

C++ controls toolbox test

Generated by Doxygen 1.9.8

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Analysis	A class for analyzing properties of discrete-time state space systems	??
Forms	A class for transforming state space systems into various canonical forms	??
Linear_Solvers	Class containing static methods for solving linear systems	??
LUResult	Result of LU decomposition of a matrix	??
LUResult_to_pass	??
Matrix	Matrix class implementation for mathematical operations	??
My_Vec	Vector class for mathematical operations	??
QR_result_to_pass	??
QRresult	Result of QR decomposition of a matrix	??

Chapter 2

File Index

2.1 File List

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

analysis.hpp	??
forms.hpp	??
linear_solvers.hpp		
Linear system solvers implementation	??
matrix_math.hpp		
Linear algebra and matrix mathematics library	??

Chapter 3

Data Structure Documentation

3.1 Analysis Class Reference

A class for analyzing properties of discrete-time state space systems.

```
#include <analysis.hpp>
```

Public Member Functions

- bool [is_controllable](#) (const Discrete_StateSpace_System &System)
Checks if the system is controllable.
- bool [is_observable](#) (const Discrete_StateSpace_System &System)
Checks if the system is observable.
- bool [Linear_Stability_discrete](#) (const Discrete_StateSpace_System &System)
Checks the stability of a discrete-time linear system.
- bool [is_stabalizable_cont](#) (const Discrete_StateSpace_System &System)
Checks if the continuous-time system is stabilizable.
- bool [is_detectable_cont](#) (const Discrete_StateSpace_System &System)
Checks if the continuous-time system is detectable.
- bool [minimality_test_cont](#) (const Discrete_StateSpace_System &System)
Performs the minimality test for continuous-time systems.
- std::tuple< Eigen::MatrixXd, Eigen::MatrixXd > [controllability_decomposition](#) (const Discrete_StateSpace_System &System)
Computes the controllability decomposition of a discrete-time state space system.
- std::tuple< Eigen::MatrixXd, Eigen::MatrixXd > [observability_decomposition](#) (const Discrete_StateSpace_System &System)
Computes the observability decomposition of a discrete-time state space system.
- Eigen::MatrixXd [compute_controllability_gramian](#) (const Discrete_StateSpace_System &System)
Computes the controllability gramian for a discrete-time state space system.
- Eigen::MatrixXd [compute_observability_gramian](#) (const Discrete_StateSpace_System &System)
Computes the observability gramian for a discrete-time state space system.
- std::vector< std::complex< double > > [poles](#) (const Discrete_StateSpace_System &System)
Computes the poles of the system (eigenvalues of A matrix)
- std::vector< std::complex< double > > [generate_z_grid](#) (double r_min, double r_max, int r_samples, int theta_samples)
Generates a grid of points in the z-plane for root locus analysis.
- std::vector< std::complex< double > > [zeros](#) (const Discrete_StateSpace_System &System, const double &r_min, const double &r_max, const int &r_samples, const int &theta_samples)
Finds the zeros of the transfer function (poles of the closed-loop system)
- Discrete_StateSpace_System [Kalman-Decomp](#) (const Discrete_StateSpace_System &System)
Performs Kalman decomposition for state estimation.

Static Public Member Functions

- static Eigen::MatrixXd [compute_controllability_matrix](#) (const Discrete_StateSpace_System &System)
Computes the controllability matrix for a discrete-time state space system.
- static Eigen::MatrixXd [compute_observability_matrix](#) (const Discrete_StateSpace_System &System)
Computes the observability matrix for a discrete-time state space system.
- static bool [Linear_Stability_cont](#) (const Discrete_StateSpace_System &System)
Checks the stability of a continuous-time linear system.

3.1.1 Detailed Description

A class for analyzing properties of discrete-time state space systems.

This class provides methods for analyzing fundamental system properties including:

- Controllability: ability to drive the system to any desired state
- Observability: ability to determine any initial state from output measurements
- Stability: behavior of the system's free response
- Stabilizability: ability to stabilize unstable modes through feedback

For a discrete-time state space system: *

$$\begin{aligned}x[k+1] &= Ax[k] + Bu[k] \\ y[k] &= Cx[k] + Du[k]\end{aligned}$$

3.1.2 Member Function Documentation

3.1.2.1 compute_controllability_gramian()

```
Eigen::MatrixXd Analysis::compute_controllability_gramian (
    const Discrete_StateSpace_System & System ) [inline]
```

Computes the controllability gramian for a discrete-time state space system.

The controllability gramian is computed using the formula: $W_c = \int_0^\infty A(t)B (A(t)B)^T dt$ where $A(t)$ is the state transition matrix.

Parameters

<i>System</i>	The discrete state space system to analyze
---------------	--

Returns

Eigen::MatrixXd The controllability gramian

3.1.2.2 compute_controllability_matrix()

```
static Eigen::MatrixXd Analysis::compute_controllability_matrix (
    const Discrete_StateSpace_System & System ) [inline], [static]
```

Computes the controllability matrix for a discrete-time state space system.

The controllability matrix is defined as:

$$C = [B \quad AB \quad A^2B \quad \dots \quad A^{n-1}B]$$

where:

- n is the number of states
- A is the state matrix
- B is the input matrix

This matrix has size $n \times nm$, where:

- n is the number of states
- m is the number of inputs

Parameters

<i>System</i>	The discrete state space system to analyze
---------------	--

Returns

Eigen::MatrixXd The controllability matrix

3.1.2.3 compute_observability_gramian()

```
Eigen::MatrixXd Analysis::compute_observability_gramian (
    const Discrete_StateSpace_System & System ) [inline]
```

Computes the observability gramian for a discrete-time state space system.

The observability gramian is computed using the formula: $W_o = \int_0^\infty (C_A(t))^T (C_A(t)) dt$ where $_A(t)$ is the state transition matrix.

Parameters

<i>System</i>	The discrete state space system to analyze
---------------	--

Returns

Eigen::MatrixXd The observability gramian

3.1.2.4 compute_observability_matrix()

```
static Eigen::MatrixXd Analysis::compute_observability_matrix (
    const Discrete_StateSpace_System & System ) [inline], [static]
```

Computes the observability matrix for a discrete-time state space system.

The observability matrix is defined as:

$$\mathcal{O} = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix}$$

This matrix has size $p \times n$, where:

- n is the number of states
- p is the number of outputs

Parameters

<i>System</i>	The discrete state space system to analyze
---------------	--

Returns

Eigen::MatrixXd The observability matrix

3.1.2.5 controllability_decomposition()

```
std::tuple< Eigen::MatrixXd, Eigen::MatrixXd > Analysis::controllability_decomposition (
    const Discrete_StateSpace_System & System ) [inline]
```

Computes the controllability decomposition of a discrete-time state space system.

This method computes the controllability matrix and performs QR decomposition to find the invariant subspaces of the system.

Parameters

<i>System</i>	The discrete state space system to analyze
---------------	--

Returns

std::tuple<Eigen::MatrixXd, Eigen::MatrixXd> The controllable and uncontrollable subspaces

3.1.2.6 generate_z_grid()

```
std::vector< std::complex< double > > Analysis::generate_z_grid (
    double r_min,
    double r_max,
    int r_samples,
    int theta_samples ) [inline]
```

Generates a grid of points in the z-plane for root locus analysis.

Parameters

<i>r_min</i>	The minimum radius for the grid
<i>r_max</i>	The maximum radius for the grid
<i>r_samples</i>	The number of radial samples
<i>theta_samples</i>	The number of angular samples

Returns

std::vector<std::complex<double>> The grid of points in the z-plane

3.1.2.7 is_controllable()

```
bool Analysis::is_controllable (
    const Discrete_StateSpace_System & System ) [inline]
```

Checks if the system is controllable.

A system is controllable if and only if its controllability matrix has full rank:

$$\text{rank}([B \ AB \ A^2B \ \dots \ A^{n-1}B]) = n$$

where n is the number of states.

Controllability implies that there exists an input sequence that can transfer the system from any initial state to any final state in finite time.

Parameters

<i>System</i>	The discrete state space system to analyze
---------------	--

Returns

bool True if rank(C) = n, false otherwise

3.1.2.8 is_detectable_cont()

```
bool Analysis::is_detectable_cont (
    const Discrete_StateSpace_System & System ) [inline]
```

Checks if the continuous-time system is detectable.

A system is detectable if all unobservable modes are stable. Uses the Popov-Belevitch-Hautus (PBH) test for each unstable eigenvalue.

Parameters

<i>System</i>	The discrete state space system to analyze
---------------	--

Returns

bool True if the system is detectable, false otherwise

3.1.2.9 is_observable()

```
bool Analysis::is_observable (
    const Discrete_StateSpace_System & System ) [inline]
```

Checks if the system is observable.

A system is observable if and only if its observability matrix has full rank:

$$\text{rank} \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix} = n$$

Observability implies that the initial state can be determined from knowledge of the input and output over a finite time interval.

Parameters

<i>System</i>	The discrete state space system to analyze
---------------	--

Returns

bool True if rank(O) = n, false otherwise

3.1.2.10 is_stabalizable_cont()

```
bool Analysis::is_stabalizable_cont (
    const Discrete_StateSpace_System & System ) [inline]
```

Checks if the continuous-time system is stabilizable.

A system is stabilizable if all uncontrollable modes are stable. This is checked using the Popov-Belevitch-Hautus (PBH) test:

$$\text{rank} \begin{bmatrix} sI - A & B \end{bmatrix} = n$$

for all eigenvalues s with $\text{Re}(s) \geq 0$, where:

- n is the number of states
- A is the state matrix
- B is the input matrix

If this condition is satisfied for all unstable eigenvalues, then the system is stabilizable.

Parameters

<i>System</i>	The discrete state space system to analyze
---------------	--

Returns

bool True if the system is stabilizable

3.1.2.11 Kalman_Decomp()

```
Discrete_StateSpace_System Analysis::Kalman_Decomp (
    const Discrete_StateSpace_System & System ) [inline]
```

Performs Kalman decomposition for state estimation.

This method computes the observability and controllability decompositions, and then combines them to form a new state space realization.

Parameters

<i>System</i>	The discrete state space system to analyze
---------------	--

Returns

Discrete_StateSpace_System The decomposed state space system

3.1.2.12 Linear_Stability_cont()

```
static bool Analysis::Linear_Stability_cont (
    const Discrete_StateSpace_System & System ) [inline], [static]
```

Checks the stability of a continuous-time linear system.

For a continuous-time system, stability requires all eigenvalues to lie in the left half-plane:

$$\operatorname{Re}(\lambda_i(A)) < 0 \quad \forall i$$

where $\lambda_i(A)$ are the eigenvalues of matrix A .

Stability types:

- $\operatorname{Re}(\lambda) < 0$: Asymptotically stable
- $\operatorname{Re}(\lambda) = 0$: Marginally stable
- $\operatorname{Re}(\lambda) > 0$: Unstable

Parameters

<i>System</i>	The discrete state space system to analyze
---------------	--

Returns

bool True if all eigenvalues have negative real parts

3.1.2.13 Linear_Stability_discrete()

```
bool Analysis::Linear_Stability_discrete (
    const Discrete_StateSpace_System & System ) [inline]
```

Checks the stability of a discrete-time linear system.

For a discrete-time system, stability requires all eigenvalues to lie inside the unit circle:

$$|\lambda_i(A)| < 1 \quad \forall i$$

where $\lambda_i(A)$ are the eigenvalues of matrix A.

Stability types:

- $|| < 1$: Asymptotically stable
- $|| = 1$: Marginally stable
- $|| > 1$: Unstable

Parameters

<i>System</i>	The discrete state space system to analyze
---------------	--

Returns

bool True if all eigenvalues have magnitude less than 1

3.1.2.14 minimality_test_cont()

```
bool Analysis::minimality_test_cont (
    const Discrete_StateSpace_System & System ) [inline]
```

Performs the minimality test for continuous-time systems.

A system is minimal if it is both controllable and observable.

Parameters

<i>System</i>	The discrete state space system to analyze
---------------	--

Returns

bool True if the system is minimal, false otherwise

3.1.2.15 observability_decomposition()

```
std::tuple< Eigen::MatrixXd, Eigen::MatrixXd > Analysis::observability_decomposition (
    const Discrete_StateSpace_System & System ) [inline]
```

Computes the observability decomposition of a discrete-time state space system.

This method computes the observability matrix and performs SVD to find the invariant subspaces of the system.

Parameters

<i>System</i>	The discrete state space system to analyze
---------------	--

Returns

std::tuple<Eigen::MatrixXd, Eigen::MatrixXd> The observable and unobservable subspaces

3.1.2.16 poles()

```
std::vector< std::complex< double > > Analysis::poles (
    const Discrete_StateSpace_System & System ) [inline]
```

Computes the poles of the system (eigenvalues of A matrix)

Parameters

<i>System</i>	The discrete state space system to analyze
---------------	--

Returns

std::vector<std::complex<double>> The eigenvalues of the A matrix

3.1.2.17 zeros()

```
std::vector< std::complex< double > > Analysis::zeros (
    const Discrete_StateSpace_System & System,
    const double & r_min,
    const double & r_max,
    const int & r_samples,
    const int & theta_samples ) [inline]
```

Finds the zeros of the transfer function (poles of the closed-loop system)

Parameters

<i>System</i>	The discrete state space system to analyze
<i>r_min</i>	The minimum radius for the grid
<i>r_max</i>	The maximum radius for the grid
<i>r_samples</i>	The number of radial samples
<i>theta_samples</i>	The number of angular samples

Returns

`std::vector<std::complex<double>>` The zeros of the transfer function

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

- analysis.hpp

3.2 Forms Class Reference

A class for transforming state space systems into various canonical forms.

```
#include <forms.hpp>
```

Public Member Functions

- Discrete_StateSpace_System [Cont_Cannonical_form](#) (const Discrete_StateSpace_System &System)
Transforms a state space system into controllable canonical form.
- Discrete_StateSpace_System [obs_Cannonical_form](#) (const Discrete_StateSpace_System &System)
Transforms a state space system into observable canonical form.
- Discrete_StateSpace_System [Phase_Variable_Form](#) (const Discrete_StateSpace_System &System)
Transforms a state space system into phase variable form.

Data Fields

- Eigen::MatrixXd **T_inv** = T.inverse()
- Discrete_StateSpace_System **new_system** = System
- new_system **A** = T_inv * System.A * T
- new_system **B** = T_inv * System.B
- new_system **C** = System.C * T
- return **new_system**

3.2.1 Detailed Description

A class for transforming state space systems into various canonical forms.

This class provides methods to transform discrete-time state space systems into different canonical forms. Each transformation is achieved through similarity transformations of the form:

- $A_{new} = T^{-1}AT$
- $B_{new} = T^{-1}B$
- $C_{new} = CT$ where T is the transformation matrix specific to each form.

3.2.2 Member Function Documentation

3.2.2.1 Cont_Cannonical_form()

```
Discrete_StateSpace_System Forms::Cont_Cannonical_form (
    const Discrete_StateSpace_System & System ) [inline]
```

Transforms a state space system into controllable canonical form.

The controllable canonical form transforms the system into the form:

$$A = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ -a_0 & -a_1 & -a_2 & \cdots & -a_{n-1} \end{bmatrix}$$

where a_i are the coefficients of the characteristic polynomial:

$$p(s) = s^n + a_{n-1}s^{n-1} + \cdots + a_1s + a_0$$

The transformation uses the controllability matrix:

$$T = [B \quad AB \quad A^2B \quad \cdots \quad A^{n-1}B]$$

Parameters

<i>System</i>	The discrete state space system to transform
---------------	--

Returns

Discrete_StateSpace_System The transformed system in controllable canonical form

3.2.2.2 obs_Cannonical_form()

```
Discrete_StateSpace_System Forms::obs_Cannonical_form (
    const Discrete_StateSpace_System & System ) [inline]
```

Transforms a state space system into observable canonical form.

The observable canonical form transforms the system into the form:

$$A = \begin{pmatrix} -a_{n-1} & -a_{n-2} & \cdots & -a_1 & -a_0 \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 1 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{pmatrix}$$

The transformation uses the observability matrix:

$$T = \begin{bmatrix} C \\ CA \\ CA^2 \\ \vdots \\ CA^{n-1} \end{bmatrix}$$

Parameters

<i>System</i>	The discrete state space system to transform
---------------	--

Returns

Discrete_StateSpace_System The transformed system in observable canonical form

3.2.2.3 Phase_Variable_Form()

```
Discrete_StateSpace_System Forms::Phase_Variable_Form (
    const Discrete_StateSpace_System & System ) [inline]
```

Transforms a state space system into phase variable form.

The phase variable form represents the system in terms of a state vector containing successive derivatives (or differences in discrete-time):

$$x = \begin{bmatrix} y \\ \Delta y \\ \Delta^2 y \\ \vdots \\ \Delta^{n-1} y \end{bmatrix}$$

This form requires the system to be controllable. The transformation matrix T is constructed from the controllability matrix.

Parameters

<i>System</i>	The discrete state space system to transform
---------------	--

Returns

Discrete_StateSpace_System The transformed system in phase variable form

Exceptions

<i>std::runtime_error</i>	if the system is not controllable
---------------------------	-----------------------------------

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

- forms.hpp

3.3 Linear_Solvers Class Reference

Class containing static methods for solving linear systems.

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

Static Public Member Functions

- static [My_Vec SolveLU](#) (const [Matrix](#) &A, const [My_Vec](#) &b)
Solves a linear system using LU decomposition.
- static [My_Vec SolveQR](#) (const [Matrix](#) &A, const [My_Vec](#) &b)
Solves a linear system using QR decomposition.
- static [Matrix Inverse](#) (const [Matrix](#) &A)
Computes the inverse of a matrix using LU decomposition.
- static [My_Vec ForwardSubstitution](#) (const [Matrix](#) &L_1, const [My_Vec](#) &b)
Performs forward substitution to solve $Lx = b$.
- static [My_Vec BackwardSubstitution](#) (const [Matrix](#) &U_1, const [My_Vec](#) &b)
Performs backward substitution to solve $Ux = b$.
- static [My_Vec ApplyPermutation](#) (const std::vector< int > &P, const [My_Vec](#) &V)
Applies a permutation to a vector.
- static double [determinant](#) (const [Matrix](#) &A)
Computes the determinant of a matrix using LU decomposition.
- static [My_Vec solve_linear_system_LU](#) (const [Matrix](#) &A, const [My_Vec](#) &b)
Alias for SolveLU for solving linear systems.
- static [My_Vec solve_linear_system_QR](#) (const [Matrix](#) &A, const [My_Vec](#) &b)
Alias for SolveQR for solving linear systems.

3.3.1 Detailed Description

Class containing static methods for solving linear systems.

Static class providing methods for solving linear systems.

using various methods including LU and QR decomposition

Implements various numerical methods for solving linear systems including:

- LU decomposition based solver
- QR decomposition based solver
- [Matrix](#) inversion
- Forward and backward substitution

3.3.2 Member Function Documentation

3.3.2.1 ApplyPermutation()

```
static My\_Vec Linear_Solvers::ApplyPermutation (
    const std::vector< int > & P,
    const My\_Vec & V ) [inline], [static]
```

Applies a permutation to a vector.

Parameters

P	Permutation vector
V	Vector to permute

Returns

Permuted vector

Exceptions

<code>std::invalid_argument</code>	if permutation size doesn't match vector size
------------------------------------	---

3.3.2.2 BackwardSubstitution()

```
static My_Vec Linear_Solvers::BackwardSubstitution (
    const Matrix & U_1,
    const My_Vec & b ) [inline], [static]
```

Performs backward substitution to solve $Ux = b$.

Parameters

$U_{\leftarrow 1}$	Upper triangular matrix
b	Right-hand side vector

Returns

Solution vector x

3.3.2.3 determinant()

```
static double Linear_Solvers::determinant (
    const Matrix & A ) [inline], [static]
```

Computes the determinant of a matrix using LU decomposition.

Parameters

A	Input matrix
-----	--------------

Returns

Determinant value

3.3.2.4 ForwardSubstitution()

```
static My_Vec Linear_Solvers::ForwardSubstitution (
    const Matrix & L_1,
    const My_Vec & b ) [inline], [static]
```

Performs forward substitution to solve $Lx = b$.

Parameters

$L_{\leftarrow 1}$	Lower triangular matrix
b	Right-hand side vector

Returns

Solution vector x

3.3.2.5 Inverse()

```
static Matrix Linear_Solvers::Inverse (
    const Matrix & A ) [inline], [static]
```

Computes the inverse of a matrix using LU decomposition.

Parameters

A	The matrix to invert
-----	----------------------

Returns

The inverse of matrix A

Exceptions

<code>std::runtime_error</code>	if matrix is singular
---------------------------------	-----------------------

3.3.2.6 solve_linear_system_LU()

```
static My_Vec Linear_Solvers::solve_linear_system_LU (
    const Matrix & A,
    const My_Vec & b ) [inline], [static]
```

Alias for SolveLU for solving linear systems.

Parameters

A	The coefficient matrix
b	The right-hand side vector

Returns

Solution vector x where $Ax = b$

3.3.2.7 solve_linear_system_QR()

```
static My_Vec Linear_Solvers::solve_linear_system_QR (  
    const Matrix & A,  
    const My_Vec & b ) [inline], [static]
```

Alias for SolveQR for solving linear systems.

Parameters

A	The coefficient matrix
b	The right-hand side vector

Returns

Solution vector x where $Ax = b$

3.3.2.8 SolveLU()

```
static My_Vec Linear_Solvers::SolveLU (  
    const Matrix & A,  
    const My_Vec & b ) [inline], [static]
```

Solves a linear system using LU decomposition.

Parameters

A	The coefficient matrix
b	The right-hand side vector

Returns

Solution vector x where $Ax = b$

3.3.2.9 SolveQR()

```
static My_Vec Linear_Solvers::SolveQR (  
    const Matrix & A,  
    const My_Vec & b ) [inline], [static]
```

Solves a linear system using QR decomposition.

Parameters

A	The coefficient matrix
b	The right-hand side vector

Returns

Solution vector x where $Ax = b$

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

- [matrix_math.hpp](#)
- [linear_solvers.hpp](#)

3.4 LUResult Struct Reference

Result of LU decomposition of a matrix.

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

Data Fields

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

3.4.1 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](#)

3.5 LUResult_to_pass Struct Reference

Collaboration diagram for LUResult_to_pass:

Data Fields

- [Matrix L](#)
- [Matrix U](#)
- `std::vector< int > P`

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

- [matrix_math.hpp](#)

3.6 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.

Data Fields

- 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.

3.6.1 Detailed Description

[Matrix](#) class implementation for mathematical operations.

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

3.6.2 Constructor & Destructor Documentation

3.6.2.1 Matrix() [1/2]

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

Constructor for [Matrix](#).

Parameters

<i>rs</i>	Number of rows (defaults to 1)
<i>cs</i>	Number of columns (defaults to 1)

3.6.2.2 Matrix() [2/2]

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

Copy constructor for [Matrix](#).

Parameters

<i>other</i>	The matrix to copy
--------------	--------------------

3.6.3 Member Function Documentation

3.6.3.1 Embed()

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

Embeds a Householder matrix into a larger identity matrix.

Parameters

<i>Householder</i>	The Householder matrix to embed
<i>A</i>	The original matrix

Returns

The embedded matrix

3.6.3.2 eye()

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

Static method to create an identity matrix of size a.

Parameters

<i>a</i>	Size of the matrix (number of rows and columns)
----------	---

Returns

Identity matrix of size a

3.6.3.3 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

<code>std::invalid_argument</code>	if the matrix is not square
<code>std::runtime_error</code>	if the matrix is singular or nearly singular

3.6.3.4 multiply()

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

Multiplies the matrix with a vector.

Parameters

<code>x</code>	The vector to multiply with
----------------	-----------------------------

Returns

Resulting vector after multiplication

Exceptions

<code>std::invalid_argument</code>	if dimensions are not compatible
------------------------------------	----------------------------------

3.6.3.5 Ones()

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

Static method to create a matrix of ones.

Parameters

<code>a</code>	Number of rows
<code>b</code>	Number of columns

Returns

[Matrix](#) of size a x b with all elements set to 1

3.6.3.6 operator*()

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

Matrix multiplication operator.

Parameters

<i>other</i>	The matrix to multiply with
--------------	-----------------------------

Returns

Resulting matrix after multiplication

Exceptions

<i>std::invalid_argument</i>	if dimensions are not compatible
------------------------------	----------------------------------

3.6.3.7 operator+()

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

Matrix addition operator.

Parameters

<i>other</i>	The matrix to add
--------------	-------------------

Returns

Resulting matrix after addition

Exceptions

<i>std::invalid_argument</i>	if matrices have different dimensions
------------------------------	---------------------------------------

3.6.3.8 operator-()

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

Matrix subtraction operator.

Parameters

<i>other</i>	The matrix to subtract
--------------	------------------------

Returns

Resulting matrix after subtraction

Exceptions

<code>std::invalid_argument</code>	if matrices have different dimensions
------------------------------------	---------------------------------------

3.6.3.9 operator=()

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

Assignment operator for [Matrix](#).

Parameters

<i>other</i>	The matrix to assign
--------------	----------------------

Returns

Reference to this matrix

3.6.3.10 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

<i>u</i>	First vector
<i>v</i>	Second vector

Returns

[Matrix](#) resulting from the outer product

3.6.3.11 QR_fact()

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

Computes the QR Decomposition of the matrix.

Returns

[QRresult](#) structure containing Q, R matrices

3.6.3.12 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

3.6.3.13 Transpose()

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

Transposes the matrix.

Returns

New matrix that is the transpose of this matrix

3.6.3.14 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

3.6.3.15 Zeros()

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

Static method to create a matrix of zeros.

Parameters

<i>a</i>	Number of rows
<i>b</i>	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](#)

3.7 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.

Data Fields

- int **length**
Length of the vector.
- `std::vector< double >` **myvector**
Vector data storage.

3.7.1 Detailed Description

Vector class for mathematical operations.

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

3.7.2 Constructor & Destructor Documentation

3.7.2.1 `My_Vec()` [1/2]

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

Construct a new vector.

Parameters

<i>l</i>	Length of the vector (defaults to 1)
----------	--------------------------------------

3.7.2.2 `My_Vec()` [2/2]

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

Copy constructor for `My_Vec`.

Parameters

<i>other</i>	The vector to copy
--------------	--------------------

3.7.3 Member Function Documentation

3.7.3.1 dot()

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

Computes the dot product with another vector.

Parameters

<i>other</i>	The other vector
--------------	------------------

Returns

Dot product result

Exceptions

<i>std::invalid_argument</i>	if vectors have different lengths
------------------------------	-----------------------------------

3.7.3.2 Norm()

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

Computes the Euclidean norm (magnitude) of the vector.

Returns

Norm of the vector

3.7.3.3 ones()

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

Creates a vector of ones.

Parameters

<i>a</i>	Length of the vector
----------	----------------------

Returns

Vector with all elements set to 1

3.7.3.4 operator+()

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

Add two vectors.

Parameters

<i>other</i>	Vector to add to this one
--------------	---------------------------

Returns

New vector containing the sum

Exceptions

<i>std::invalid_argument</i>	if vectors have different lengths
------------------------------	-----------------------------------

3.7.3.5 operator-()

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

Vector subtraction operator.

Parameters

<i>other</i>	The vector to subtract
--------------	------------------------

Returns

Result of vector subtraction

Exceptions

<i>std::invalid_argument</i>	if vectors have different lengths
------------------------------	-----------------------------------

3.7.3.6 operator=()

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

Assignment operator for [My_Vec](#).

Parameters

<i>other</i>	The vector to assign
--------------	----------------------

Returns

Reference to this vector

3.7.3.7 Scalar_Mul()

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

Scalar multiplication of the vector.

Parameters

<i>k</i>	Scalar value
----------	--------------

Returns

New vector resulting from scalar multiplication

3.7.3.8 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>i</i>	Position of the 1 in the unit vector
<i>L</i>	Total length of the vector

Returns

Unit vector with 1 at position i

3.7.3.9 Zeros()

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

Creates a zero vector of length i.

Parameters

<i>i</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](#)

3.8 QR_result_to_pass Struct Reference

Collaboration diagram for QR_result_to_pass:

Data Fields

- [Matrix Q](#)
- [Matrix R](#)

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

- [matrix_math.hpp](#)

3.9 QRresult Struct Reference

Result of QR decomposition of a matrix.

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

Data Fields

- `std::vector< std::vector< double > > Q`
Orthogonal matrix.
- `std::vector< std::vector< double > > R`
Upper triangular matrix.

3.9.1 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 analysis.hpp

```
00001 #ifndef ANALYSIS_HPP
00002 #define ANALYSIS_HPP
00003
00004 #include <iostream>
00005 #include <cmath>
00006 #include <complex>
00007 #include <Eigen/Dense>
00008 #include "discrete_state_space.hpp"
00009 #include <Eigen/SVD>
00010 #include <eigen3/unsupported/Eigen/KroneckerProduct>
00011
00026 class Analysis {
00027 public:
00048     static Eigen::MatrixXd compute_controllability_matrix(const Discrete_StateSpace_System& System)
00049     {
00050         int n = System.A.rows();
00051         int m = System.B.cols();
00052         Eigen::MatrixXd controllability_mat(n, n * m);
00053
00054         for (int i = 0; i < n; ++i) {
00055             controllability_mat.block(0, i * m, n, m) = System.A.pow(i) * System.B;
00056         }
00057
00058         return controllability_mat;
00059     }
00060
00077 bool is_controllable(const Discrete_StateSpace_System& System)
00078 {
00079     Eigen::MatrixXd controllability_mat = compute_controllability_matrix(System);
00080     Eigen::FullPivLU<Eigen::MatrixXd> lu(controllability_mat);
00081     return lu.rank() == System.n_states;
00082 }
00083
00105 static Eigen::MatrixXd compute_observability_matrix(const Discrete_StateSpace_System& System)
00106 {
00107     int n = System.A.rows();
00108     int p = System.C.rows();
00109     Eigen::MatrixXd observability_mat(p * n, System.A.cols());
00110
00111     for (int i = 0; i < n; ++i) {
00112         observability_mat.block(i * p, 0, p, System.A.cols()) = System.C * System.A.pow(i);
00113     }
00114
00115     return observability_mat;
00116 }
00117
00138 bool is_observable(const Discrete_StateSpace_System& System)
00139 {
00140     Eigen::MatrixXd observability_mat = compute_observability_matrix(System);
00141     Eigen::FullPivLU<Eigen::MatrixXd> lu(observability_mat);
00142     return lu.rank() == System.n_states;
00143 }
00144
00164 bool Linear_Stability_discrete(const Discrete_StateSpace_System& System)
00165 {
00166     Eigen::VectorXcd eigenvals=System.A.eigenvalues();
00167     for(int i=0;i<eigenvals.size();i++){
00168
```

```

00169
00170         if(std::abs(eigenvals[i])>=1.0){
00171             return false;
00172         }
00173     }
00174     return true;
00175
00195     static bool Linear_Stability_cont(const Discrete_StateSpace_System& System)
00196     {
00197         Eigen::VectorXcd eigenvals = System.A.eigenvalues();
00198         for (int i = 0; i < eigenvals.size(); ++i) {
00199             if (eigenvals[i].real() >= 0.0) {
00200                 return false; // Unstable or marginally stable
00201             }
00202         }
00203         return true;
00204     }
00205
00206
00207
00228     bool is_stabalizable_cont(const Discrete_StateSpace_System &System){
00229         Eigen::VectorXcd eigs=System.A.eigenvalues();
00230         int f=eigs.size();
00231         std::vector<bool> unstable_flag;
00232         int a=System.A.rows();
00233         int b=System.A.cols();
00234         for(int i=0;i<eigs.size();i++){
00235             if(eigs[i].real()>0){
00236                 Eigen::MatrixXcd eye(a,b);
00237                 eye=Eigen::MatrixXcd::Identity(a,b);
00238                 Eigen::MatrixXcd PBH_part_1=eigs[i]*eye-System.A;
00239                 int n=System.B.rows();
00240                 int m=System.B.cols();
00241                 Eigen::MatrixXcd PBH(n,n+m);
00242                 PBH.block(0, 0, n, n) = PBH_part_1;
00243                 PBH.block(0, n, n, m) = System.B.cast<std::complex<double>>();
00244                 Eigen::FullPivLU<Eigen::MatrixXcd> lu(PBH);
00245                 if(lu.rank()<n){
00246                     return false;
00247                 }
00248             }
00249         }
00250     }
00251
00252     }
00253
00254     return true;
00255
00256 }
00257
00258
00268     bool is_detectable_cont(const Discrete_StateSpace_System &System){
00269         Eigen::VectorXcd eigs=System.A.eigenvalues();
00270         int f=eigs.size();
00271         std::vector<bool> unstable_flag;
00272         int a=System.A.rows();
00273         int b=System.A.cols();
00274         for(int i=0;i<eigs.size();i++){
00275             if(eigs[i].real()>0){
00276                 Eigen::MatrixXcd eye = Eigen::MatrixXcd::Identity(a, b);
00277                 Eigen::MatrixXcd PBH_part_1=eigs[i]*eye-System.A;
00278                 int n=System.A.rows();
00279                 int p=System.C.rows();
00280                 Eigen::MatrixXcd PBH(n + p, n);
00281                 PBH.block(0, 0, n, n) = PBH_part_1;
00282                 PBH.block(n, 0, p, n) = System.C.cast<std::complex<double>>();
00283                 Eigen::FullPivLU<Eigen::MatrixXcd> lu(PBH);
00284                 if(lu.rank()<n){
00285                     return false;
00286                 }
00287             }
00288         }
00289     }
00290
00291     }
00292
00293     return true;
00294
00295 }
00296
00297
00306     bool minimality_test_cont(const Discrete_StateSpace_System &System){
00307
00308         if (is_controllable(System) && is_observable(System)) {
00309
00310             return true;
00311         }

```

```

00312         else{
00313             return false;
00314         }
00315     }
00316
00326     std::tuple<Eigen::MatrixXd, Eigen::MatrixXd> controllability_decomposition(const
Discrete_StateSpace_System& System){
00327         Eigen::MatrixXd cont_mat=compute_controllability_matrix(System);
00328         int n=cont_mat.rows();
00329         Eigen::ColPivHouseholderQR <Eigen::MatrixXd> QR(cont_mat);
00330         Eigen::MatrixXd Q=QR.householderQ();
00331         int r=QR.rank();
00332         Eigen::MatrixXd T(n, n);
00333         T <= Q.leftCols(r), Q.rightCols(n - r);
00334
00335         Eigen::MatrixXd Aprime=(T.inverse())*System.A*T;
00336         Eigen::MatrixXd Bprime=(T.inverse())*System.B;
00337         Eigen::MatrixXd Cprime=System.C*T;
00338
00339         Eigen::MatrixXd A_cc=Aprime.topLeftCorner(r,r);
00340         Eigen::MatrixXd A_cu=Aprime.topRightCorner(r,n-r);
00341         std::tuple<Eigen::MatrixXd, Eigen::MatrixXd> Ans={A_cc,A_cu};
00342
00343         return Ans;
00344     }
00345
00346
00347
00357     std::tuple<Eigen::MatrixXd, Eigen::MatrixXd> observability_decomposition(const
Discrete_StateSpace_System& System){
00358
00359         Eigen::MatrixXd obs_mat=compute_observability_matrix(System);
00360         int n=System.A.rows();
00361         Eigen::JacobiSVD<Eigen::MatrixXd> SVD(obs_mat);
00362
00363         Eigen::MatrixXd V=SVD.matrixV();
00364         Eigen::MatrixXd VT=V.transpose();
00365         double tol = 1e-9;
00366         int r = (SVD.singularValues().array() > tol).count();
00367
00368         Eigen::MatrixXd T(n, n);
00369         T <= V.leftCols(r), V.rightCols(n - r);
00370
00371         Eigen::MatrixXd Aprime=(T.inverse())*System.A*T;
00372         Eigen::MatrixXd Bprime=(T.inverse())*System.B;
00373         Eigen::MatrixXd Cprime=System.C*T;
00374
00375         Eigen::MatrixXd A_oo=Aprime.topLeftCorner(r,r);
00376         Eigen::MatrixXd A_ou=Aprime.topRightCorner(r,n-r);
00377         std::tuple<Eigen::MatrixXd, Eigen::MatrixXd> Ans={A_oo,A_ou};
00378
00379         return Ans;
00380
00381
00382     }
00383
00384
00395     Eigen::MatrixXd compute_controllability_gramian(const Discrete_StateSpace_System& System) {
00396
00397         Eigen::MatrixXd Q = System.B * System.B.transpose();
00398         int n = System.A.rows();
00399         Eigen::MatrixXd I = Eigen::MatrixXd::Identity(n, n);
00400         Eigen::VectorXd vecQ = Eigen::Map<const Eigen::VectorXd>(Q.data(), Q.size());
00401
00402         Eigen::MatrixXd kron1 = Eigen::kroneckerProduct(System.A, I);
00403         Eigen::MatrixXd kron2 = Eigen::kroneckerProduct(I, System.A);
00404
00405         Eigen::VectorXd w = (kron1 + kron2).fullPivLu().solve(-vecQ);
00406
00407         Eigen::MatrixXd W = Eigen::Map<Eigen::MatrixXd>(w.data(), n, n);
00408
00409         return W;
00410 }
00411
00412
00423     Eigen::MatrixXd compute_observability_gramian(const Discrete_StateSpace_System& System){
00424
00425         Eigen::MatrixXd Q = (System.C) * System.C.transpose();
00426         int n = System.A.rows();
00427         Eigen::MatrixXd I = Eigen::MatrixXd::Identity(n, n);
00428         Eigen::VectorXd vecQ = Eigen::Map<const Eigen::VectorXd>(Q.data(), Q.size());
00429
00430         Eigen::MatrixXd kron1 = Eigen::kroneckerProduct(System.A, I);
00431         Eigen::MatrixXd kron2 = Eigen::kroneckerProduct(I, System.A);
00432
00433         Eigen::VectorXd w = (kron1 + kron2).fullPivLu().solve(-vecQ);
00434

```

```

00435     Eigen::MatrixXd W = Eigen::Map<Eigen::MatrixXd>(w.data(), n, n);
00436
00437     return W;
00438 }
00439
00446 std::vector<std::complex<double>> poles(const Discrete_StateSpace_System& System){
00447
00448     Eigen::EigenSolver<Eigen::MatrixXd> eigen_solver(System.A);
00449     Eigen::VectorXcd eigvals = eigen_solver.eigenvalues();
00450     std::vector<std::complex<double>> eigs(eigvals.data(), eigvals.data() + eigvals.size());
00451
00452     return eigs;
00453 }
00454
00455
00465 std::vector<std::complex<double>> generate_z_grid(
00466 double r_min, double r_max,
00467 int r_samples,
00468 int theta_samples)
00469 {
00470     std::vector<std::complex<double>> z_grid;
00471     z_grid.reserve(r_samples * theta_samples);
00472
00473     for (int i = 0; i < r_samples; ++i) {
00474         double r = r_min + i * (r_max - r_min) / (r_samples - 1);
00475         for (int j = 0; j < theta_samples; ++j) {
00476             double theta = 2.0 * M_PI * j / theta_samples;
00477             std::complex<double> z = std::polar(r, theta);
00478             z_grid.push_back(z);
00479         }
00480     }
00481     return z_grid;
00482 }
00483
00494 std::vector<std::complex<double>> zeros(
00495     const Discrete_StateSpace_System& System,
00496     const double& r_min, const double& r_max,
00497     const int& r_samples,
00498     const int& theta_samples){
00499
00500
00501     int n = System.A.rows();
00502     int p = System.C.rows();
00503     int m = System.B.cols();
00504     Eigen::MatrixXcd R(n + p, n + m);
00505
00506     std::vector<std::complex<double>> zgrid=generate_z_grid(r_min,r_max,r_samples,theta_samples);
00507     std::vector<std::complex<double>> zeros;
00508     Eigen::MatrixXcd zI_minus_A;
00509
00510     Eigen::MatrixXcd eye=Eigen::MatrixXd::Identity(n, n);
00511     int r;
00512     double tol = 1e-9;
00513     for(int i=0;i<zgrid.size();i++){
00514         zI_minus_A=(zgrid[i]*eye)-(System.A.cast<std::complex<double>>());
00515         R.block(0, 0, n, n) = zI_minus_A;
00516         R.block(0, n, n, m) = -System.B.cast<std::complex<double>>();
00517         R.block(n, 0, p, n) = System.C.cast<std::complex<double>>();
00518         R.block(n, n, p, m) = System.D.cast<std::complex<double>>();
00519
00520         Eigen::JacobiSVD<Eigen::MatrixXcd>SVD(R);
00521         r = (SVD.singularValues().array() > tol).count();
00522         if(r<n+p){
00523
00524             zeros.push_back(zgrid[i]);
00525         }
00526     }
00527
00528 }
00529
00530     return zeros;
00531 }
00532
00542 Discrete_StateSpace_System Kalman_Decomp( const Discrete_StateSpace_System& System){
00543
00544     std::tuple<Eigen::MatrixXd, Eigen::MatrixXd> Cont,Obs;
00545
00546     Obs=observability_decomposition(System);
00547     Cont=controllability_decomposition(System);
00548
00549     Eigen::MatrixXd Obs_subspace=std::get<0>(Obs);
00550     Eigen::MatrixXd Cont_subspace=std::get<0>(Cont);
00551     Eigen::MatrixXcd Diff(Obs_subspace.rows(), Obs_subspace.cols() + Cont_subspace.cols());
00552     Diff « Obs_subspace, -Cont_subspace;
00553     Eigen::FullPivLU<Eigen::MatrixXcd> lul(Diff);
00554     Eigen::MatrixXcd Null_S = lul.kernel();
00555

```



```

00556
00557     int k=Obs_subspace.cols();
00558
00559     Eigen::MatrixXd Null_S_S=Null_S.topRows(k);
00560     Eigen::MatrixXd Basis_1=Null_S_S*Obs_subspace;
00561
00562
00563     Eigen::MatrixXd Obs_ortho=(Obs_subspace.transpose()).fullPivLu().kernel();
00564     Eigen::MatrixXd Cont_ortho=(Cont_subspace.transpose()).fullPivLu().kernel();
00565
00566     Eigen::MatrixXd Diff2(Obs_ortho.rows(), Obs_ortho.cols() + Cont_subspace.cols());
00567     Diff2 << Obs_ortho, -Cont_ortho;
00568     Eigen::MatrixXd Null_S2=Diff2.fullPivLu().kernel();
00569
00570     int l=Obs_ortho.cols();
00571
00572
00573     Eigen::MatrixXd Null_S_S2=Null_S2.topRows(l);
00574     Eigen::MatrixXd Basis_2=Null_S_S2*Obs_ortho;
00575
00576
00577     Eigen::MatrixXd Diff3(Obs_subspace.rows(), Obs_subspace.cols() + Cont_ortho.cols());
00578     Diff3 << Obs_subspace, -Cont_ortho;
00579     Eigen::MatrixXd Null_S3=Diff3.fullPivLu().kernel();
00580
00581
00582     Eigen::MatrixXd Null_S_S3=Null_S3.topRows(k);
00583     Eigen::MatrixXd Basis_3=Null_S_S3*Obs_subspace;
00584
00585
00586
00587     Eigen::MatrixXd Diff4(Obs_ortho.rows(), Obs_subspace.cols() + Cont_ortho.cols());
00588     Diff4 << Obs_ortho, -Cont_ortho;
00589     Eigen::MatrixXd Null_S4=Diff3.fullPivLu().kernel();
00590
00591
00592     Eigen::MatrixXd Null_S_S4=Null_S4.topRows(l);
00593     Eigen::MatrixXd Basis_4=Null_S_S4*Obs_ortho;
00594
00595     int n=Basis_1.rows();
00596     Eigen::MatrixXd T(n, n);
00597     T << Basis_1, Basis_2, Basis_3, Basis_4;
00598
00599     if (T.fullPivLu().isInvertible()) {
00600     }
00601     else {
00602         Eigen::MatrixXd Q = QR.householderQ();
00603         T = Q.leftCols(n); // Truncate if necessary
00604
00605     }
00606
00607     Eigen::MatrixXd T_inv = T.inverse();
00608
00609
00610     Eigen::Matrix A_new=T_inv*System.A*T;
00611     Eigen::Matrix B_new=T_inv*System.B;
00612     Eigen::Matrix C_new=System.C*T;
00613     Eigen::Matrix D_new=System.D;
00614
00615
00616     Discrete_StateSpace_System Decom;
00617     Decom.A=A_new;
00618     Decom.B=B_new;
00619     Decom.C=C_new;
00620     Decom.D=D_new;
00621
00622     return Decom;
00623
00624 }
00625
00626
00627 private:
00628 };
00629
00630 #endif

```

4.2 forms.hpp

```

00001 #ifndef FORMS_HPP
00002 #define FORMS_HPP
00003
00004 #include <iostream>
00005 #include <cmath>

```

```

00006 #include <complex>
00007 #include <Eigen/Dense>
00008 #include "discrete_state_space.hpp"
00009 #include "analysis.hpp"
00010 #include <Eigen/SVD>
00011 #include <eigen3/unsupported/Eigen/KroneckerProduct>
00012
00013
00025 class Forms{
00026
00027     public:
00055     Discrete_StateSpace_System Cont_Cannonical_form(const Discrete_StateSpace_System& System)
00056     {
00057
00058
00059         int n = System.A.rows();
00060         Eigen::MatrixXd B0 = Eigen::MatrixXd::Identity(n, n);
00061         Eigen::MatrixXd Bk;
00062         std::vector<double> coeffs;
00063         double ak_1 = 0;
00064         double trk = 0;
00065
00066         for (int k = 1; k <= n; k++) {
00067             if (k == 1) {
00068                 // First iteration: trace(A), a0 coefficient, initialize Bk
00069                 trk = System.A.diagonal().sum(); // trace(A)
00070                 ak_1 = -trk / k;
00071                 coeffs.push_back(ak_1);
00072                 Bk = B0;
00073             } else {
00074                 // Subsequent iterations: compute Bk, trace, and coefficient
00075                 Eigen::MatrixXd Bk_1 = System.A * Bk + ak_1 * B0;
00076                 trk = (System.A * Bk_1).diagonal().sum();
00077                 ak_1 = -trk / k;
00078                 coeffs.push_back(ak_1);
00079                 Bk = Bk_1;
00080             }
00081         };
00082
00083
00084         Eigen::MatrixXd Accf = Eigen::MatrixXd::Zero(n, n);
00085         for(int i=0;i<n-1;i++){
00086
00087             Accf(i,i+1)=1;
00088
00089         }
00090         for(int j=0;j<n;j++){
00091
00092             Accf(n-1,j)=-coeffs[j];
00093
00094         }
00095
00096         //Compute Controlability Matrix
00097         Analysis A;
00098         Eigen::MatrixXd T = A.compute_controllability_matrix(System);
00099         Eigen::MatrixXd T_inv=T.inverse();
00100
00101         // Ensure the constructor matches the expected signature, e.g. (A, B, C, D)
00102         Discrete_StateSpace_System new_system = System;
00103         new_system.A = Accf;
00104         new_system.B = T_inv * System.B;
00105         new_system.C = System.C * T;
00106         new_system.D = System.D;
00107
00108
00109
00110
00111         return new_system;
00112     }
00113
00114
00137     Discrete_StateSpace_System obs_Cannonical_form(const Discrete_StateSpace_System& System){
00138
00139         int n = System.A.rows();
00140         Eigen::MatrixXd B0 = Eigen::MatrixXd::Identity(n, n);
00141         Eigen::MatrixXd Bk;
00142         std::vector<double> coeffs;
00143         double ak_1 = 0;
00144         double trk = 0;
00145
00146         for (int k = 1; k <= n; k++) {
00147             if (k == 1) {
00148                 // First iteration: trace(A), a0 coefficient, initialize Bk
00149                 trk = System.A.diagonal().sum(); // trace(A)
00150                 ak_1 = -trk / k;
00151                 coeffs.push_back(ak_1);
00152                 Bk = B0;

```

```

00153         } else {
00154             // Subsequent iterations: compute Bk, trace, and coefficient
00155             Eigen::MatrixX<double> Bk_1 = System.A * Bk + ak_1 * B0;
00156             trk = (System.A * Bk_1).diagonal().sum();
00157             ak_1 = -trk / k;
00158             coeffs.push_back(ak_1);
00159             Bk = Bk_1;
00160         }
00161     };
00162
00163     Eigen::MatrixX<double> Aocf = Eigen::MatrixX<double>::Zero(n, n);
00164     for(int i=0; i<n-1; i++){
00165         Aocf(i, i+1)=1;
00166     }
00167     for(int j=0; j<n; j++){
00168         Aocf(j, 0) = -coeffs[n - j - 1];
00169     }
00170
00171     //Compute Controlability Matrix
00172     Analysis A;
00173     Eigen::MatrixX<double> T = Analysis::compute_observability_matrix(System);
00174     Eigen::MatrixX<double> T_inv=T.inverse();
00175
00176     // Ensure the constructor matches the expected signature, e.g. (A, B, C, D)
00177     Discrete_StateSpace_System new_system = System;
00178     new_system.A = Aocf;
00179     new_system.B = T_inv * System.B;
00180     new_system.C = System.C * T;
00181     new_system.D = System.D;
00182
00183     return new_system;
00184 }
00185
00186 Discrete_StateSpace_System Phase_Variable_Form(const Discrete_StateSpace_System& System) {
00187     int n = System.A.rows();
00188
00189     // Build controllability matrix and check invertibility
00190     Eigen::MatrixX<double> T = Analysis::compute_controllability_matrix(System);
00191     if (T.determinant() == 0)
00192         throw std::runtime_error("System is not controllable. Cannot convert to phase variable form.");
00193     Eigen::MatrixX<double> T_inv = T.inverse();
00194
00195     // Transform system to phase variable form
00196     Discrete_StateSpace_System new_system = System;
00197     new_system.A = T_inv * System.A * T;
00198     new_system.B = T_inv * System.B;
00199     new_system.C = System.C * T;
00200     // D remains unchanged
00201
00202     return new_system;
00203 }
00204
00205 Discrete_StateSpace_System Schur_Form(const Discrete_StateSpace_System& System) {
00206     Eigen::RealSchur<Eigen::MatrixX<double>> schur(System.A);
00207     Eigen::MatrixX<double> Q = schur.matrixU();
00208     Eigen::MatrixX<double> T = schur.matrixT();
00209
00210     Eigen::MatrixX<double> Q_inv=Q.transpose();
00211
00212     Discrete_StateSpace_System new_system = System;
00213     new_system.A = T;
00214     new_system.B = Q_inv * System.B;
00215     new_system.C = System.C * Q;
00216
00217     return new_system;
00218 }
00219
00220 Discrete_StateSpace_System Diagonalize(const Discrete_StateSpace_System& System) {
00221     Eigen::EigenSolver<Eigen::MatrixX<double>> es(System.A);
00222     Eigen::MatrixX<double> eigenvectors = es.eigenvectors();
00223     Eigen::VectorX<double> eigenvalues = es.eigenvalues();
00224
00225     // Check if matrix is diagonalizable (eigenvectors matrix invertible)
00226     if (eigenvectors.determinant() == 0) {
00227         throw std::runtime_error("Matrix is not diagonalizable");
00228     }
00229 }

```

```

00298         }
00299
00300         Eigen::MatrixXcd P = eigenvectors;
00301         Eigen::MatrixXcd P_inv = P.inverse();
00302
00303         Eigen::MatrixXcd D = eigenvalues.asDiagonal();
00304
00305         Discrete_StateSpace_System new_system;
00306
00307         new_system.A = (P_inv * System.A.cast<std::complex<double>>() * P).real();
00308         new_system.B = (P_inv * System.B.cast<std::complex<double>>() * P).real();
00309         new_system.C = (System.C.cast<std::complex<double>>() * P).real();
00310         new_system.D = System.D;
00311
00312         return new_system;
00313     }
00314
00315
00316
00317     private:
00318
00319
00320
00321
00322 };
00323 #endif

```

4.3 linear_solvers.hpp File Reference

Linear system solvers implementation.

```

#include <iostream>
#include <cmath>
#include "matrix_math.hpp"

```

Include dependency graph for linear_solvers.hpp:

4.4 linear_solvers.hpp

[Go to the documentation of this file.](#)

```

00001
00008 #ifndef LINEAR_SOLVER_HPP
00009 #define LINEAR_SOLVER_HPP
00010
00011 #include <iostream>
00012 #include <cmath>
00013 #include "matrix_math.hpp"
00014
00024 class Linear_Solvers {
00025 public:
00032     static My_Vec SolveLU(const Matrix& A, const My_Vec& b) {
00033
00034         LUResult LU_temp=A.L_U();
00035         LUResult_to_pass LU=conv_LU(LU_temp);
00036         My_Vec pb= ApplyPermutation(LU.P,b);
00037         My_Vec fwd_sub=ForwardSubstitution(LU.L,pb);
00038         My_Vec bck_sub=BackwardSubstitution(LU.U,fwd_sub);
00039
00040         return bck_sub;
00041
00042     }
00049     static My_Vec SolveQR(const Matrix& A, const My_Vec& b) {
00050
00051         QRresult decomp_temp=A.QR_fact();
00052         QR_result_to_pass decomp=conv_QR(decomp_temp);
00053         My_Vec qTb=(decomp.Q.Transpose()).multiply(b);
00054         My_Vec x=BackwardSubstitution(decomp.R,qTb);
00055         return x;
00056     };
00063     static Matrix Inverse(const Matrix& A) {
00064
00065         Matrix I=Matrix::eye(A.rows);
00066         LUResult decomp_temp=A.L_U();
00067         LUResult_to_pass decomp=conv_LU(decomp_temp);

```

```

00068     Matrix L=decomp.L;
00069     Matrix U=decomp.U;
00070     My_Vec e_i = My_Vec::ones(A.rows);
00071     Matrix inverse_matrix=Matrix::Zeros(A.rows,A.cols);
00072     for(int j=0;j<I.cols;j++){
00073         e_i.Scalar_Mul(0);
00074         e_i.myvector[j] = 1;
00075         My_Vec Pe_j = ApplyPermutation(decomp.P, e_i);
00076
00077         My_Vec F1=ForwardSubstitution(L,Pe_j);
00078         My_Vec B1=BackwardSubstitution(U,F1);
00079         for (int row = 0; row < A.rows; row++) {
00080             inverse_matrix.MyMAT[row][j] = B1.myvector[row];
00081         }
00082     }
00083
00084     return inverse_matrix;
00085
00086 };
00093 static My_Vec ForwardSubstitution(const Matrix& L_1, const My_Vec& b) {
00094
00095     My_Vec solution_VEC=My_Vec::ones(L_1.rows);
00096     solution_VEC.Scalar_Mul(0);
00097
00098     Matrix L= L_1;
00099     std::vector<double> knowns;
00100     double new_unknown;
00101     for(int i=0;i<L.rows;i++){
00102
00103         double known_sum=0;
00104         for(int j=0;j<i;j++){
00105             known_sum+=L.MyMAT[i][j]*solution_VEC.myvector[j];
00106         }
00107         new_unknown=(b.myvector[i]-known_sum)/L.MyMAT[i][i];
00108
00109
00110         solution_VEC.myvector[i]=new_unknown;
00111
00112     }
00113
00114     return solution_VEC;
00115
00116 };
00117
00118 };
00125 static My_Vec BackwardSubstitution(const Matrix& U_1, const My_Vec& b) {
00126     My_Vec solution_VEC=My_Vec::ones(U_1.rows);
00127     solution_VEC.Scalar_Mul(0);
00128
00129     Matrix U= U_1;
00130     std::vector<double> knowns;
00131     double new_unknown;
00132     for(int i=U.rows-1;i>=0;i--){
00133
00134         double known_sum=0;
00135         for (int j = i + 1; j < U.cols; j++){
00136             known_sum+=U.MyMAT[i][j]*solution_VEC.myvector[j];
00137         }
00138         new_unknown=(b.myvector[i]-known_sum)/U.MyMAT[i][i];
00139
00140
00141         solution_VEC.myvector[i]=new_unknown;
00142
00143     }
00144
00145     return solution_VEC;
00146
00147 };
00148
00149 };
00157 static My_Vec ApplyPermutation(const std::vector<int>& P,const My_Vec& V){
00158     if (P.size() != V.myvector.size()) {
00159         throw std::invalid_argument("Permutation size must match vector size");
00160     }
00161     My_Vec result;
00162
00163     My_Vec result = My_Vec::Zeros(V.myvector.size());
00164
00165     for (size_t i = 0; i < P.size(); i++) {
00166         result.myvector[i] = V.myvector[P[i]];
00167     }
00168
00169     return result;
00170 }
00171
00177 static double determinant(const Matrix& A) {
00178     LUResult lu_temp = A.L_U();

```

```

00179     LUResult_to_pass lu=conv_LU(lu_temp);
00180     double det = 1.0;
00181     for(int i = 0; i < A.rows; i++) {
00182         det *= lu.U.MyMAT[i][i];
00183     }
00184     return det;
00185 }
00186
00187
00194     static My_Vec solve_linear_system_LU(const Matrix& A, const My_Vec& b) {
00195         return SolveLU(A, b);
00196     }
00197
00204     static My_Vec solve_linear_system_QR(const Matrix& A, const My_Vec& b) {
00205         return SolveQR(A, b);
00206     }
00207
00208     };
00209 #endif

```

4.5 matrix_math.hpp File Reference

Linear algebra and matrix mathematics library.

```

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

```

Include dependency graph for matrix_math.hpp: This graph shows which files directly or indirectly include this file:

Data Structures

- struct [LUResult](#)
Result of LU decomposition of a matrix.
- struct [QRresult](#)
Result of QR decomposition of a matrix.
- class [My_Vec](#)
Vector class for mathematical operations.
- class [Matrix](#)
Matrix class implementation for mathematical operations.
- class [Linear_Solvers](#)
Class containing static methods for solving linear systems.
- struct [LUResult_to_pass](#)
- struct [QR_result_to_pass](#)

Functions

- [LUResult_to_pass conv_LU](#) (const [LUResult](#) &LU)
- [QR_result_to_pass conv_QR](#) (const [QRresult](#) &QR)

4.5.1 Detailed Description

Linear algebra and matrix mathematics library.

This library provides implementation for basic linear algebra operations, including matrix operations, LU decomposition, and QR factorization.

4.6 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
00283     LUResult L_U() const {
00284         // Check for square matrix
00285         if(this->rows != this->cols) {
00286             throw std::invalid_argument("Not square");
00287         }
00288
00289         Matrix L = eye(this->rows);
00290         Matrix U = Zeros(this->rows, this->cols);
00291         std::vector<int> P(this->rows);
00292         for(int i = 0; i < this->rows; i++) {
00293             P[i] = i;
00294         }
00295
00296         Matrix A_work(*this);
00297
00298         for(int j = 0; j < this->cols; j++) {
00299             int pivot_row = j;
00300             double max_val = std::abs(A_work.MyMAT[j][j]);
00301
00302             for(int i = j+1; i < this->rows; i++) {
00303                 if(std::abs(A_work.MyMAT[i][j]) > max_val) {
00304                     pivot_row = i;
00305                     max_val = std::abs(A_work.MyMAT[i][j]);
00306                 }
00307             }
00308
00309             if(pivot_row != j) {
00310                 for(int k = 0; k < this->cols; k++) {
00311                     std::swap(A_work.MyMAT[j][k], A_work.MyMAT[pivot_row][k]);
00312                 }
00313
00314                 std::swap(P[j], P[pivot_row]);
00315
00316                 if(j > 0) {
00317                     for(int k = 0; k < j; k++) {
00318                         std::swap(L.MyMAT[j][k], L.MyMAT[pivot_row][k]);
00319                     }
00320                 }
00321             }
00322
00323             if(std::abs(A_work.MyMAT[j][j]) < 1e-10) {
00324                 throw std::runtime_error("Matrix is singular or nearly singular");
00325             }
00326
00327             for(int i = 0; i <= j; i++) {
00328                 double sum = 0.0;
00329                 for(int k = 0; k < i; k++) {
00330                     sum += L.MyMAT[i][k] * U.MyMAT[k][j];
00331                 }
00332                 U.MyMAT[i][j] = A_work.MyMAT[i][j] - sum;
00333             }
00334
00335             for(int i = j+1; i < this->rows; i++) {
00336                 double sum = 0.0;
00337                 for(int k = 0; k < j; k++) {
00338                     sum += L.MyMAT[i][k] * U.MyMAT[k][j];
00339                 }
00340                 L.MyMAT[i][j] = (A_work.MyMAT[i][j] - sum) / U.MyMAT[j][j];
00341             }
00342         }
00343
00344         LUResult result;
00345         result.L = L.MyMAT;
00346         result.U = U.MyMAT;
00347         result.P = P;
00348         return result;
00349     }
00350
00351     static My_Vec UV(int i, int L) {
00352         My_Vec unit_vec(L);
00353         for (int j=0; j<L; j++) {
00354             unit_vec.myvector[j] = (j == i) ? 1 : 0;
00355         }
00356         return unit_vec;
00357     }
00358
00359     static Matrix Embed(const Matrix& Householder, const Matrix& A) {
00360         Matrix Hp = eye(A.rows);
00361         int i = A.rows - Householder.rows;
00362         for (int a = 0; a < Householder.rows; a++) {
00363             for (int b = 0; b < Householder.cols; b++) {
00364                 Hp.MyMAT[i+a][i+b] = Householder.MyMAT[a][b];
00365             }
00366         }
00367         return Hp;
00368     }

```

```

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 reflc_vec;
00407             if (vecx.myvector[0] < 0) {
00408                 reflc_vec = vecx + unit.Scalar_Mul(n);
00409             } else {
00410                 reflc_vec = vecx - unit.Scalar_Mul(n);
00411             }
00412
00413             double normalize_ref = reflc_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 = reflc_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     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     return ans;
00496 }
00497
00498 static Matrix Outer_Product(const My_Vec& u, const My_Vec& v) {
00505     Matrix Output(u.length, v.length);
00506     for(int i=0; i<u.length; i++) {
00507         for(int j=0; j<v.length; j++) {
00508             Output.MyMAT[i][j] = u.myvector[i] * v.myvector[j];
00509         }
00510     }
00511     return Output;
00512 }
00513
00514 Matrix Transpose() const {
00519     Matrix New_Mat(this->cols, this->rows);
00520     for (int i = 0; i<rows; i++) {
00521         for(int j=0; j<cols; j++) {
00522             New_Mat.MyMAT[j][i] = this->MyMAT[i][j];
00523         }
00524     }
00525     return New_Mat;
00526 }
00527
00528 Matrix Scalar_Mul(double k) const {
00534     Matrix new_mat(this->rows, this->cols);
00535     for (int i = 0; i<rows; i++) {
00536         for(int j=0; j<cols; j++) {
00537             new_mat.MyMAT[i][j] = this->MyMAT[i][j] * k;
00538         }
00539     }
00540     return new_mat;
00541 }
00542
00543 static Matrix eye(int a) {
00550     Matrix Identity(a, a);
00551     for(int i=0; i<a; i++) {
00552         Identity.MyMAT[i][i] = 1;
00553     }
00554     return Identity;
00555 }
00556
00557 static Matrix Ones(int a, int b) {
00564     Matrix Ones(a, b);
00565     for(int i=0; i<a; i++) {
00566         for(int j=0; j<b; j++) {
00567             Ones.MyMAT[i][j] = 1;
00568         }
00569     }
00570     return Ones;
00571 }
00572
00573 static Matrix Zeros(int a, int b) {
00580     Matrix Zeros(a, b);
00581     for(int i=0; i<a; i++) {
00582         for(int j=0; j<b; j++) {
00583             Zeros.MyMAT[i][j] = 0;
00584         }
00585     }
00586     return Zeros;
00587 }
00588
00589 };
00590
00591 class Linear_Solvers {
00596     static My_Vec SolveLU(const Matrix& A, const My_Vec& b);
00597     static My_Vec SolveQR(const Matrix& A, const My_Vec& b);
00598     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 struct QR_result_to_pass {
00612

```

```
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
```