```

Static method to create a matrix of zeros.

Parameters

a	Number of rows
b	Number of columns

Returns

Matrix of size a x b with all elements set to 0

The documentation for this class was generated from the following file:

- matrix_math.hpp

3.5 My_Vec Class Reference

Vector class for mathematical operations.

```
#include <matrix_math.hpp>
```

Public Member Functions

- My_Vec (int l=1)
  Construct a new vector.
- My_Vec operator+ (const My_Vec &other) const
  Add two vectors.
- My_Vec operator- (const My_Vec &other) const
  Vector subtraction operator.
- double Norm () const
  Computes the Euclidean norm (magnitude) of the vector.
- double dot (const My_Vec &other) const
  Computes the dot product with another vector.
- My_Vec Scalar_Mul (double k) const
  Scalar multiplication of the vector.
- My_Vec (const My_Vec &other)
  Copy constructor for My_Vec.
- My_Vec & operator= (const My_Vec &other)
  Assignment operator for My_Vec.

Static Public Member Functions

- static My_Vec ones (int a)

  Creates a vector of ones.
- static My_Vec unit_vec (int i, int L)

  Creates a unit vector with a 1 at position i.
- static My_Vec Zeros (const int &i)

  Creates a zero vector of length i.

Public Attributes

- int **length**

  Length of the vector.
- std::vector< double > **myvector**

  Vector data storage.

3.5.1 Detailed Description

Vector class for mathematical operations.

Implements a mathematical vector with common operations like addition, subtraction, dot product, and scalar multiplication

3.5.2 Constructor & Destructor Documentation

3.5.2.1 My_Vec() [1/2]

```
My_Vec::My_Vec (
            int l = 1 )  [inline]
```

Construct a new vector.

Parameters

l	Length of the vector (defaults to 1)

3.5.2.2 My_Vec() [2/2]

```
My_Vec::My_Vec (
            const My_Vec & other )  [inline]
```

Copy constructor for My_Vec.

Parameters

other	The vector to copy

3.5.3 Member Function Documentation

3.5.3.1 dot()

```
double My_Vec::dot (
            const My_Vec & other ) const  [inline]
```

Computes the dot product with another vector.

Parameters

other	The other vector

Returns

Dot product result

Exceptions

std::invalid_argument	if vectors have different lengths

3.5.3.2 Norm()

```
double My_Vec::Norm ( ) const  [inline]
```

Computes the Euclidean norm (magnitude) of the vector.

Returns

Norm of the vector

3.5.3.3 ones()

```
static My_Vec My_Vec::ones (
            int a )  [inline], [static]
```

Creates a vector of ones.

Parameters

a	Length of the vector

Returns

Vector with all elements set to 1

3.5.3.4 operator+()

```
My_Vec My_Vec::operator+ (
            const My_Vec & other ) const  [inline]
```

Add two vectors.

Parameters

other	Vector to add to this one

Returns

New vector containing the sum

Exceptions

std::invalid_argument	if vectors have different lengths

3.5.3.5 operator-()

```
My_Vec My_Vec::operator- (
            const My_Vec & other ) const  [inline]
```

Vector subtraction operator.

Parameters

other	The vector to subtract

Returns

Result of vector subtraction

Exceptions

std::invalid_argument	if vectors have different lengths

3.5.3.6 operator=()

```
My_Vec & My_Vec::operator= (
            const My_Vec & other )  [inline]
```

Assignment operator for My_Vec.

Parameters

other	The vector to assign

Returns

Reference to this vector

3.5.3.7 Scalar_Mul()

```
My_Vec My_Vec::Scalar_Mul (
            double k ) const  [inline]
```

Scalar multiplication of the vector.

Parameters

k	Scalar value

Returns

New vector resulting from scalar multiplication

3.5.3.8 unit_vec()

```
static My_Vec My_Vec::unit_vec (
            int i,
            int L )  [inline], [static]
```

Creates a unit vector with a 1 at position i.

Parameters

i	Position of the 1 in the unit vector
L	Total length of the vector

Returns

Unit vector with 1 at position i

3.5.3.9 Zeros()

```
static My_Vec My_Vec::Zeros (
            const int & i )  [inline], [static]
```

Creates a zero vector of length i.

Parameters

i	Length of the zero vector

Returns

Zero vector of length i

The documentation for this class was generated from the following file:

- matrix_math.hpp

3.6 QR_result_to_pass Struct Reference

Collaboration diagram for QR_result_to_pass:

Public Attributes

- Matrix **Q**
- Matrix **R**

The documentation for this struct was generated from the following file:

- matrix_math.hpp

3.7 QRresult Struct Reference

Result of QR decomposition of a matrix.

`#include <matrix_math.hpp>`

Public Attributes

- std::vector< std::vector< double > > **Q**

  Orthogonal matrix.
- std::vector< std::vector< double > > **R**

  Upper triangular matrix.

3.7.1 Detailed Description

Result of QR decomposition of a matrix.

Stores the orthogonal matrix Q and upper triangular matrix R

The documentation for this struct was generated from the following file:

- matrix_math.hpp

Chapter 4

File Documentation

4.1 matrix_math.hpp File Reference

Linear algebra and matrix mathematics library.

```
#include <iostream>
#include <vector>
#include <cmath>
```

Include dependency graph for matrix_math.hpp:

4.2 matrix_math.hpp

Go to the documentation of this file.

```
00001
00008 //gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
00009 // Standard library includes for basic operations
00010 #include <iostream>
00011 #include <vector>
00012 #include <cmath>
00013
00014 #ifndef MATRIX_MATH_HPP
00015 #define MATRIX_MATH_HPP
00016
00023 struct LUResult {
00024     std::vector<std::vector<double» L;
00025     std::vector<std::vector<double» U;
00026     std::vector<int> P;
00027 };
00028
00034 struct QRresult {
00035      std::vector<std::vector<double» Q;
00036      std::vector<std::vector<double» R;
00037 };
00038
00045 class My_Vec {
00046 public:
00047     int length;
00048     std::vector<double> myvector;
00049
00054     My_Vec(int l=1) {
00055         length = l;
00056         myvector.resize(length);
00057     }
00058
00065     My_Vec operator+ (const My_Vec& other) const {
00066         if(this->length != other.length) {
00067             throw std::invalid_argument("Invalid lengths");
00068         }
00069         My_Vec Vec(this->length);
00070         for (int i=0; i<this->length; i++) {
```

```
00071             Vec.myvector[i] = this->myvector[i] + other.myvector[i];
00072         }
00073         return Vec;
00074     }
00075
00081     static My_Vec ones(int a) {
00082         My_Vec ones_vec(a);
00083         for(int i=0; i<a; i++) {
00084             ones_vec.myvector[i] = 1;
00085         }
00086         return ones_vec;
00087     }
00088
00095     My_Vec operator- (const My_Vec& other) const{
00096
00097         if(this->length != other.length) {
00098             throw std::invalid_argument("Invalid lengths");
00099         }
00100         My_Vec Vec(this->length);
00101
00102         for (int i=0; i<this->length; i++) {
00103             Vec.myvector[i] = this->myvector[i] - other.myvector[i];
00104         }
00105         return Vec;
00106     }
00107
00112     double Norm() const {
00113         double a=0;
00114         for (int i=0; i<this->length; i++) {
00115             a += pow(this->myvector[i], 2.0);
00116         }
00117         return sqrt(a);
00118     }
00119
00126     double dot (const My_Vec& other) const {
00127         if(this->length != other.length) {
00128             throw std::invalid_argument("Invalid lengths");
00129         }
00130
00131         double a=0;
00132         for(int i=0; i<this->length; i++) {
00133             a += this->myvector[i] * other.myvector[i];
00134         }
00135         return a;
00136     }
00137
00143     My_Vec Scalar_Mul(double k) const {
00144         My_Vec New(this->length);
00145         for (int i=0; i<this->length; i++) {
00146             New.myvector[i] = k * this->myvector[i];
00147         }
00148         return New;
00149     }
00150
00155     My_Vec(const My_Vec& other)
00156     : length(other.length), myvector(other.myvector) {};
00157
00163     My_Vec& operator=(const My_Vec& other) {
00164         if (this != &other) {
00165             length = other.length;
00166             myvector = other.myvector;
00167         }
00168         return *this;
00169     }
00170
00177     static My_Vec unit_vec(int i, int L) {
00178         My_Vec unit_vec(L);
00179         for (int j=0; j<L; j++) {
00180             unit_vec.myvector[j] = (j == i) ? 1 : 0;
00181         }
00182         return unit_vec;
00183     }
00184
00190     static My_Vec Zeros(const int& i) {
00191         My_Vec unit_vec(i);
00192         for (int j=0; j<i; j++) {
00193             unit_vec.myvector[j] = 0;
00194         }
00195         return unit_vec;
00196     }
00197 };
00198
00205 class Matrix {
00206 public:
00207     int rows;
00208     int cols;
00209     std::vector<std::vector<double>> MyMAT;
```

```
00210
00216     Matrix(int rs=1, int cs=1) {
00217         rows = rs;
00218         cols = cs;
00219         MyMAT.resize(rs, std::vector<double>(cs));
00220     }
00221
00226     Matrix(const Matrix& other)
00227     : rows(other.rows), cols(other.cols), MyMAT(other.MyMAT) {}
00228
00234     Matrix& operator=(const Matrix& other) {
00235         if (this != &other) {
00236             rows = other.rows;
00237             cols = other.cols;
00238             MyMAT = other.MyMAT;
00239         }
00240         return *this;
00241     }
00242
00249     Matrix operator+(const Matrix& other) const {
00250         Matrix New_Mat(other.rows, other.cols);
00251         if (this->cols != other.cols || other.rows != this->rows) {
00252             throw std::invalid_argument("Dimension mismatch");
00253         } else {
00254             for (int i=0; i<rows; i++) {
00255                 for (int j=0; j<cols; j++) {
00256                     New_Mat.MyMAT[i][j] = other.MyMAT[i][j] + this->MyMAT[i][j];
00257                 }
00258             }
00259         }
00260         return New_Mat;
00261     }
00262
00269     Matrix operator- (const Matrix& other) const {
00270         Matrix New_Mat(other.rows, other.cols);
00271         if (this->cols != other.cols || other.rows != this->rows) {
00272             throw std::invalid_argument("Dimension mismatch");
00273         } else {
00274             for (int i=0; i<rows; i++) {
00275                 for (int j=0; j<cols; j++) {
00276                     New_Mat.MyMAT[i][j] = this->MyMAT[i][j] - other.MyMAT[i][j];
00277                 }
00278             }
00279         }
00280         return New_Mat;
00281     }
00282
00289     LUResult L_U() const {
00290         // Check for square matrix
00291         if(this->rows != this->cols) {
00292             throw std::invalid_argument("Not square");
00293         }
00294
00295         Matrix L = eye(this->rows);
00296         Matrix U = Zeros(this->rows, this->cols);
00297         std::vector<int> P(this->rows);
00298         for(int i = 0; i < this->rows; i++) {
00299             P[i] = i;
00300         }
00301
00302         Matrix A_work(*this);
00303
00304         for(int j = 0; j < this->cols; j++) {
00305             int pivot_row = j;
00306             double max_val = std::abs(A_work.MyMAT[j][j]);
00307
00308             for(int i = j+1; i < this->rows; i++) {
00309                 if(std::abs(A_work.MyMAT[i][j]) > max_val) {
00310                     pivot_row = i;
00311                     max_val = std::abs(A_work.MyMAT[i][j]);
00312                 }
00313             }
00314
00315             if(pivot_row != j) {
00316                 for(int k = 0; k < this->cols; k++) {
00317                     std::swap(A_work.MyMAT[j][k], A_work.MyMAT[pivot_row][k]);
00318                 }
00319
00320                 std::swap(P[j], P[pivot_row]);
00321
00322                 if(j > 0) {
00323                     for(int k = 0; k < j; k++) {
00324                         std::swap(L.MyMAT[j][k], L.MyMAT[pivot_row][k]);
00325                     }
00326                 }
00327             }
00328
```

```
00329             if(std::abs(A_work.MyMAT[j][j]) < 1e-10) {
00330                 throw std::runtime_error("Matrix is singular or nearly singular");
00331             }
00332 
00333             for(int i = 0; i <= j; i++) {
00334                 double sum = 0.0;
00335                 for(int k = 0; k < i; k++) {
00336                     sum += L.MyMAT[i][k] * U.MyMAT[k][j];
00337                 }
00338                 U.MyMAT[i][j] = A_work.MyMAT[i][j] - sum;
00339             }
00340 
00341             for(int i = j+1; i < this->rows; i++) {
00342                 double sum = 0.0;
00343                 for(int k = 0; k < j; k++) {
00344                     sum += L.MyMAT[i][k] * U.MyMAT[k][j];
00345                 }
00346                 L.MyMAT[i][j] = (A_work.MyMAT[i][j] - sum) / U.MyMAT[j][j];
00347             }
00348         }
00349 
00350         LUResult result;
00351         result.L = L.MyMAT;
00352         result.U = U.MyMAT;
00353         result.P = P;
00354         return result;
00355     }
00356 
00363     static My_Vec UV(int i, int L) {
00364         My_Vec unit_vec(L);
00365         for (int j=0; j<L; j++) {
00366             unit_vec.myvector[j] = (j == i) ? 1 : 0;
00367         }
00368         return unit_vec;
00369     }
00370 
00377     static Matrix Embed(const Matrix& Householder, const Matrix& A) {
00378         Matrix Hp = eye(A.rows);
00379         int i = A.rows - Householder.rows;
00380         for (int a = 0; a < Householder.rows; a++) {
00381             for (int b = 0; b < Householder.cols; b++) {
00382                 Hp.MyMAT[i+a][i+b] = Householder.MyMAT[a][b];
00383             }
00384         }
00385         return Hp;
00386     }
00387 
00392     QRresult QR_fact() const {
00393         Matrix Q = eye(this->rows);
00394         Matrix Aupdate = *this;
00395 
00396         for (int i = 0; i < std::min(this->rows-1, this->cols); i++) {
00397             My_Vec vecx(this->rows - i);
00398             for (int row = i; row < this->rows; row++) {
00399                 vecx.myvector[row - i] = Aupdate.MyMAT[row][i];
00400             }
00401 
00402             double n = vecx.Norm();
00403 
00404             My_Vec unit = My_Vec::unit_vec(0, this->rows - i);  // Use consistent naming
00405 
00406             My_Vec reflec_vec;
00407             if (vecx.myvector[0] < 0) {
00408                 reflec_vec = vecx + unit.Scalar_Mul(n);
00409             } else {
00410                 reflec_vec = vecx - unit.Scalar_Mul(n);
00411             }
00412 
00413             double normalize_ref = reflec_vec.Norm();
00414 
00415             // Check for zero vector (avoid division by zero)
00416             if (normalize_ref < 1e-10) {
00417                 continue;  // Skip this iteration if vector is too small
00418             }
00419 
00420             My_Vec V = reflec_vec.Scalar_Mul(1.0 / normalize_ref);
00421 
00422             // Create Householder matrix H = I - 2*vv^T
00423             Matrix I = Matrix::eye(V.length);
00424             Matrix vvT = Matrix::Outer_Product(V, V);
00425             Matrix Householder = I - vvT.Scalar_Mul(2.0);
00426 
00427             // Embed the Householder matrix into a larger identity matrix
00428             Matrix Hprime = Matrix::eye(this->rows);
00429             for (int r = 0; r < Householder.rows; r++) {
00430                 for (int c = 0; c < Householder.cols; c++) {
00431                     Hprime.MyMAT[i + r][i + c] = Householder.MyMAT[r][c];
```

```
00432                 }
00433             }
00434
00435             Aupdate = Hprime * Aupdate;
00436             Q = Q * Hprime;
00437         }
00438
00439         QRresult QR;
00440         std::vector<std::vector<double» Q_n = Q.Transpose().MyMAT;
00441         std::vector<std::vector<double» R_n = Aupdate.MyMAT;
00442
00443         QR.Q = Q_n;
00444         QR.R = R_n;
00445         return QR;
00446     }
00447
00454     Matrix operator*(const Matrix& other) const {
00455         if (this->cols != other.rows) {
00456             throw std::invalid_argument("Dimension mismatch");
00457         }
00458
00459         Matrix Ans(this->rows, other.cols); // Ensure constructor handles allocation
00460
00461         for (int i = 0; i < this->rows; ++i) {
00462             for (int j = 0; j < other.cols; ++j) {
00463                 double sum_of_multiples = 0.0;
00464                 for (int k = 0; k < this->cols; ++k) {
00465                     sum_of_multiples += this->MyMAT[i][k] * other.MyMAT[k][j];
00466                 }
00467                 Ans.MyMAT[i][j] = sum_of_multiples;
00468             }
00469         }
00470
00471         return Ans;
00472     }
00473
00474
00481     My_Vec multiply(const My_Vec& x) const {
00482         if(this->cols != x.length) {
00483             throw std::invalid_argument("Dimension mismatch");
00484         }
00485
00486         My_Vec ans(this->rows);
00487
00488         for (int i=0; i<this->rows; i++) {
00489             double dot_prod = 0;
00490             for(int j=0; j<this->cols; j++) {
00491                 dot_prod += this->MyMAT[i][j] * x.myvector[j];
00492             }
00493             ans.myvector[i] = dot_prod;
00494         }
00495
00496         return ans;
00497     }
00498
00505     static Matrix Outer_Product(const My_Vec& u, const My_Vec& v) {
00506         Matrix Output(u.length, v.length);
00507         for(int i=0; i<u.length; i++) {
00508             for(int j=0; j<v.length; j++) {
00509                 Output.MyMAT[i][j] = u.myvector[i] * v.myvector[j];
00510             }
00511         }
00512         return Output;
00513     }
00514
00519     Matrix Transpose() const {
00520         Matrix New_Mat(this->cols, this->rows);
00521         for (int i = 0; i<rows; i++) {
00522             for(int j=0; j<cols; j++) {
00523                 New_Mat.MyMAT[j][i] = this->MyMAT[i][j];
00524             }
00525         }
00526         return New_Mat;
00527     }
00528
00534     Matrix Scalar_Mul(double k) const {
00535         Matrix new_mat(this->rows, this->cols);
00536
00537         for (int i = 0; i<rows; i++) {
00538             for(int j=0; j<cols; j++) {
00539                 new_mat.MyMAT[i][j] = this->MyMAT[i][j] * k;
00540             }
00541         }
00542         return new_mat;
00543     }
00544
00550     static Matrix eye(int a) {
```

```
00551         Matrix Identity(a, a);
00552         for(int i=0; i<a; i++) {
00553             Identity.MyMAT[i][i] = 1;
00554         }
00555         return Identity;
00556     }
00557
00564     static Matrix Ones(int a, int b) {
00565         Matrix Ones(a, b);
00566         for(int i=0; i<a; i++) {
00567             for(int j=0; j<b; j++) {
00568                 Ones.MyMAT[i][j] = 1;
00569             }
00570         }
00571         return Ones;
00572     }
00573
00580     static Matrix Zeros(int a, int b) {
00581         Matrix Zeros(a, b);
00582         for(int i=0; i<a; i++) {
00583             for(int j=0; j<b; j++) {
00584                 Zeros.MyMAT[i][j] = 0;
00585             }
00586         }
00587         return Zeros;
00588     }
00589 };
00590
00596 class Linear_Solvers {
00597     static My_Vec SolveLU(const Matrix& A, const My_Vec& b);
00598     static My_Vec SolveQR(const Matrix& A, const My_Vec& b);
00599     static My_Vec Inverse(const Matrix& A);
00600
00601     static My_Vec ForwardSubstitution(const Matrix& L, const My_Vec& b);
00602     static My_Vec BackwardSubstitution(const Matrix& U, const My_Vec& y);
00603 };
00604
00605 struct LUResult_to_pass {
00606    Matrix L;
00607    Matrix U;
00608    std::vector<int> P;
00609 };
00610
00611
00612 struct QR_result_to_pass {
00613      Matrix Q;
00614      Matrix R;
00615 };
00616
00617
00618 LUResult_to_pass conv_LU(const LUResult& LU) {
00619     int a = LU.L.size();
00620     int b = LU.L[0].size();
00621
00622     Matrix L(a, b);
00623     L.MyMAT = LU.L;
00624
00625
00626     int c = LU.U.size();
00627     int d = LU.U[0].size();
00628
00629     Matrix U(c, d);
00630     U.MyMAT = LU.U;
00631
00632     LUResult_to_pass LU_new;
00633
00634     LU_new.L = L;
00635     LU_new.U = U;
00636     LU_new.P = LU.P;
00637     return LU_new;
00638 }
00639
00640
00641 QR_result_to_pass conv_QR(const QRresult& QR) {
00642     int a = QR.Q.size();
00643     int b = QR.Q[0].size();
00644
00645     Matrix Q(a, b);
00646     Q.MyMAT = QR.Q;
00647
00648
00649     int c = QR.R.size();
00650     int d = QR.R[0].size();
00651
00652     Matrix R(c, d);
00653     R.MyMAT = QR.R;
00654
```

```
00655     QR_result_to_pass QR_new;
00656
00657     QR_new.Q = Q;
00658     QR_new.R = R;
00659     return QR_new;
00660 }
00661
00662
00663 #endif
```

Index