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:

Chapter 2

File Index

2.1 File List

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

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: Wc = (_A(t)B) (_A(t)B)^T dt where _A(t) is the state transition matrix.

Parameters

System	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:

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

where:

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

This matrix has size n × nm, where:

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

Parameters

System	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: Wo = (C_A(t))^T (C_A(t)) dt where _A(t) is the state transition matrix.

Parameters

System	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 pn × n, where:

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

Parameters

System	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

System	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

r_min	The minimum radius for the grid
r_max	The maximum radius for the grid
r_samples	The number of radial samples
theta_samples	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:

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

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

System	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

System	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

System	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:

$$\text{rank}[sI - A \quad B] = n$$

for all eigenvalues s with 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

System	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

System	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:

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

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

Stability types:

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

Parameters

System	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

System	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

System	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

System	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

System	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

System	The discrete state space system to analyze
r_min	The minimum radius for the grid
r_max	The maximum radius for the grid
r_samples	The number of radial samples
theta_samples	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:

$$p(s) = s^n + a_{n-1}s^{n-1} + \cdots + a_1 s + a_0$$

The transformation uses the controllability matrix:

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

Parameters

System	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

System	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

System	The discrete state space system to transform

Returns

Discrete_StateSpace_System The transformed system in phase variable form

Exceptions

std::runtime_error	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

std::invalid_argument	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_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↩_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

std::runtime_error	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

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

3.6.2.2 Matrix() [2/2]

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

Copy constructor for Matrix.

Parameters

other	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

Householder	The Householder matrix to embed
A	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

a	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

std::invalid_argument	if the matrix is not square
std::runtime_error	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

x	The vector to multiply with

Returns

Resulting vector after multiplication

Exceptions

std::invalid_argument	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

a	Number of rows
b	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

other	The matrix to multiply with

Returns

Resulting matrix after multiplication

Exceptions

std::invalid_argument	if dimensions are not compatible

3.6.3.7 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

3.6.3.8 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

3.6.3.9 operator=()

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

Assignment operator for Matrix.

Parameters

other	The matrix to assign

Returns

Reference to this matrix

3.6.3.10 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

u	First vector
v	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

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

3.7 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

l	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

other	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

other	The other vector

Returns

Dot product result

Exceptions

std::invalid_argument	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

a	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

other	Vector to add to this one

Returns

New vector containing the sum

Exceptions

std::invalid_argument	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

other	The vector to subtract

Returns

Result of vector subtraction

Exceptions

std::invalid_argument	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

other	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

k	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	Position of the 1 in the unit vector
L	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	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

```
#ifndef ANALYSIS_HPP
#define ANALYSIS_HPP

#include <iostream>
#include <cmath>
#include <complex>
#include <Eigen/Dense>
#include "discrete_state_space.hpp"
#include <Eigen/SVD>
#include <eigen3/unsupported/Eigen/KroneckerProduct>

class Analysis {
public:
    static Eigen::MatrixXd compute_controllability_matrix(const Discrete_StateSpace_System& System)
    {
        int n = System.A.rows();
        int m = System.B.cols();
        Eigen::MatrixXd controllability_mat(n, n * m);

        for (int i = 0; i < n; ++i) {
            controllability_mat.block(0, i * m, n, m) = System.A.pow(i) * System.B;
        }

        return controllability_mat;
    }

    bool is_controllable(const Discrete_StateSpace_System& System)
    {
        Eigen::MatrixXd controllability_mat = compute_controllability_matrix(System);
        Eigen::FullPivLU<Eigen::MatrixXd> lu(controllability_mat);
        return lu.rank() == System.n_states;
    }

    static Eigen::MatrixXd compute_observability_matrix(const Discrete_StateSpace_System& System)
    {
        int n = System.A.rows();
        int p = System.C.rows();
        Eigen::MatrixXd observability_mat(p * n, System.A.cols());

        for (int i = 0; i < n; ++i) {
            observability_mat.block(i * p, 0, p, System.A.cols()) = System.C * System.A.pow(i);
        }

        return observability_mat;
    }

    bool is_observable(const Discrete_StateSpace_System& System)
    {
        Eigen::MatrixXd observability_mat = compute_observability_matrix(System);
        Eigen::FullPivLU<Eigen::MatrixXd> lu(observability_mat);
        return lu.rank() == System.n_states;
    }

    bool Linear_Stability_discrete(const Discrete_StateSpace_System& System)
    {
        Eigen::VectorXcd eigenvals=System.A.eigenvalues();
        for(int i=0;i<eigenvals.size();i++){

```

```
        if(std::abs(eigenvals[i])>=1.0){
            return false;
        }
    }
return true;}

static bool Linear_Stability_cont(const Discrete_StateSpace_System& System)
{
Eigen::VectorXcd eigenvals = System.A.eigenvalues();
for (int i = 0; i < eigenvals.size(); ++i) {
    if (eigenvals[i].real() >= 0.0) {
        return false;  // Unstable or marginally stable
    }
}
return true;
}



bool is_stabalizable_cont(const Discrete_StateSpace_System &System){
    Eigen::VectorXcd eigs=System.A.eigenvalues();
    int f=eigs.size();
    std::vector<bool> unstable_flag;
    int a=System.A.rows();
    int b=System.A.cols();
    for(int i=0;i<eigs.size();i++){
        if(eigs[i].real()>0){
            Eigen::MatrixXcd eye(a,b);
            eye=Eigen::MatrixXcd::Identity(a,b);
            Eigen::MatrixXcd PBH_part_1=eigs[i]*eye-System.A;
            int n=System.B.rows();
            int m=System.B.cols();
            Eigen::MatrixXcd PBH(n,n+m);
            PBH.block(0, 0, n, n) = PBH_part_1;
            PBH.block(0, n, n, m) = System.B.cast<std::complex<double»();
            Eigen::FullPivLU<Eigen::MatrixXcd> lu(PBH);
            if(lu.rank()<n){
                return false;
            }


        }

    }

    return true;

}


bool is_detectable_cont(const Discrete_StateSpace_System &System){
    Eigen::VectorXcd eigs=System.A.eigenvalues();
    int f=eigs.size();
    std::vector<bool> unstable_flag;
    int a=System.A.rows();
    int b=System.A.cols();
    for(int i=0;i<eigs.size();i++){
        if(eigs[i].real()>0){
            Eigen::MatrixXcd eye = Eigen::MatrixXcd::Identity(a, b);
            Eigen::MatrixXcd PBH_part_1=eigs[i]*eye-System.A;
            int n=System.A.rows();
            int p=System.C.rows();
            Eigen::MatrixXcd PBH(n + p, n);
            PBH.block(0, 0, n, n) = PBH_part_1;
            PBH.block(n, 0, p, n) = System.C.cast<std::complex<double»();
            Eigen::FullPivLU<Eigen::MatrixXcd> lu(PBH);
            if(lu.rank()<n){
                return false;
            }


        }

    }

    return true;

}


bool minimality_test_cont(const Discrete_StateSpace_System &System){

    if (is_controllable(System) && is_observable(System)) {

        return true;
    }
```

```
        else{
            return false;
        }
    }

    std::tuple<Eigen::MatrixXd, Eigen::MatrixXd> controllability_decomposition(const Discrete_StateSpace_System& System){
        Eigen::MatrixXd cont_mat=compute_controllability_matrix(System);
        int n=cont_mat.rows();
        Eigen::ColPivHouseholderQR <Eigen::MatrixXd> QR(cont_mat);
        Eigen::MatrixXd Q=QR.householderQ();
        int r=QR.rank();
        Eigen::MatrixXd T(n, n);
        T « Q.leftCols(r), Q.rightCols(n - r);

        Eigen::MatrixXd Aprime=(T.inverse())*System.A*T;
        Eigen::MatrixXd Bprime=(T.inverse())*System.B;
        Eigen::MatrixXd Cprime=System.C*T;

        Eigen::MatrixXd A_cc=Aprime.topLeftCorner(r,r);
        Eigen::MatrixXd A_cu=Aprime.topRightCorner(r,n-r);
        std::tuple<Eigen::MatrixXd, Eigen::MatrixXd> Ans={A_cc,A_cu};

        return Ans;
    }



    std::tuple<Eigen::MatrixXd, Eigen::MatrixXd> observability_decomposition(const Discrete_StateSpace_System& System){

        Eigen::MatrixXd obs_mat=compute_observability_matrix(System);
        int n=System.A.rows();
        Eigen::JacobiSVD<Eigen::MatrixXd> SVD(obs_mat);

        Eigen::MatrixXd V=SVD.matrixV();
        Eigen::MatrixXd VT=V.transpose();
        double tol = 1e-9;
        int r = (SVD.singularValues().array() > tol).count();

        Eigen::MatrixXd T(n, n);
        T « V.leftCols(r), V.rightCols(n - r);

        Eigen::MatrixXd Aprime=(T.inverse())*System.A*T;
        Eigen::MatrixXd Bprime=(T.inverse())*System.B;
        Eigen::MatrixXd Cprime=System.C*T;

        Eigen::MatrixXd A_oo=Aprime.topLeftCorner(r,r);
        Eigen::MatrixXd A_ou=Aprime.topRightCorner(r,n-r);
        std::tuple<Eigen::MatrixXd, Eigen::MatrixXd> Ans={A_oo,A_ou};

        return Ans;


    }


    Eigen::MatrixXd compute_controllability_gramian(const Discrete_StateSpace_System& System) {

        Eigen::MatrixXd Q = System.B * System.B.transpose();
        int n = System.A.rows();
        Eigen::MatrixXd I = Eigen::MatrixXd::Identity(n, n);
        Eigen::VectorXd vecQ = Eigen::Map<const Eigen::VectorXd>(Q.data(), Q.size());

        Eigen::MatrixXd kron1 = Eigen::kroneckerProduct(System.A, I);
        Eigen::MatrixXd kron2 = Eigen::kroneckerProduct(I, System.A);

        Eigen::VectorXd w = (kron1 + kron2).fullPivLu().solve(-vecQ);

        Eigen::MatrixXd W = Eigen::Map<Eigen::MatrixXd>(w.data(), n, n);

        return W;
}


    Eigen::MatrixXd compute_observability_gramian(const Discrete_StateSpace_System& System){

        Eigen::MatrixXd Q = (System.C) * System.C.transpose();
        int n = System.A.rows();
        Eigen::MatrixXd I = Eigen::MatrixXd::Identity(n, n);
        Eigen::VectorXd vecQ = Eigen::Map<const Eigen::VectorXd>(Q.data(), Q.size());

        Eigen::MatrixXd kron1 = Eigen::kroneckerProduct(System.A, I);
        Eigen::MatrixXd kron2 = Eigen::kroneckerProduct(I, System.A);

        Eigen::VectorXd w = (kron1 + kron2).fullPivLu().solve(-vecQ);

```

```
        Eigen::MatrixXd W = Eigen::Map<Eigen::MatrixXd>(w.data(), n, n);

        return W;
    }

    std::vector<std::complex<double» poles(const Discrete_StateSpace_System& System){

        Eigen::EigenSolver<Eigen::MatrixXd> eigen_solver(System.A);
        Eigen::VectorXcd eigvals = eigen_solver.eigenvalues();
        std::vector<std::complex<double» eigs(eigvals.data(), eigvals.data() + eigvals.size());

        return eigs;

    }

    std::vector<std::complex<double» generate_z_grid(
    double r_min, double r_max,
    int r_samples,
    int theta_samples)
    {
    std::vector<std::complex<double» z_grid;
        z_grid.reserve(r_samples * theta_samples);

        for (int i = 0; i < r_samples; ++i) {
            double r = r_min + i * (r_max - r_min) / (r_samples - 1);
            for (int j = 0; j < theta_samples; ++j) {
                double theta = 2.0 * M_PI * j / theta_samples;
                std::complex<double> z = std::polar(r, theta);
                z_grid.push_back(z);
            }
        }
        return z_grid;
}

    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){


        int n =System.A.rows();
        int p= System.C.rows();
        int m= System.B.cols();
        Eigen::MatrixXcd R(n + p, n + m);

        std::vector<std::complex<double» zgrid=generate_z_grid(r_min,r_max,r_samples,theta_samples);
        std::vector<std::complex<double» zeros;
        Eigen::MatrixXcd zI_minus_A;

        Eigen::MatrixXcd eye=Eigen::MatrixXd::Identity(n, n);
        int r;
        double tol = 1e-9;
        for(int i=0;i<zgrid.size();i++){
            zI_minus_A=(zgrid[i]*eye)-(System.A.cast<std::complex<double»);
            R.block(0, 0, n, n) = zI_minus_A;
            R.block(0, n, n, m) = -System.B.cast<std::complex<double»();
            R.block(n, 0, p, n) = System.C.cast<std::complex<double»();
            R.block(n, n, p, m) = System.D.cast<std::complex<double»();

            Eigen::JacobiSVD<Eigen::MatrixXcd>SVD(R);
            r = (SVD.singularValues().array() > tol).count();
            if(r<n+p){

                zeros.push_back(zgrid[i]);
            }


        }

        return zeros;
    }

    Discrete_StateSpace_System Kalman_Decomp( const Discrete_StateSpace_System& System){

        std::tuple<Eigen::MatrixXd, Eigen::MatrixXd> Cont,Obs;

        Obs=observability_decomposition(System);
        Cont=controllability_decomposition(System);

        Eigen::MatrixXd Obs_subspace=std::get<0>(Obs);
        Eigen::MatrixXd Cont_subspace=std::get<0>(Cont);
        Eigen::MatrixXd Diff(Obs_subspace.rows(), Obs_subspace.cols() + Cont_subspace.cols());
        Diff « Obs_subspace, -Cont_subspace;
        Eigen::FullPivLU<Eigen::MatrixXd> lu1(Diff);
        Eigen::MatrixXd Null_S = lu1.kernel();

```

```
        int k=Obs_subspace.cols();

        Eigen::MatrixXd Null_S_S=Null_S.topRows(k);
        Eigen::MatrixXd Basis_1=Null_S_S*Obs_subspace;


        Eigen::MatrixXd Obs_ortho=(Obs_subspace.transpose()).fullPivLu().kernel();
        Eigen::MatrixXd Cont_ortho=(Cont_subspace.transpose()).fullPivLu().kernel();

        Eigen::MatrixXd Diff2(Obs_ortho.rows(), Obs_ortho.cols() + Cont_subspace.cols());
        Diff2 « Obs_ortho, -Cont_subspace;
        Eigen::MatrixXd Null_S2=Diff2.fullPivLu().kernel();

        int l=Obs_ortho.cols();


        Eigen::MatrixXd Null_S_S2=Null_S2.topRows(l);
        Eigen::MatrixXd Basis_2=Null_S_S2*Obs_ortho;


        Eigen::MatrixXd Diff3(Obs_subspace.rows(), Obs_subspace.cols() + Cont_ortho.cols());
        Diff3 « Obs_subspace, -Cont_ortho;
        Eigen::MatrixXd Null_S3=Diff3.fullPivLu().kernel();


        Eigen::MatrixXd Null_S_S3=Null_S3.topRows(k);
        Eigen::MatrixXd Basis_3=Null_S_S3*Obs_subspace;



        Eigen::MatrixXd Diff4(Obs_ortho.rows(), Obs_subspace.cols() + Cont_ortho.cols());
        Diff4« Obs_ortho, -Cont_ortho;
        Eigen::MatrixXd Null_S4=Diff3.fullPivLu().kernel();


        Eigen::MatrixXd Null_S_S4=Null_S4.topRows(l);
        Eigen::MatrixXd Basis_4=Null_S_S4*Obs_ortho;

        int n=Basis_1.rows();
        Eigen::MatrixXd T(n, n);
        T « Basis_1, Basis_2, Basis_3, Basis_4;

        if (T.fullPivLu().isInvertible()) {
        }
        else {
            Eigen::MatrixXd Q = QR.householderQ();
    T = Q.leftCols(n);  // Truncate if necessary

            }

            Eigen::MatrixXd T_inv = T.inverse();


            Eigen::Matrix A_new=T_inv*System.A*T;
            Eigen::Matrix B_new=T_inv*System.B;
            Eigen::Matrix C_new=System.C*T;
            Eigen::Matrix D_new=System.D;


            Discrete_StateSpace_System Decomp;
            Decomp.A=A_new;
            Decomp.B=B_new;
            Decomp.C=C_new;
            Decomp.D=D_new;

            return Decomp;


    }

    private:
    };

#endif
```

4.2 forms.hpp

```
#ifndef FORMS_HPP
#define FORMS_HPP

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

```
#include <complex>
#include <Eigen/Dense
#include "discrete_state_space.hpp"
#include "analysis.hpp"
#include <Eigen/SVD>
#include <eigen3/unsupported/Eigen/KroneckerProduct>


class Forms{

    public:
    Discrete_StateSpace_System Cont_Cannonical_form(const Discrete_StateSpace_System& System)
    {


        int n = System.A.rows();
        Eigen::MatrixXd B0 = Eigen::MatrixXd::Identity(n, n);
        Eigen::MatrixXd Bk;
        std::vector<double> coeffs;
        double ak_1 = 0;
        double trk = 0;

        for (int k = 1; k <= n; k++) {
            if (k == 1) {
                // First iteration: trace(A), a0 coefficient, initialize Bk
                trk = System.A.diagonal().sum();  // trace(A)
                ak_1 = -trk / k;
                coeffs.push_back(ak_1);
                Bk = B0;
            } else {
                // Subsequent iterations: compute Bk, trace, and coefficient
                Eigen::MatrixXd Bk_1 = System.A * Bk + ak_1 * B0;
                trk = (System.A * Bk_1).diagonal().sum();
                ak_1 = -trk / k;
                coeffs.push_back(ak_1);
                Bk = Bk_1;
            }
        };


        Eigen::MatrixXd Accf = Eigen::MatrixXd::Zero(n, n);
        for(int i=0;i<n-1;i++){

            Accf(i,i+1)=1;

        }
        for(int j=0;j<n;j++){

            Accf(n-1,j)=-coeffs[j];

        }

        //Compute Controlability Matrix
        Analysis A;
        Eigen::MatrixXd T = A.compute_controllability_matrix(System);
        Eigen::MatrixXd T_inv=T.inverse();

        // Ensure the constructor matches the expected signature, e.g. (A, B, C, D)
        Discrete_StateSpace_System new_system = System;
        new_system.A = Accf;
        new_system.B = T_inv * System.B;
        new_system.C = System.C * T;
        new_system.D = System.D;




        return new_system;
    }


        Discrete_StateSpace_System obs_Cannonical_form(const Discrete_StateSpace_System& System){

        int n = System.A.rows();
        Eigen::MatrixXd B0 = Eigen::MatrixXd::Identity(n, n);
        Eigen::MatrixXd Bk;
        std::vector<double> coeffs;
        double ak_1 = 0;
        double trk = 0;

        for (int k = 1; k <= n; k++) {
            if (k == 1) {
                // First iteration: trace(A), a0 coefficient, initialize Bk
                trk = System.A.diagonal().sum();  // trace(A)
                ak_1 = -trk / k;
                coeffs.push_back(ak_1);
                Bk = B0;
```

```
            } else {
                // Subsequent iterations: compute Bk, trace, and coefficient
                Eigen::MatrixXd Bk_1 = System.A * Bk + ak_1 * B0;
                trk = (System.A * Bk_1).diagonal().sum();
                ak_1 = -trk / k;
                coeffs.push_back(ak_1);
                Bk = Bk_1;
            }
        };

        Eigen::MatrixXd Aocf = Eigen::MatrixXd::Zero(n, n);
        for(int i=0;i<n-1;i++){

            Aocf(i,i+1)=1;

        }
        for(int j=0;j<n;j++){

        Aocf(j,0) = -coeffs[n - j - 1];
        }

        //Compute Controlability Matrix
        Analysis A;
        Eigen::MatrixXd T = Analysis::compute_observability_matrix(System);
        Eigen::MatrixXd T_inv=T.inverse();

        // Ensure the constructor matches the expected signature, e.g. (A, B, C, D)
        Discrete_StateSpace_System new_system = System;
        new_system.A = Aocf;
        new_system.B = T_inv * System.B;
        new_system.C = System.C * T;
        new_system.D = System.D;

        return new_system;


     }


    Discrete_StateSpace_System Phase_Variable_Form(const Discrete_StateSpace_System& System) {
    int n = System.A.rows();

    // Build controllability matrix and check invertibility
    Eigen::MatrixXd T = Analysis::compute_controllability_matrix(System);
    if (T.determinant() == 0)
        throw std::runtime_error("System is not controllable. Cannot convert to phase variable form.");
    }
    Eigen::MatrixXd T_inv = T.inverse();

    // Transform system to phase variable form
    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;
    // D remains unchanged

    return new_system;
}


    Discrete_StateSpace_System Schur_Form(const Discrete_StateSpace_System& System){


        Eigen::RealSchur<Eigen::MatrixXd> Schur(System.A)
        Eigen::MatrixXd Q = schur.matrixU();
        Eigen::MatrixXd T = schur.matrixT();

        Eigen::MatrixXd Q_inv=Q.transpose();

        Discrete_StateSpace_System new_system = System;
        new_system.A = T;
        new_system.B = Q_inv * System.B;
        new_system.C = System.C * Q;

        return new_system
}


        Discrete_StateSpace_System Diagonalize(const Discrete_StateSpace_System& System) {
            Eigen::EigenSolver<Eigen::MatrixXd> es(System.A);
            Eigen::MatrixXcd eigenvectors = es.eigenvectors();
            Eigen::VectorXcd eigenvalues = es.eigenvalues();

            // Check if matrix is diagonalizable (eigenvectors matrix invertible)
            if (eigenvectors.determinant() == 0) {
                throw std::runtime_error("Matrix is not diagonalizable");
```

```
            }

            Eigen::MatrixXcd P = eigenvectors;
            Eigen::MatrixXcd P_inv = P.inverse();

            Eigen::MatrixXcd D = eigenvalues.asDiagonal();

            Discrete_StateSpace_System new_system;

            new_system.A = (P_inv * System.A.cast<std::complex<double»() * P).real();
            new_system.B = (P_inv * System.B.cast<std::complex<double»()).real();
            new_system.C = (System.C.cast<std::complex<double»() * P).real();
            new_system.D = System.D;

            return new_system;
        }



    private:




};
#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.

```

#ifndef LINEAR_SOLVER_HPP
#define LINEAR_SOLVER_HPP

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

class Linear_Solvers {
public:
    static My_Vec SolveLU(const Matrix& A, const My_Vec& b) {

        LUResult LU_temp=A.L_U();
        LUResult_to_pass LU=conv_LU(LU_temp);
        My_Vec pb= ApplyPermutation(LU.P,b);
        My_Vec fwd_sub=ForwardSubstitution(LU.L,pb);
        My_Vec bck_sub=BackwardSubstitution(LU.U,fwd_sub);

        return bck_sub;

    }
    static My_Vec SolveQR(const Matrix& A, const My_Vec& b) {

        QRresult decomp_temp=A.QR_fact();
        QR_result_to_pass decomp=conv_QR(decomp_temp);
        My_Vec qTb=(decomp.Q.Transpose()).multiply(b);
        My_Vec x=BackwardSubstitution(decomp.R,qTb);
        return x;
    };
    static Matrix Inverse(const Matrix& A) {

        Matrix I=Matrix::eye(A.rows);
        LUResult decomp_temp=A.L_U();
        LUResult_to_pass decomp=conv_LU(decomp_temp);
```

```
        Matrix L=decomp.L;
        Matrix U=decomp.U;
        My_Vec e_i = My_Vec::ones(A.rows);
        Matrix inverse_matrix=Matrix::Zeros(A.rows,A.cols);
        for(int j=0;j<I.cols;j++){
            e_i.Scalar_Mul(0);
            e_i.myvector[j] = 1;
            My_Vec Pe_j = ApplyPermutation(decomp.P, e_i);

            My_Vec F1=ForwardSubstitution(L,Pe_j);
            My_Vec B1=BackwardSubstitution(U,F1);
            for (int row = 0; row < A.rows; row++) {
                inverse_matrix.MyMAT[row][j] = B1.myvector[row];
            }
        }

        return inverse_matrix;

    };
    static My_Vec ForwardSubstitution(const Matrix& L_1, const My_Vec& b) {

       My_Vec solution_VEC=My_Vec::ones(L_1.rows);
       solution_VEC.Scalar_Mul(0);

       Matrix L= L_1;
       std::vector<double> knowns;
       double new_unknown;
       for(int i=0;i<L.rows;i++){

        double known_sum=0;
        for(int j=0;j<i;j++){
            known_sum+=L.MyMAT[i][j]*solution_VEC.myvector[j];
        }
        new_unknown=(b.myvector[i]-known_sum)/L.MyMAT[i][i];


        solution_VEC.myvector[i]=new_unknown;


       }

       return solution_VEC;


    };
    static My_Vec BackwardSubstitution(const Matrix& U_1, const My_Vec& b) {
        My_Vec solution_VEC=My_Vec::ones(U_1.rows);
        solution_VEC.Scalar_Mul(0);

        Matrix U= U_1;
        std::vector<double> knowns;
        double new_unknown;
        for(int i=U.rows-1;i>=0;i--){

         double known_sum=0;
         for (int j = i + 1; j < U.cols; j++){
             known_sum+=U.MyMAT[i][j]*solution_VEC.myvector[j];
         }
         new_unknown=(b.myvector[i]-known_sum)/U.MyMAT[i][i];


         solution_VEC.myvector[i]=new_unknown;


        }

        return solution_VEC;


     };
     static My_Vec ApplyPermutation(const std::vector<int>& P,const My_Vec& V){
        if (P.size() != V.myvector. size()) {
            throw std::invalid_argument("Permutation size must match vector size");
        }
        My_Vec result;

        My_Vec result = My_Vec::Zeros(V.myvector.size());

        for (size_t i = 0; i < P.size(); i++) {
            result.myvector[i] = V.myvector[P[i]];
        }

        return result;
    }

    static double determinant(const Matrix& A) {
    LUResult lu_temp = A.L_U();
```

```
    LUResult_to_pass lu=conv_LU(lu_temp);
    double det = 1.0;
    for(int i = 0; i < A.rows; i++) {
        det *= lu.U.MyMAT[i][i];
    }
    return det;
}


    static My_Vec solve_linear_system_LU(const Matrix& A, const My_Vec& b) {
    return SolveLU(A, b);
}

    static My_Vec solve_linear_system_QR(const Matrix& A, const My_Vec& b) {
    return SolveQR(A, b);
}

    };
#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.

```
//gcc (Ubuntu 13.3.0-6ubuntu2~24.04) 13.3.0
// Standard library includes for basic operations
#include <iostream>
#include <vector>
#include <cmath>

#ifndef MATRIX_MATH_HPP
#define MATRIX_MATH_HPP

struct LUResult {
    std::vector<std::vector<double» L;
    std::vector<std::vector<double» U;
    std::vector<int> P;
};

struct QRresult {
     std::vector<std::vector<double» Q;
     std::vector<std::vector<double» R;
};

class My_Vec {
public:
    int length;
    std::vector<double> myvector;

    My_Vec(int l=1) {
        length = l;
        myvector.resize(length);
    }

    My_Vec operator+ (const My_Vec& other) const {
        if(this->length != other.length) {
            throw std::invalid_argument("Invalid lengths");
        }
        My_Vec Vec(this->length);
        for (int i=0; i<this->length; i++) {
            Vec.myvector[i] = this->myvector[i] + other.myvector[i];
        }
        return Vec;
    }

    static My_Vec ones(int a) {
        My_Vec ones_vec(a);
        for(int i=0; i<a; i++) {
            ones_vec.myvector[i] = 1;
        }
        return ones_vec;
    }

    My_Vec operator- (const My_Vec& other) const{

        if(this->length != other.length) {
            throw std::invalid_argument("Invalid lengths");
        }
        My_Vec Vec(this->length);

        for (int i=0; i<this->length; i++) {
            Vec.myvector[i] = this->myvector[i] - other.myvector[i];
        }
        return Vec;
    }

    double Norm() const {
        double a=0;
        for (int i=0; i<this->length; i++) {
            a += pow(this->myvector[i], 2.0);
        }
        return sqrt(a);
    }

    double dot (const My_Vec& other) const {
        if(this->length != other.length) {
            throw std::invalid_argument("Invalid lengths");
        }

        double a=0;
        for(int i=0; i<this->length; i++) {
            a += this->myvector[i] * other.myvector[i];
        }
        return a;
    }
```

```
    My_Vec Scalar_Mul(double k) const {
        My_Vec New(this->length);
        for (int i=0; i<this->length; i++) {
            New.myvector[i] = k * this->myvector[i];
        }
        return New;
    }

    My_Vec(const My_Vec& other)
    : length(other.length), myvector(other.myvector) {};

    My_Vec& operator=(const My_Vec& other) {
        if (this != &other) {
            length = other.length;
            myvector = other.myvector;
        }
        return *this;
    }

    static My_Vec unit_vec(int i, int L) {
        My_Vec unit_vec(L);
        for (int j=0; j<L; j++) {
            unit_vec.myvector[j] = (j == i) ? 1 : 0;
        }
        return unit_vec;
    }

    static My_Vec Zeros(const int& i) {
        My_Vec unit_vec(i);
        for (int j=0; j<i; j++) {
            unit_vec.myvector[j] = 0;
        }
        return unit_vec;
    }
};

class Matrix {
public:
    int rows;
    int cols;
    std::vector<std::vector<double» MyMAT;

    Matrix(int rs=1, int cs=1) {
        rows = rs;
        cols = cs;
        MyMAT.resize(rs, std::vector<double>(cs));
    }

    Matrix(const Matrix& other)
    : rows(other.rows), cols(other.cols), MyMAT(other.MyMAT) {}

    Matrix& operator=(const Matrix& other) {
        if (this != &other) {
            rows = other.rows;
            cols = other.cols;
            MyMAT = other.MyMAT;
        }
        return *this;
    }

    Matrix operator+(const Matrix& other) const {
        Matrix New_Mat(other.rows, other.cols);
        if (this->cols != other.cols || other.rows != this->rows) {
            throw std::invalid_argument("Dimension mismatch");
        } else {
            for (int i=0; i<rows; i++) {
                for (int j=0; j<cols; j++) {
                    New_Mat.MyMAT[i][j] = other.MyMAT[i][j] + this->MyMAT[i][j];
                }
            }
        }
        return New_Mat;
    }

    Matrix operator- (const Matrix& other) const {
        Matrix New_Mat(other.rows, other.cols);
        if (this->cols != other.cols || other.rows != this->rows) {
            throw std::invalid_argument("Dimension mismatch");
        } else {
            for (int i=0; i<rows; i++) {
                for (int j=0; j<cols; j++) {
                    New_Mat.MyMAT[i][j] = this->MyMAT[i][j] - other.MyMAT[i][j];
                }
            }
        }
        return New_Mat;
```

```
    }

    LUResult L_U() const {
        // Check for square matrix
        if(this->rows != this->cols) {
            throw std::invalid_argument("Not square");
        }

        Matrix L = eye(this->rows);
        Matrix U = Zeros(this->rows, this->cols);
        std::vector<int> P(this->rows);
        for(int i = 0; i < this->rows; i++) {
            P[i] = i;
        }

        Matrix A_work(*this);

        for(int j = 0; j < this->cols; j++) {
            int pivot_row = j;
            double max_val = std::abs(A_work.MyMAT[j][j]);

            for(int i = j+1; i < this->rows; i++) {
                if(std::abs(A_work.MyMAT[i][j]) > max_val) {
                    pivot_row = i;
                    max_val = std::abs(A_work.MyMAT[i][j]);
                }
            }

            if(pivot_row != j) {
                for(int k = 0; k < this->cols; k++) {
                    std::swap(A_work.MyMAT[j][k], A_work.MyMAT[pivot_row][k]);
                }

                std::swap(P[j], P[pivot_row]);

                if(j > 0) {
                    for(int k = 0; k < j; k++) {
                        std::swap(L.MyMAT[j][k], L.MyMAT[pivot_row][k]);
                    }
                }
            }

            if(std::abs(A_work.MyMAT[j][j]) < 1e-10) {
                throw std::runtime_error("Matrix is singular or nearly singular");
            }

            for(int i = 0; i <= j; i++) {
                double sum = 0.0;
                for(int k = 0; k < i; k++) {
                    sum += L.MyMAT[i][k] * U.MyMAT[k][j];
                }
                U.MyMAT[i][j] = A_work.MyMAT[i][j] - sum;
            }

            for(int i = j+1; i < this->rows; i++) {
                double sum = 0.0;
                for(int k = 0; k < j; k++) {
                    sum += L.MyMAT[i][k] * U.MyMAT[k][j];
                }
                L.MyMAT[i][j] = (A_work.MyMAT[i][j] - sum) / U.MyMAT[j][j];
            }
        }

        LUResult result;
        result.L = L.MyMAT;
        result.U = U.MyMAT;
        result.P = P;
        return result;
    }

    static My_Vec UV(int i, int L) {
        My_Vec unit_vec(L);
        for (int j=0; j<L; j++) {
            unit_vec.myvector[j] = (j == i) ? 1 : 0;
        }
        return unit_vec;
    }

    static Matrix Embed(const Matrix& Householder, const Matrix& A) {
        Matrix Hp = eye(A.rows);
        int i = A.rows - Householder.rows;
        for (int a = 0; a < Householder.rows; a++) {
            for (int b = 0; b < Householder.cols; b++) {
                Hp.MyMAT[i+a][i+b] = Householder.MyMAT[a][b];
            }
        }
        return Hp;
```

```
    }

    QRresult QR_fact() const {
        Matrix Q = eye(this->rows);
        Matrix Aupdate = *this;

        for (int i = 0; i < std::min(this->rows-1, this->cols); i++) {
            My_Vec vecx(this->rows - i);
            for (int row = i; row < this->rows; row++) {
                vecx.myvector[row - i] = Aupdate.MyMAT[row][i];
            }

            double n = vecx.Norm();

            My_Vec unit = My_Vec::unit_vec(0, this->rows - i);  // Use consistent naming

            My_Vec reflec_vec;
            if (vecx.myvector[0] < 0) {
                reflec_vec = vecx + unit.Scalar_Mul(n);
            } else {
                reflec_vec = vecx - unit.Scalar_Mul(n);
            }

            double normalize_ref = reflec_vec.Norm();

            // Check for zero vector (avoid division by zero)
            if (normalize_ref < 1e-10) {
                continue;  // Skip this iteration if vector is too small
            }

            My_Vec V = reflec_vec.Scalar_Mul(1.0 / normalize_ref);

            // Create Householder matrix H = I - 2*vv^T
            Matrix I = Matrix::eye(V.length);
            Matrix vvT = Matrix::Outer_Product(V, V);
            Matrix Householder = I - vvT.Scalar_Mul(2.0);

            // Embed the Householder matrix into a larger identity matrix
            Matrix Hprime = Matrix::eye(this->rows);
            for (int r = 0; r < Householder.rows; r++) {
                for (int c = 0; c < Householder.cols; c++) {
                    Hprime.MyMAT[i + r][i + c] = Householder.MyMAT[r][c];
                }
            }

            Aupdate = Hprime * Aupdate;
            Q = Q * Hprime;
        }

        QRresult QR;
        std::vector<std::vector<double>> Q_n = Q.Transpose().MyMAT;
        std::vector<std::vector<double>> R_n = Aupdate.MyMAT;

        QR.Q = Q_n;
        QR.R = R_n;
        return QR;
    }

    Matrix operator*(const Matrix& other) const {
        if (this->cols != other.rows) {
            throw std::invalid_argument("Dimension mismatch");
        }

        Matrix Ans(this->rows, other.cols); // Ensure constructor handles allocation

        for (int i = 0; i < this->rows; ++i) {
            for (int j = 0; j < other.cols; ++j) {
                double sum_of_multiples = 0.0;
                for (int k = 0; k < this->cols; ++k) {
                    sum_of_multiples += this->MyMAT[i][k] * other.MyMAT[k][j];
                }
                Ans.MyMAT[i][j] = sum_of_multiples;
            }
        }

        return Ans;
    }


    My_Vec multiply(const My_Vec& x) const {
        if(this->cols != x.length) {
            throw std::invalid_argument("Dimension mismatch");
        }

        My_Vec ans(this->rows);

        for (int i=0; i<this->rows; i++) {
```

```
            double dot_prod = 0;
            for(int j=0; j<this->cols; j++) {
                dot_prod += this->MyMAT[i][j] * x.myvector[j];
            }
            ans.myvector[i] = dot_prod;
        }

        return ans;
    }

    static Matrix Outer_Product(const My_Vec& u, const My_Vec& v) {
        Matrix Output(u.length, v.length);
        for(int i=0; i<u.length; i++) {
            for(int j=0; j<v.length; j++) {
                Output.MyMAT[i][j] = u.myvector[i] * v.myvector[j];
            }
        }
        return Output;
    }

    Matrix Transpose() const {
        Matrix New_Mat(this->cols, this->rows);
        for (int i = 0; i<rows; i++) {
            for(int j=0; j<cols; j++) {
                New_Mat.MyMAT[j][i] = this->MyMAT[i][j];
            }
        }
        return New_Mat;
    }

    Matrix Scalar_Mul(double k) const {
        Matrix new_mat(this->rows, this->cols);

        for (int i = 0; i<rows; i++) {
            for(int j=0; j<cols; j++) {
                new_mat.MyMAT[i][j] = this->MyMAT[i][j] * k;
            }
        }
        return new_mat;
    }

    static Matrix eye(int a) {
        Matrix Identity(a, a);
        for(int i=0; i<a; i++) {
            Identity.MyMAT[i][i] = 1;
        }
        return Identity;
    }

    static Matrix Ones(int a, int b) {
        Matrix Ones(a, b);
        for(int i=0; i<a; i++) {
            for(int j=0; j<b; j++) {
                Ones.MyMAT[i][j] = 1;
            }
        }
        return Ones;
    }

    static Matrix Zeros(int a, int b) {
        Matrix Zeros(a, b);
        for(int i=0; i<a; i++) {
            for(int j=0; j<b; j++) {
                Zeros.MyMAT[i][j] = 0;
            }
        }
        return Zeros;
    }
};

class Linear_Solvers {
    static My_Vec SolveLU(const Matrix& A, const My_Vec& b);
    static My_Vec SolveQR(const Matrix& A, const My_Vec& b);
    static My_Vec Inverse(const Matrix& A);

    static My_Vec ForwardSubstitution(const Matrix& L, const My_Vec& b);
    static My_Vec BackwardSubstitution(const Matrix& U, const My_Vec& y);
};

struct LUResult_to_pass {
   Matrix L;
   Matrix U;
   std::vector<int> P;
};


struct QR_result_to_pass {
```

```
     Matrix Q;
     Matrix R;
};


LUResult_to_pass conv_LU(const LUResult& LU) {
    int a = LU.L.size();
    int b = LU.L[0].size();

    Matrix L(a, b);
    L.MyMAT = LU.L;


    int c = LU.U.size();
    int d = LU.U[0].size();

    Matrix U(c, d);
    U.MyMAT = LU.U;

    LUResult_to_pass LU_new;

    LU_new.L = L;
    LU_new.U = U;
    LU_new.P = LU.P;
    return LU_new;
}


QR_result_to_pass conv_QR(const QRresult& QR) {
    int a = QR.Q.size();
    int b = QR.Q[0].size();

    Matrix Q(a, b);
    Q.MyMAT = QR.Q;


    int c = QR.R.size();
    int d = QR.R[0].size();

    Matrix R(c, d);
    R.MyMAT = QR.R;

    QR_result_to_pass QR_new;

    QR_new.Q = Q;
    QR_new.R = R;
    return QR_new;
}


#endif
```