QMath Matrix Library Reference Manual

Vilas K. Chitrakaran

Mon Sep 11 14:52:24 2006

Contents

Chapter 1

QMath Matrix Library Hierarchical Index

1.1 QMath Matrix Library Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

Chapter 2

QMath Matrix Library Class Index

2.1 QMath Matrix Library Class List

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

Chapter 3

QMath Matrix Library File Index

3.1 QMath Matrix Library File List

Here is a list of all files with brief descriptions:

Chapter 4

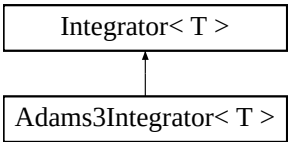
QMath Matrix Library Class Documentation

4.1 Adams3Integrator< T > Class Template Reference

Numerical integration using Adam's 3'rd order method.

`#include <Adams3Integrator.hpp>`

Inheritance diagram for Adams3Integrator< T >::



Public Member Functions

- Adams3Integrator ()
- Adams3Integrator (double period, const T &init)
- virtual ~Adams3Integrator ()
- virtual void reset (const T &init)
- virtual T integrate (const T &input)

4.1.1 Detailed Description

template<class T = double> class Adams3Integrator< T >

Numerical integration using Adam's 3'rd order method.

Example Program:

```
//==========================================================================
// Package              : The Math Library - Ex
// Authors              : Vilas Kumar Chitrakaran
// Start Date           : Wed Dec 20 11:08:28 GMT 2000
```

```
// Compiler             : GNU C++ 2.95.3 and above
// ---------------------------------------------------------------------------
// File: Adams3Integrator.t.cpp
// Example program for the Integrator.
//============================================================================

//============================================================================
// Adams3Integrator.t.cpp
//----------------------------------------------------------------------------
// Integrates a trignometric function using Adams method, and compares
// result with analytical integration.
//============================================================================

#include "Adams3Integrator.hpp"
#include "ColumnVector.hpp"

#include <stdio.h>
#include <math.h>
#ifndef M_PI
 #define M_PI 3.14159265358979323846
#endif

int main()
{
 FILE *outfile;                          // File to store results
 double velocity;                        // some data
 double position_adams;                  // numerical integral
 double position_actual;                 // actual integral
 double initValue;                       // initial value of integration
 double sampling_period;                 // sampling period
 Adams3Integrator< double > myIntegrator; // numerical integrator

 outfile = fopen("Adams3Integrator.dat", "w+");
 initValue = 0;
 sampling_period = 0.001;

 myIntegrator.setSamplingPeriod(sampling_period);
 myIntegrator.reset(initValue);

 fprintf(outfile, "%s\n%s %s %s\n", "%Adams 3rd order integrator output file",
        "%velocity", "position_adams", "position_actual" );
 for (int i=0; i<1000; i++)
 {
  // input data
  velocity = sin(2*M_PI*i*sampling_period);

  // integrate
  position_adams = myIntegrator.integrate(velocity);
  position_actual = 1.0/(2*M_PI) * (1 - cos(2*M_PI*i*sampling_period));

  // simply write the outputs to a file...
  fprintf(outfile, "%f %f %f\n", velocity, position_adams, position_actual);
 }

 fclose(outfile);
 return(0);
}
```

4.1.2 Constructor & Destructor Documentation

4.1.2.1 template<class T = double> Adams3Integrator< T >::Adams3Integrator () [inline]

The default constructor. The sampling period is set to default of 0.001 seconds. The initial output value is set to 0.

4.1.2.2 template<class T = double> Adams3Integrator< T >::Adams3Integrator (double *period*, const T & *init*) [inline]

The constructor with initialization for the sampling period and initial Value.

Parameters:

period The sampling period in seconds.

init The initial value at the start of integration.

4.1.2.3 template<class T = double> virtual Adams3Integrator< T >::~Adams3Integrator () [inline, virtual]

4.1.3 Member Function Documentation

4.1.3.1 template<class T = double> virtual void Adams3Integrator< T >::reset (const T & *init*) [virtual]

The default destructor This function resets the output of the Integrator to the value *value* and further integration restarts from this initial value.

Reimplemented from Integrator< T >.

4.1.3.2 template<class T = double> virtual T Adams3Integrator< T >::integrate (const T & *input*) [inline, virtual]

This function provides the numerical method for integration.

Parameters:

input The current value of the time-varying signal to be integrated.

Reimplemented from Integrator< T >.

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

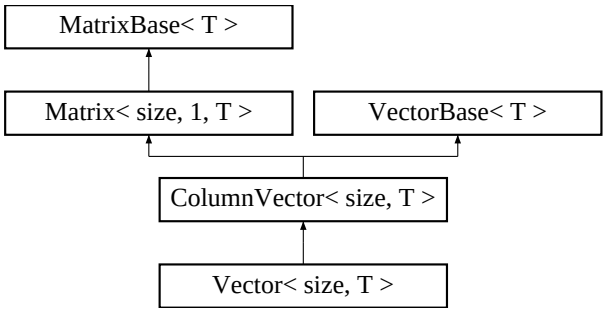
- Adams3Integrator.hpp

4.2 ColumnVector< size, T > Class Template Reference

A class for column vectors.

`#include <ColumnVector.hpp>`

Inheritance diagram for ColumnVector< size, T >::



Public Member Functions

- ColumnVector ()
- ColumnVector (const ColumnVector< size, T > &v)
- ColumnVector (const Matrix< size, 1, T > &m)
- ColumnVector (const Vector< size, T > &v)
- virtual ~ColumnVector ()
- virtual T * getElementsPointer () const
- virtual T getElement (int index) const
- virtual void setElement (int index, T value)
- virtual bool isRowVector () const
- virtual int getNumElements () const
- T operator() (int index) const
- T & operator() (int index)
- ColumnVector< size, T > & operator= (const VectorBase< T > &v)
- MatrixInitializer< T > operator= (const T &value)

4.2.1 Detailed Description

template<int size, class T = double> class ColumnVector< size, T >

A class for column vectors.

The class `ColumnVector` is derived from the base classes `Matrix` and `VectorBase`, and provides methods for operations such as cross product, dot product and element-by-element multiplication.

Example Program:

```
//============================================================================
// Package              : The Math Library - Ex
// Authors              : Vilas Kumar Chitrakaran
// Start Date    : Wed Dec 20 11:08:28 GMT 2000
// Compiler             : GNU C++ 2.95.3 and above
// ---------------------------------------------------------------------------
```

```
// File: Vector.t.cpp
// Example program for the vector classes.
//========================================================================

#include "Vector.hpp"
#include "RowVector.hpp"

using namespace std;

int main()
{
 Vector<3> v1, v2, v3;

 v1 = 1, 1, 2;
 v2 = 2, 3, 4;
 double dp;

 // dot product: component of v1 along v2
 dp = dotProduct(v1, v2);
 cout << "Dot product: v1 . v2 = " << dp << endl;

 // cross product: v1 x v2
 v3 = crossProduct(v1, v2);
 cout << "Cross product: v1 x v2 = " << transpose(v3) << endl;

 // 2-norm of a vector
 cout << "norm(v1): " << v1.norm() << endl;
 return 0;
}
```

4.2.2 Constructor & Destructor Documentation

4.2.2.1 template<int size, class T = double> ColumnVector< size, T >::ColumnVector () [inline]

The default constructor. The elements are not initialized.

4.2.2.2 template<int size, class T = double> ColumnVector< size, T >::ColumnVector (const ColumnVector< size, T > & *v*) [inline]

Copy Constructor.

4.2.2.3 template<int size, class T = double> ColumnVector< size, T >::ColumnVector (const Matrix< size, 1, T > & *m*) [inline]

The conversion constructor for conversion of a `Matrix` type of single column into type `ColumnVector`.

4.2.2.4 template<int size, class T = double> ColumnVector< size, T >::ColumnVector (const Vector< size, T > & *v*) [inline]

The conversion constructor for conversion of a `Vector` type into `ColumnVector`.

4.2.2.5 template<int size, class T = double> virtual ColumnVector< size, T >::~ColumnVector () [inline, virtual]

The default destructor.

4.2.3 Member Function Documentation

4.2.3.1 template<int size, class T = double> virtual T* ColumnVector< size, T >::getElementsPointer () const [inline, virtual]

Returns:
A pointer to the first element in the vector.

Implements VectorBase< T >.

4.2.3.2 template<int size, class T = double> virtual T ColumnVector< size, T >::getElement (int *index*) const [inline, virtual]

Returns:
The value at position specified by index (index = 1 is the first element).

Implements VectorBase< T >.

4.2.3.3 template<int size, class T = double> virtual void ColumnVector< size, T >::setElement (int *index*, T *value*) [inline, virtual]

Sets an element to a value at the specified position.

Parameters:
index Position of the desired element.
value The desired element is set to this value.

Implements VectorBase< T >.

4.2.3.4 template<int size, class T = double> virtual bool ColumnVector< size, T >::isRowVector () const [inline, virtual]

Returns:
false

Implements VectorBase< T >.

4.2.3.5 template<int size, class T = double> virtual int ColumnVector< size, T >::getNumElements () const [inline, virtual]

Returns:
The number of elements in the vector.

Implements VectorBase< T >.

4.2.3.6 template<int size, class T = double> T ColumnVector< size, T >::operator() (int *index*) const [inline]

4.2.3.7 template<int size, class T = double> T& ColumnVector< size, T >::operator() (int *index*) [inline]

Access or assign the element at the position specified by index. For example:

```
myVector(2)=12.65;
```

4.2.3.8 template<int size, class T = double> ColumnVector<size, T>& ColumnVector< size, T >::operator= (const VectorBase< T > & *v*) [inline]

Assign a VectorBase type to a ColumnVector type. Both objects must have the same dimensions.

Reimplemented from VectorBase< T >.

Reimplemented in Vector< size, T >.

4.2.3.9 template<int size, class T = double> MatrixInitializer<T> ColumnVector< size, T >::operator= (const T & *value*) [virtual]

Initialize a vector object.

Parameters:

value The value to which all elements in the vector are initialized. The initialization of the vector object can also be done as a comma seperated list. For example:

```
ColumnVector<3> myVector;
myVector = 67.88, 45.89, 90;
```

Implements VectorBase< T >.

Reimplemented in Vector< size, T >.

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

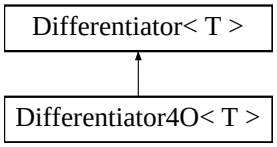
- ColumnVector.hpp

4.3 Differentiator< T > Class Template Reference

This is the base class for differentiators.

`#include <Differentiator.hpp>`

Inheritance diagram for Differentiator< T >::



Public Member Functions

- Differentiator (double period=0.001)
- virtual ~Differentiator ()
- void setSamplingPeriod (double period)
- void setCutOffFrequencyHz (double f)
- void setCutOffFrequencyRad (double cutOffFrequencyRad)
- void setDampingRatio (double d)
- void disableFilter ()
- void enableFilter ()
- void reset ()
- virtual T differentiate (const T &input)

4.3.1 Detailed Description

template<class T = double> class Differentiator< T >

This is the base class for differentiators.

This class implements numerical differentation using backward difference followed by low-pass filtering.

The Differentiator performs numerical differentiation of a signal using backward difference, followed by smoothening of the differentiated signal by a 2nd order butterworth low pass filter. The filtering is usually required due to the noisy nature of the result of numerical differentiation. Unfortunately low pass filter also introduces lag and innacuracy to the result. Filtering action should be disabled using disableFilter() if your input signal is sufficiently smooth.

A derived class from Differentiator can override the differenting algorithm provided.The class `Differentiator` can be used with many data types (double, int, RowVector, ColumnVector, Matrix, etc).

Example Program:

```
//====================================================================
// Package                      : The Math Library - Ex
// Authors                      : Vilas Kumar Chitrakaran
// Start Date           : Wed Dec 20 11:08:28 GMT 2000
// Compiler                     : GNU C++ 2.95.3 and above
// -------------------------------------------------------------------
// File: Differentiator.t.cpp
// Example program for the Differentiator.
```

```
//===========================================================================

//===========================================================================
//Differentiator.t.cpp
//---------------------------------------------------------------------------
// Demonstration of Differentiator class. The numerical differentiation
// result is compared with analytical solution.
//===========================================================================

#include "Differentiator.hpp"
#include "ColumnVector.hpp"
#include <stdio.h>

int main()
{
 FILE *outfile;  // This file holds the input and output waveforms.
 outfile = fopen("Differentiator.dat", "w+");

 double input;             // input signal
 double output_numerical;  // numerically computed derivative
 double output_actual;     // analytically computed derivative
 double error;             // error between numerical and analytical results
 double samplingPeriod;

 // Create Differentiator with a sampling period of 1 milli-second.
 samplingPeriod = 0.001;
 Differentiator<double> differentiator(samplingPeriod);

 // Set filter parameters.
 differentiator.setCutOffFrequencyHz(500);
 differentiator.setDampingRatio(1);
 differentiator.reset();

 fprintf(outfile, "%s\n%s %s %s %s\n", "%Differentiator output file",
        "%input", "output_actual", "output_numerical", "error" );
 for (int i = 0; i < 1.0/samplingPeriod; i++)
 {
  input = cos(2*M_PI*i*samplingPeriod); // 1 Hz signal

  // Differentiate analytically and numerically
  output_actual = -(2 * M_PI) * sin(2*M_PI*i*samplingPeriod);
  output_numerical = differentiator.differentiate(input);

  error = output_actual - output_numerical;

  // write the outputs to a file...
  fprintf(outfile, "%f %f %f %f\n", input, output_actual, output_numerical, error);
 }
 fclose(outfile);
 return(0);
}
```

4.3.2 Constructor & Destructor Documentation

4.3.2.1 template<class T = double> Differentiator< T >::Differentiator (double *period* = 0.001) [inline]

The Constructor initializes. The low pass filter is enabled by default, the cut-off frequency is set to half the sampling frequency and damping ratio of the low-pass filter in the differentiator is set to 1.

Parameters:

period The sampling period of the differentiator in seconds.

4.3.2.2 template<class T = double> virtual Differentiator< T >::~Differentiator () [inline, virtual]

The default destructor.

4.3.3 Member Function Documentation

4.3.3.1 template<class T = double> void Differentiator< T >::setSamplingPeriod (double *period*) [inline]

Sets the sampling period of the differentiator

4.3.3.2 template<class T = double> void Differentiator< T >::setCutOffFrequencyHz (double *f*) [inline]

Differentiation is followed by a low pass filtering process. This function sets the cut-off frequency of the filter in *hertz*.

4.3.3.3 template<class T = double> void Differentiator< T >::setCutOffFrequencyRad (double *cutOffFrequencyRad*) [inline]

Differentiation is followed by a low pass filtering process. This function sets the cut-off frequency of the filter in *rad/sec*.

4.3.3.4 template<class T = double> void Differentiator< T >::setDampingRatio (double *d*) [inline]

Sets the damping factor of the butterworth filter.

4.3.3.5 template<class T = double> void Differentiator< T >::disableFilter ()

Disable the low pass filtering after the differentiation. (Low pass filter is enabled by default.)

4.3.3.6 template<class T = double> void Differentiator< T >::enableFilter ()

Enable the low pass filtering after the differentiation.

4.3.3.7 template<class T = double> void Differentiator< T >::reset ()

Resets the differentiator output to zero.

Reimplemented in Differentiator4O< T >.

4.3.3.8 template<class T = double> virtual T Differentiator< T >::differentiate (const T & *input*) [virtual]

This function implements the numerical method for differentiation. The user can derive a different method of differentiation in a derived class. The differentiator output in the first cycle is smoothened to zero.

Parameters:

input The current value of the signal being differentiated.

Reimplemented in Differentiator4O< T >.

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

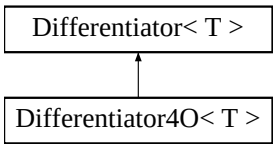
- Differentiator.hpp

4.4 Differentiator4O< T > Class Template Reference

Fourth order differentation followed by low-pass filtering.

`#include <Differentiator4O.hpp>`

Inheritance diagram for Differentiator4O< T >::



Public Member Functions

- Differentiator4O (double period=0.001)
- virtual ~Differentiator4O ()
- void reset ()
- virtual T differentiate (const T &input)

4.4.1 Detailed Description

template<class T = double> class Differentiator4O< T >

Fourth order differentation followed by low-pass filtering.

Example Program:

```
//====================================================================
// Package                      : The Math Library - Ex
// Authors                      : Vilas Kumar Chitrakaran
// Start Date           : Wed Dec 20 11:08:28 GMT 2000
// Compiler                     : GNU C++ 2.95.3 and above
// -------------------------------------------------------------------
// File: Differentiator4O.t.cpp
// Example program for the Differentiator.
//====================================================================

//====================================================================
//Differentiator4O.t.cpp
//--------------------------------------------------------------------
// Demonstration of Differentiator class. The numerical differentiation
// result is compared with analytical solution.
//====================================================================

#include "Differentiator4O.hpp"
#include "ColumnVector.hpp"
#include <stdio.h>

int main()
{
 FILE *outfile;  // This file holds the input and output waveforms.
 outfile = fopen("Differentiator4O.dat", "w+");

 double input;             // input signal
 double output_numerical;  // numerically computed derivative
 double output_actual;     // analytically computed derivative
```

```
 double error;              // error between numerical and analytical results
 double samplingPeriod;

 // Create Differentiator with a sampling period of 1 milli-second.
 samplingPeriod = 0.001;
 Differentiator4O<double> differentiator(samplingPeriod);

 // Set filter parameters.
 differentiator.setCutOffFrequencyHz(500);
 differentiator.setDampingRatio(1);
 differentiator.reset();

 fprintf(outfile, "%s\n%s %s %s %s\n", "%Differentiator output file",
         "%input", "output_actual", "output_numerical", "error" );
 for (int i = 0; i < 1.0/samplingPeriod; i++)
 {
  input = cos(2*M_PI*i*samplingPeriod); // 1 Hz signal

  // Differentiate analytically and numerically
  output_actual = -(2 * M_PI) * sin(2*M_PI*i*samplingPeriod);
  output_numerical = differentiator.differentiate(input);

  error = output_actual - output_numerical;

  // write the outputs to a file...
  fprintf(outfile, "%f %f %f %f\n", input, output_actual, output_numerical, error);
 }
 fclose(outfile);
 return(0);
}
```

4.4.2 Constructor & Destructor Documentation

4.4.2.1 template<class T = double> Differentiator4O< T >::Differentiator4O (double *period* = 0.001) [inline]

The Constructor initializes. The cut-off frequency is set to half the sampling frequency and damping ratio of the low-pass filter in the differentiator is set to 1.

Parameters:

period The sampling period of the differentiator in seconds.

4.4.2.2 template<class T = double> virtual Differentiator4O< T >::~Differentiator4O () [inline, virtual]

The default destructor.

4.4.3 Member Function Documentation

4.4.3.1 template<class T = double> void Differentiator4O< T >::reset ()

Resets the differentiator output to zero.

Reimplemented from Differentiator< T >.

4.4.3.2 template<class T = double> virtual T Differentiator4O< T >::differentiate (const T & *input*) [virtual]

This function implements the numerical method for differentiation. The output in the first cycle is smoothened to zero.

Parameters:

input The present value of the variable being differentiated.

Reimplemented from Differentiator< T >.

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

- Differentiator4O.hpp

4.5 HighpassFilter< T > Class Template Reference

A high-pass second order butterworth filter.

`#include <HighpassFilter.hpp>`

Public Member Functions

- HighpassFilter ()
- HighpassFilter (double hz, double period)
- ~HighpassFilter ()
- void setCutOffFrequencyHz (double hz)
- void setCutOffFrequencyRad (double rads)
- double getCutOffFrequencyHz ()
- double getCutOffFrequencyRad ()
- void setSamplingPeriod (double period)
- double getSamplingPeriod ()
- void setAutoInit ()
- void initializeFilter (const T &initInput, const T &initOutput)
- T filter (const T &input)

Protected Member Functions

- void calculateInternalParameters ()

Protected Attributes

- double d_samplingPeriod
- double d_cutOffFrequencyHz
- double d_cutOffFrequencyRad
- double d_numeratorParameter [3]
- double d_denumParameter [3]
- int d_initFlag
- T d_previousInputX [3]
- T d_previousOutputY [3]
- T d_filteredOut

4.5.1 Detailed Description

template<class T = double> class HighpassFilter< T >

A high-pass second order butterworth filter.

Example Program:

```
//==========================================================================
// Package                      : The Math Library - Ex
// Authors                      : Vilas Kumar Chitrakaran
// Start Date           : Wed Dec 20 11:08:28 GMT 2000
// Compiler                     : GNU C++ 2.95.3 and above
// --------------------------------------------------------------------------
```

```
// File: HighpassFilter.t.cpp
// Example program for the high-pass filter class.
//============================================================================

#include "HighpassFilter.hpp"
#include <math.h>
#include <stdio.h>

//============================================================================
// HighpassFilter.t.cpp
//----------------------------------------------------------------------------
// Demonstration of high-pass filtering of a signal with multiple
// frequency components to extract the high frequency component.
//============================================================================

int main()
{
 double wn = 600;              // cut-off freq. in Hz
 double dirtySignal;           // signal with multiple frequencies
 double filteredSignal;        // filtered signal
 FILE *outfile;                // file to store results
 double p = 0;

 // Define filter, with a sampling period as the second argument.
 HighpassFilter<double> myFilter(wn, 0.000142857);

 myFilter.initializeFilter(p, p);
 outfile = fopen("HighpassFilter.dat", "w+");

 fprintf(outfile, "%s\n%s %s\n", "%Highpass filter output file",
        "%dirty_signal", "filtered_signal" );
 for (int i=0; i<1000; i++)
 {
  // generate signal with 1Hz, 60Hz and 700Hz components
  p = ((double)i)/7000.0;
  dirtySignal = 2 +  sin(2*M_PI*p/10.0) + sin(60*2.0*M_PI*p) + sin(700*2.0*M_PI*p);

  // filter off low frequency compoments to extract 700Hz component
  filteredSignal = myFilter.filter(dirtySignal);

  //simply write the outputs to a file...
  fprintf(outfile, "%f %f\n", dirtySignal, filteredSignal);
 }

 fclose(outfile);

 return(0);
}
```

4.5.2 Constructor & Destructor Documentation

4.5.2.1 template<class T = double> HighpassFilter< T >::HighpassFilter () [inline]

The default Constructor. Sets cut-off frequency to 1 Hertz and sampling period to 0.001 sec.

4.5.2.2 template<class T = double> HighpassFilter< T >::HighpassFilter (double *hz*, double *period*) [inline]

This constructor initializes filter parameters.

Parameters:

hz The cut-off frequency in *hertz*.

period The sampling period in seconds.

4.5.2.3 template<class T = double> HighpassFilter< T >::~HighpassFilter () [inline]

The default destructor.

4.5.3 Member Function Documentation

4.5.3.1 template<class T = double> void HighpassFilter< T >::setCutOffFrequencyHz (double *hz*) [inline]

This function sets the cut-off frequency of the filter in *hertz*.

4.5.3.2 template<class T = double> void HighpassFilter< T >::setCutOffFrequencyRad (double *rads*) [inline]

This function sets the cut-off frequency of the filter in *rad/s*.

4.5.3.3 template<class T = double> double HighpassFilter< T >::getCutOffFrequencyHz () [inline]

Returns:

The cut-off frequency of the filter in *hertz*.

4.5.3.4 template<class T = double> double HighpassFilter< T >::getCutOffFrequencyRad () [inline]

Returns:

The cut-off frequency of the filter in rad/sec.

4.5.3.5 template<class T = double> void HighpassFilter< T >::setSamplingPeriod (double *period*) [inline]

Sets the sampling period of the filter.

4.5.3.6 template<class T = double> double HighpassFilter< T >::getSamplingPeriod () [inline]

Returns:

The sampling period in seconds.

4.5.3.7 template<class T = double> void HighpassFilter< T >::setAutoInit () [inline]

Automatic initialization of the filter.

4.5.3.8 template<class T = double> void HighpassFilter< T >::initializeFilter (const T & *initInput*, const T & *initOutput*) [inline]

Initializes the initial value of input and output.

Parameters:

initInput Initial value of the input to the filter.

initOutput Initial output of the filter.

4.5.3.9 template<class T = double> T HighpassFilter< T >::filter (const T & *input*) [inline]

The filter.

4.5.3.10 template<class T = double> void HighpassFilter< T >::calculateInternalParameters () [inline, protected]

Calculates the internel parameters based on cut-off frequency and sampling period.

4.5.4 Member Data Documentation

4.5.4.1 template<class T = double> double HighpassFilter< T >::d_samplingPeriod [protected]

4.5.4.2 template<class T = double> double HighpassFilter< T >::d_cutOffFrequencyHz [protected]

4.5.4.3 template<class T = double> double HighpassFilter< T >::d_cutOffFrequencyRad [protected]

4.5.4.4 template<class T = double> double HighpassFilter< T >::d_numeratorParameter[3] [protected]

4.5.4.5 template<class T = double> double HighpassFilter< T >::d_denumParameter[3] [protected]

4.5.4.6 template<class T = double> int HighpassFilter< T >::d_initFlag [protected]

4.5.4.7 template<class T = double> T HighpassFilter< T >::d_previousInputX[3] [protected]

4.5.4.8 template<class T = double> T HighpassFilter< T >::d_previousOutputY[3] [protected]

4.5.4.9 template<class T = double> T HighpassFilter< T >::d_filteredOut [protected]

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

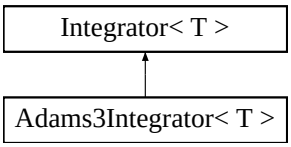
- HighpassFilter.hpp

4.6 Integrator< T > Class Template Reference

The base class for integrators.

`#include <Integrator.hpp>`

Inheritance diagram for Integrator< T >::



Public Member Functions

- Integrator ()
- Integrator (double period, const T &init)
- virtual ~Integrator ()
- void setSamplingPeriod (double period)
- virtual void reset (const T &init)
- virtual T integrate (const T &input)

4.6.1 Detailed Description

template<class T = double> class Integrator< T >

The base class for integrators.

This class implements the trapeziodal rule as the numerical method for integration. The user can reimplement a derived class with any other method of integration if desired. The class `Integrator` can be used with any data type (double, int, RowVector, ColumnVector, Matrix, etc).

Example Program:

```
//============================================================================
// Package                      : The Math Library - Ex
// Authors                      : Vilas Kumar Chitrakaran
// Start Date           : Wed Dec 20 11:08:28 GMT 2000
// Compiler                     : GNU C++ 2.95.3 and above
// ---------------------------------------------------------------------------
// File: Integrator.t.cpp
// Example program for the Integrator.
//============================================================================

//============================================================================
// Integrator.t.cpp
//----------------------------------------------------------------------------
// Numerically integrates a time varying function and compares result
// with analytical solution
//============================================================================

#include "Integrator.hpp"
#include "ColumnVector.hpp"

#include <stdio.h>
#include <math.h>
```

```
#ifndef M_PI
 #define M_PI 3.14159265358979323846
#endif

int main()
{
 FILE *outfile;                          // File to store results
 double velocity;                        // some data
 double position_numerical;              // numerical integral
 double position_actual;                 // actual integral
 double error;
 double initValue;                       // initial value of integration
 double sampling_period;                 // sampling period
 Integrator< double > myIntegrator;      // numerical integrator

 outfile = fopen("Integrator.dat", "w+");
 initValue = 0;
 sampling_period = 0.001;

 myIntegrator.setSamplingPeriod(sampling_period);
 myIntegrator.reset(initValue);

 fprintf(outfile, "%s\n%s %s %s %s\n", "%Integrator output file",
         "%velocity", "position_actual", "position_numerical", "error" );
 for (int i=0; i<1000; i++)
 {
  // input data
  velocity = sin(2*M_PI*i*sampling_period);

  // integrate
  position_numerical = myIntegrator.integrate(velocity);
  position_actual = 1.0/(2*M_PI) * (1 - cos(2*M_PI*i*sampling_period));

  error = position_actual - position_numerical;

  // simply write the outputs to a file...
  fprintf(outfile, "%f %f %f %f\n", velocity, position_actual, position_numerical, error);
 }

 fclose(outfile);
 return(0);
}
```

4.6.2 Constructor & Destructor Documentation

4.6.2.1 template<class T = double> Integrator< T >::Integrator () [inline]

The default constructor. The sampling period is set to default of 0.001 seconds. The initial output value is set to 0.

4.6.2.2 template<class T = double> Integrator< T >::Integrator (double *period*, const T & *init*) [inline]

The constructor to initialize the sampling period and initial Value.

Parameters:

period The sampling period in seconds.

init The initial value at the start of integration.

4.6.2.3 template<class T = double> virtual Integrator< T >::~Integrator () [inline, virtual]

The default destructor.

4.6.3 Member Function Documentation

4.6.3.1 template<class T = double> void Integrator< T >::setSamplingPeriod (double *period*) [inline]

Sets the sampling period of the integrator

4.6.3.2 template<class T = double> virtual void Integrator< T >::reset (const T & *init*) [virtual]

Reset the output of the Integrator.

Reimplemented in Adams3Integrator< T >.

4.6.3.3 template<class T = double> virtual T Integrator< T >::integrate (const T & *input*) [inline, virtual]

This function provides the numerical method for integration. The default is trapezoidal rule of integration. Override this method in a derived class to use another algorithm.

Parameters:

input The time-varying signal to be integrated.

Reimplemented in Adams3Integrator< T >.

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

- Integrator.hpp

4.7 LowpassFilter< T > Class Template Reference

A second order butterworth lowpass filter.

`#include <LowpassFilter.hpp>`

Public Member Functions

- LowpassFilter (double hz=1, double period=0.001, double damp=1)
- ~LowpassFilter ()
- void setCutOffFrequencyHz (double hz)
- void setCutOffFrequencyRad (double rads)
- double getCutOffFrequencyHz () const
- double getCutOffFrequencyRad () const
- void setDampingRatio (double damp)
- double getDampingRatio () const
- void setSamplingPeriod (double period)
- double getSamplingPeriod () const
- void setAutoInit ()
- void initializeFilter (T &initInput, T &initOutput)
- T filter (const T &input)

Protected Member Functions

- void calculateInternalParameters ()

Protected Attributes

- double d_samplingPeriod
- double d_cutOffFrequencyHz
- double d_cutOffFrequencyRad
- double d_dampingRatio
- double d_numeratorParameter [3]
- double d_denumParameter [3]
- int d_initFlag
- T d_previousInputX [3]
- T d_previousOutputY [3]
- T d_numerator
- T d_denumerator
- T d_filteredOut

4.7.1 Detailed Description

template<class T = double> class LowpassFilter< T >

A second order butterworth lowpass filter.

Example Program:

```
//============================================================================
// Package                      : The Math Library - Ex
// Authors                      : Vilas Kumar Chitrakaran
// Start Date           : Wed Dec 20 11:08:28 GMT 2000
// Compiler                     : GNU C++ 2.95.3 and above
// ---------------------------------------------------------------------------
// File: LowpassFilter.t.cpp
// Example program for the high-pass filter class.
//============================================================================


//============================================================================
// LowpassFilter.t.cpp
//----------------------------------------------------------------------------
// Demonstration of low-pass filtering of a noisy sine wave signal.
//============================================================================

#include "LowpassFilter.hpp"

#include <stdio.h>
#include <stdlib.h>
#include <math.h>



int main ()
{
 double wn = 3;            //cutoff frequency in hertz.
 double damp =1.0;         // damping ratio
 double dirtySignal;       // noisy signal
 double filteredSignal;    // filtered signal
 double p = 0;
 FILE *outfile;

 //define Filter
 LowpassFilter<double>  velocityFilter (wn, 0.001, damp);
 velocityFilter.initializeFilter(p, p);

 outfile = fopen("LowpassFilter.dat", "w+");

 fprintf(outfile, "%s\n%s %s\n", "%Lowpass filter output file",
        "%dirty_signal", "filtered_signal" );
 for (int i=0; i<1000; i++)
 {
  // generate sine wave corrupted by random noise
  p = ((double)i)/1000.0;
  dirtySignal = sin(2*M_PI*p) + ((double)rand()/(double)RAND_MAX) - 0.5;

  // filter the data
  filteredSignal = velocityFilter.filter(dirtySignal);

  // simply write the outputs to a file...
  fprintf(outfile, "%f %f\n", dirtySignal, filteredSignal);
 }

 fclose(outfile);
 return(0);
}
```

4.7.2 Constructor & Destructor Documentation

4.7.2.1 template<class T = double> LowpassFilter< T >::LowpassFilter (double *hz* = 1, double *period* = 0.001, double *damp* = 1)

This constructor for the filter initializes the parameters of the filter.

Parameters:

hz The cut-off frequency in *hertz* (default is 1).

period The sampling period in seconds (default is 1ms).

damp Desired damping ratio (default is 1).

4.7.2.2 template<class T = double> LowpassFilter< T >::~LowpassFilter () [inline]

The default destructor.

4.7.3 Member Function Documentation

4.7.3.1 template<class T = double> void LowpassFilter< T >::setCutOffFrequencyHz (double *hz*) [inline]

This function sets the cut-off frequency of the filter in *hertz*.

4.7.3.2 template<class T = double> void LowpassFilter< T >::setCutOffFrequencyRad (double *rads*) [inline]

This function sets the cut-off frequency of the filter in *rad/s*.

4.7.3.3 template<class T = double> double LowpassFilter< T >::getCutOffFrequencyHz () const [inline]

Returns:

The cut-off frequency of the filter in *hertz*.

4.7.3.4 template<class T = double> double LowpassFilter< T >::getCutOffFrequencyRad () const [inline]

Returns:

The cut-off frequency of the filter in *rad/s*.

4.7.3.5 template<class T = double> void LowpassFilter< T >::setDampingRatio (double *damp*) [inline]

Sets the damping factor of the butterworth filter.

4.7.3.6 template<class T = double> double LowpassFilter< T >::getDampingRatio () const [inline]

Returns:

The damping factor of the filter.

4.7.3.7 template<class T = double> void LowpassFilter< T >::setSamplingPeriod (double *period*) [inline]

Sets the sampling period of the low-pass filter.

4.7.3.8 template<class T = double> double LowpassFilter< T >::getSamplingPeriod () const [inline]

Returns:
The sampling period in seconds.

4.7.3.9 template<class T = double> void LowpassFilter< T >::setAutoInit () [inline]

Automatic initialization of the filter.

4.7.3.10 template<class T = double> void LowpassFilter< T >::initializeFilter (T & *initInput*, T & *initOutput*) [inline]

Initializes the initial value of input and output.

Parameters:
initInput Initial value of the input to the filter.

initOutput Initial output of the filter.

4.7.3.11 template<class T = double> T LowpassFilter< T >::filter (const T & *input*)

The filter.

4.7.3.12 template<class T = double> void LowpassFilter< T >::calculateInternalParameters () [protected]

Calculates the internel parameters based on user inputs of cut-off frequency, etc.

4.7.4 Member Data Documentation

4.7.4.1 template<class T = double> double LowpassFilter< T >::d_samplingPeriod [protected]

4.7.4.2 template<class T = double> double LowpassFilter< T >::d_cutOffFrequencyHz [protected]

4.7.4.3 template<class T = double> double LowpassFilter< T >::d_cutOffFrequencyRad [protected]

4.7.4.4 template<class T = double> double LowpassFilter< T >::d_dampingRatio [protected]

4.7.4.5 template<class T = double> double LowpassFilter< T >::d_numeratorParameter[3] [protected]

4.7.4.6 template<class T = double> double LowpassFilter< T >::d_denumParameter[3] [protected]

4.7.4.7 template<class T = double> int LowpassFilter< T >::d_initFlag [protected]

4.7.4.8 template<class T = double> T LowpassFilter< T >::d_previousInputX[3] [protected]

4.7.4.9 template<class T = double> T LowpassFilter< T >::d_previousOutputY[3] [protected]

4.7.4.10 template<class T = double> T LowpassFilter< T >::d_numerator [protected]

4.7.4.11 template<class T = double> T LowpassFilter< T >::d_denumerator [protected]

4.7.4.12 template<class T = double> T LowpassFilter< T >::d_filteredOut [protected]

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

- LowpassFilter.hpp

4.8 MathException Class Reference

Run-time exception handling for the math library.

`#include <MathException.hpp>`

Public Member Functions

- MathException ()
- ~MathException ()
- const char * getErrorMessage () const
- QMathException_t getErrorType () const
- bool isErrorType (QMathException_t error)
- void setErrorType (QMathException_t error)

4.8.1 Detailed Description

Run-time exception handling for the math library.

The error type is internally set by the library.

Example Program:

```
//===========================================================================
// Package              : The Math Library - Ex
// Authors              : Vilas Kumar Chitrakaran
// Start Date           : Wed Dec 20 11:08:28 GMT 2000
// Compiler             : GNU C++ 2.95.3 and above
// ---------------------------------------------------------------------------
// File: MathException.t.cpp
// Example program for the class MathExceptions.
//===========================================================================

#include "Matrix.hpp"

using namespace std;

int main()
{

 Matrix<2,2> m1, m2;

 m1 = 1.0, 5.6, 2.7, 8.4;

 // Enclose critical code inside try block.
 //subsequent catch block catches exceptions
 try
 {
  m2 = m1/0.0; /* divide by zero! */
 }
 catch (MathException &ex)
 {
  cout << ex.getErrorMessage() << endl;
  /* do exception recovery here */
  return -1;
 }
 cout << "This line won't print" << endl;
 return 0;
}
```

4.8.2 Constructor & Destructor Documentation

4.8.2.1 MathException::MathException () [inline]

4.8.2.2 MathException::~MathException () [inline]

4.8.3 Member Function Documentation

4.8.3.1 const char* MathException::getErrorMessage () const [inline]

4.8.3.2 QMathException_t MathException::getErrorType () const [inline]

4.8.3.3 bool MathException::isErrorType (QMathException_t *error*) [inline]

4.8.3.4 void MathException::setErrorType (QMathException_t *error*) [inline]

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

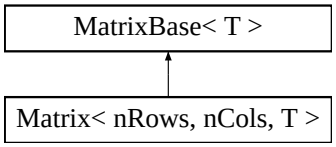
- MathException.hpp

4.9 Matrix< nRows, nCols, T > Class Template Reference

Methods for mathematical operations on matrices.

`#include <Matrix.hpp>`

Inheritance diagram for Matrix< nRows, nCols, T >::



Public Member Functions

- Matrix ()
- Matrix (const Matrix< nRows, nCols, T > &matrix)
- virtual ~Matrix ()
- virtual T * getElementsPointer () const
- virtual int getNumRows () const
- virtual int getNumColumns () const
- ColumnVector< nRows, T > getColumn (int c) const
- RowVector< nCols, T > getRow (int r) const
- virtual T getElement (int r, int c) const
- virtual void setElement (int r, int c, T val)
- template<int sr, int sc, class X> void getSubMatrix (int pivotRow, int pivotColumn, Matrix< sr, sc, X > &m) const
- template<int sr, int sc, class X> void setSubMatrix (int pivotRow, int pivotColumn, const Matrix< sr, sc, X > &m)
- T operator() (int r, int c) const
- T & operator() (int r, int c)
- MatrixInitializer< T > operator= (const T &val)
- Matrix & operator= (const MatrixBase< T > &m)
- Matrix & operator+= (const Matrix< nRows, nCols, T > &rhs)
- Matrix & operator-= (const Matrix< nRows, nCols, T > &rhs)
- Matrix & operator *= (const T &scalar)
- Matrix & operator/= (const T &scalar)

Protected Member Functions

- template<int nCols, int nRows, class T> Matrix< nCols, nRows, T > transpose (const Matrix< nRows, nCols, T > &matrix)
- template<int size, class T> Matrix< size, size, T > inverse (const Matrix< size, size, T > &m)
- template<int size, class T> T determinant (const Matrix< size, size, T > &matrix)
- template<int size, class T> T trace (const Matrix< size, size, T > &matrix)
- template<int size, class T> Matrix< size, size, T > unitMatrix ()
- template<int r1, int c1r2, int c2, class T> Matrix< r1, c2, T > operator * (const Matrix< r1, c1r2, T > &m1, const Matrix< c1r2, c2, T > &m2)
- template<int c1r2, class T> T operator * (const Matrix< 1, c1r2, T > &m1, const Matrix< c1r2, 1, T > &m2)

Protected Attributes

- T d_element [nRows ∗nCols]
- int d_size

Friends

- Matrix operator+ (const Matrix< nRows, nCols, T > &lhs, const Matrix< nRows, nCols, T > &rhs)
- Matrix operator- (const Matrix< nRows, nCols, T > &lhs, const Matrix< nRows, nCols, T > &rhs)
- Matrix operator ∗ (const Matrix< nRows, nCols, T > &lhs, const T &scalar)
- Matrix operator ∗ (const T &s, const Matrix< nRows, nCols, T > &rhs)
- Matrix operator/ (const Matrix< nRows, nCols, T > &lhs, const T &scalar)
- std::ostream & operator<< (std::ostream &output, const Matrix< nRows, nCols, T > &matrix)
- std::istream & operator>> (std::istream &input, Matrix< nRows, nCols, T > &matrix)
- bool operator== (const Matrix< nRows, nCols, T > &lhs, const Matrix< nRows, nCols, T > &rhs)
- bool operator!= (const Matrix< nRows, nCols, T > &lhs, const Matrix< nRows, nCols, T > &rhs)

4.9.1 Detailed Description

template<int nRows, int nCols, class T = double> class Matrix< nRows, nCols, T >

Methods for mathematical operations on matrices.

The class `Matrix` is derived from its base class `MatrixBase`.

This class provides common mathematical functions for matrices such as addition, multipication and subtraction between matrices, along with methods to get/set elements/sub-matrices. The template class also provides methods for determination of the inverse of a matrix upto 4 x 4, the transpose of a matrix and generation of unit matrices. The classes `ColumnVector`, `RowVector` and `Transform` are derived from this class.

Example Program:

```
//===========================================================================
// Package              : The Math Library - Ex
// Authors              : Vilas Kumar Chitrakaran
// Start Date    : Wed Dec 20 11:08:28 GMT 2000
// Compiler             : GNU C++ 2.95.3 and above
// ---------------------------------------------------------------------
// File: Matrix.t.cpp
// Example program for the class Matrix.
//===========================================================================

#include "Matrix.hpp"
#include "ColumnVector.hpp"
#include "RowVector.hpp"

using namespace std;

//===========================================================================
// This example demonstrates solving the foll. simultaneous eqns
// 2 * x1 + 8 * x2 + 5 * x3 = 5,
// 1 * x1 + 1 * x2 + 1 * x3 = -2,
// 1 * x1 + 2 * x2 - 1 * x3 = 2.
//===========================================================================
int main()
{
 Matrix<3,3> A;
```

```
 ColumnVector<3> x;
 ColumnVector<3> b;

 // Write in Ax = b form
 A = 2, 8, 5,
     1, 1, 1,
     1, 2, -1;
 b = 5, -2, 2;

 // solve for x
 x = inverse(A) * b;
 cout << "solution: " << transpose(x) << endl;

 return 0;
}
```

4.9.2 Constructor & Destructor Documentation

4.9.2.1 template<int nRows, int nCols, class T = double> Matrix< nRows, nCols, T >::Matrix () [inline]

4.9.2.2 template<int nRows, int nCols, class T = double> Matrix< nRows, nCols, T >::Matrix (const Matrix< nRows, nCols, T > & *matrix*) [inline]

The default constructor for the Matrix object. The elements of Matrix are not initialized. Copy Constructor.

4.9.2.3 template<int nRows, int nCols, class T = double> virtual Matrix< nRows, nCols, T >::~Matrix () [inline, virtual]

The default destructor of the Matrix object.

4.9.3 Member Function Documentation

4.9.3.1 template<int nRows, int nCols, class T = double> virtual T* Matrix< nRows, nCols, T >::getElementsPointer () const [inline, virtual]

Returns:
The pointer to the first element in the Matrix.

Implements MatrixBase< T >.

Reimplemented in ColumnVector< size, T >, and RowVector< size, T >.

4.9.3.2 template<int nRows, int nCols, class T = double> virtual int Matrix< nRows, nCols, T >::getNumRows () const [inline, virtual]

Returns:
The number of rows in the Matrix.

Implements MatrixBase< T >.

4.9.3.3 template<int nRows, int nCols, class T = double> virtual int Matrix< nRows, nCols, T >::getNumColumns () const [inline, virtual]

Returns:
The number of columns in the Matrix.

Implements MatrixBase< T >.

4.9.3.4 template<int nRows, int nCols, class T = double> ColumnVector<nRows, T> Matrix< nRows, nCols, T >::getColumn (int *c*) const [inline]

Returns:
The column specified by *c*. Example:

```
ColumnVector<3> c;
Matrix<3,3> m;
c = m.getColumn(1); // extract first column
```

4.9.3.5 template<int nRows, int nCols, class T = double> RowVector<nCols, T> Matrix< nRows, nCols, T >::getRow (int *r*) const [inline]

Returns:
The row specified by *r*. Example: see getColumn()

4.9.3.6 template<int nRows, int nCols, class T = double> virtual T Matrix< nRows, nCols, T >::getElement (int *r*, int *c*) const [inline, virtual]

returns the element at the specified position.

Parameters:
r Row number of the desired element.

c Column number of the desired element.

Implements MatrixBase< T >.

4.9.3.7 template<int nRows, int nCols, class T = double> virtual void Matrix< nRows, nCols, T >::setElement (int *r*, int *c*, T *val*) [inline, virtual]

Sets an element to a value at the specified position.

Parameters:
r Row number of the desired element.

c Column number of the desired element.

val The desired element is set to this value.

Implements MatrixBase< T >.

4.9.3.8 template<int nRows, int nCols, class T = double> template<int sr, int sc, class X> void Matrix< nRows, nCols, T >::getSubMatrix (int *pivotRow*, int *pivotColumn*, Matrix< sr, sc, X > & *m*) const

This function extracts a sub-matrix of the size of *m* (sr x sc) starting from the specified pivotal row and column (that defines the top left corner of sub matrix).

Parameters:

pivotRow,pivotColumn The position of the first element of the sub-matrix in the matrix from which it is extracted.

m The extracted sub matrix. Example:

```
Matrix<2,2> s;
Matrix<4,4> m;
//extract s = [m(1,3), m(1,4); m(2,3), m(2,4)]
m.getSubMatrix(1,3,s);
```

4.9.3.9 template<int nRows, int nCols, class T = double> template<int sr, int sc, class X> void Matrix< nRows, nCols, T >::setSubMatrix (int *pivotRow*, int *pivotColumn*, const Matrix< sr, sc, X > & *m*)

This function sets a sub-matrix of the size of *m* (sr x sc) starting from the pivotal row and column within the matrix.

Parameters:

pivotRow,pivotColumn The position of the first element of the sub-matrix in the matrix to which it is extracted.

m The sub-matrix. Example:

```
Matrix<2,2> s;
Matrix<3,2> m;
//set m = [m(1,1), s(1,1), s(1,2); m(2,1), s(2,1), s(2,2)]
m.getSubMatrix(1,2,s);
```

4.9.3.10 template<int nRows, int nCols, class T = double> T Matrix< nRows, nCols, T >::operator() (int *r*, int *c*) const [inline]

4.9.3.11 template<int nRows, int nCols, class T = double> T& Matrix< nRows, nCols, T >::operator() (int *r*, int *c*) [inline]

Access or assign the element at *r* row and *c* column of the matrix. Example:

```
myMatrix(2,3)=22.2;
```

4.9.3.12 template<int nRows, int nCols, class T = double> MatrixInitializer<T> Matrix< nRows, nCols, T >::operator= (const T & *val*) [inline, virtual]

Assignment operator for initializing a Matrix object.

Parameters:

val This is the value to which all elements in the matrix are initialized. The initialization of the Matrix object can also be done as a comma seperated list. For example:

```
Matrix<2,2> myMatrix;
   myMatrix = 67.899, 23.45, 6, 98;
```

Implements MatrixBase< T >.

Reimplemented in ColumnVector< size, T >, RowVector< size, T >, Transform, and Vector< size, T >.

4.9.3.13 template<int nRows, int nCols, class T = double> Matrix& Matrix< nRows, nCols, T >::operator= (const MatrixBase< T > & *m*) [inline]

Assign a *MatrixBase* type to a *Matrix* type. The dimensions of both the objects must be the same.

Parameters:
m The object of the base class *MatrixBase*.

Reimplemented from MatrixBase< T >.

4.9.3.14 template<int nRows, int nCols, class T = double> Matrix& Matrix< nRows, nCols, T >::operator+= (const Matrix< nRows, nCols, T > & *rhs*) [inline]

Matrix addition and assignment operator.

Parameters:
rhs The right hand side Matrix.

Returns:
Assign matrix to the sum of itself and *rhs* matrix. Example:

```
Matrix<2,2> m1, m2;
m1 += m2;
```

4.9.3.15 template<int nRows, int nCols, class T = double> Matrix& Matrix< nRows, nCols, T >::operator-= (const Matrix< nRows, nCols, T > & *rhs*) [inline]

Matrix subtraction and assignment operator.

Parameters:
rhs The right hand side Matrix.

Returns:
Assign matrix to the difference of itself and *rhs* matrix. Example:

```
Matrix<2,2> m1, m2;
m1 -= m2;
```

4.9.3.16 template<int nRows, int nCols, class T = double> Matrix& Matrix< nRows, nCols, T >::operator *= (const T & *scalar*) [inline]

Matrix product with scalar and assignment operator.

Parameters:
scalar The scalar value to be multiplied with the matrix.

Returns:
Assign matrix to the product of itself and *scalar*. Example:

```
Matrix<2,2> m1;
double s;
m1 *= s;
```

4.9.3.17 template<int nRows, int nCols, class T = double> Matrix& Matrix< nRows, nCols, T >::operator/= (const T & *scalar*) [inline]

Matrix division with scalar and assignment operator.

Parameters:
scalar The scalar value to divide the matrix by.

Returns:
Assign matrix to the product of itself and *scalar*. Example:

```
Matrix<2,2> m1;
double s;
m1 /= s;
```

4.9.3.18 template<int nRows, int nCols, class T = double> template<int nCols, int nRows, class T> Matrix<nCols, nRows, T> Matrix< nRows, nCols, T >::transpose (const Matrix< nRows, nCols, T > & *matrix*) [protected]

Additional template functions

Returns:
The transpose of the matrix *matrix*.

4.9.3.19 template<int nRows, int nCols, class T = double> template<int size, class T> Matrix<size, size, T> Matrix< nRows, nCols, T >::inverse (const Matrix< size, size, T > & *m*) [protected]

Inverse of a square matrix *m*. This function can be used to compute inverses of matrices of size 2x2, 3x3, 4x4, 6x6, and 8x8. This library does not support inverse operation on matrices of any other dimensions.

Returns:
inverse of the matrix m.

4.9.3.20 template<int nRows, int nCols, class T = double> template<int size, class T> T Matrix< nRows, nCols, T >::determinant (const Matrix< size, size, T > & *matrix*) [protected]

Returns:
Determinant of a matrix.

4.9.3.21 template<int nRows, int nCols, class T = double> template<int size, class T> T Matrix< nRows, nCols, T >::trace (const Matrix< size, size, T > & *matrix*) [protected]

Returns:
Trace of a size x size matrix.

4.9.3.22 template<int nRows, int nCols, class T = double> template<int size, class T> Matrix<size, size, T> Matrix< nRows, nCols, T >::unitMatrix () [protected]

Generate a unit matrix of size x size. Example:

```
Matrix<3,3> A;
 A=unitMatrix<3>();
```

4.9.3.23 template<int nRows, int nCols, class T = double> template<int r1, int c1r2, int c2, class T> Matrix<r1, c2, T> Matrix< nRows, nCols, T >::operator * (const Matrix< r1, c1r2, T > & *m1*, const Matrix< c1r2, c2, T > & *m2*) [protected]

Matrix multiplication.

Returns:
The product of *m1* and *m2*. Example:

```
Matrix<3,2> m1, m2, m3;
m1 = m2 * m3;
```

4.9.3.24 template<int nRows, int nCols, class T = double> template<int c1r2, class T> T Matrix< nRows, nCols, T >::operator * (const Matrix< 1, c1r2, T > & *m1*, const Matrix< c1r2, 1, T > & *m2*) [protected]

Multiplication between a row Matrix object and a column Matrix.

Returns:
The scalar product of row matrix *m1* and column matrix *m2*.

4.9.4 Friends And Related Function Documentation

4.9.4.1 template<int nRows, int nCols, class T = double> Matrix operator+ (const Matrix< nRows, nCols, T > & *lhs*, const Matrix< nRows, nCols, T > & *rhs*) [friend]

Matrix addition operator.

Parameters:
lhs Left hand side matrix

rhs The right hand side Matrix.

Returns:
The sum of *rhs* and the matrix to the left hand side of the addition operator. Example:

```
Matrix<2,2> m1, m2, m3;
m1 = m2 + m3;
```

4.9.4.2 template<int nRows, int nCols, class T = double> Matrix operator- (const Matrix< nRows, nCols, T > & *lhs*, const Matrix< nRows, nCols, T > & *rhs*) [friend]

Matrix difference operator.

Parameters:

lhs Left hand side matrix

rhs The right hand side Matrix.

Returns:

The matrix after subtracting *rhs* matrix from the matrix on the left hand side of the difference operator. Example: see operator+()

4.9.4.3 template<int nRows, int nCols, class T = double> Matrix operator * (const Matrix< nRows, nCols, T > & *lhs*, const T & *scalar*) [friend]

Post-multiplication of a matrix with a scalar.

Parameters:

lhs Left hand side matrix

scalar The scalar value to be multiplied with the matrix.

Returns:

The product of *scalar* and the matrix. Example:

```
double s;
Matrix<3,2> m1, m2;
m1 = m2 * s;
```

4.9.4.4 template<int nRows, int nCols, class T = double> Matrix operator * (const T & *s*, const Matrix< nRows, nCols, T > & *rhs*) [friend]

Pre-multiplication of a matrix with scalar.

Returns:

The product of *scalar* and *matrix*. Example:

```
double s;
Matrix<3,2> m1, m2;
m1 = s * m2;
```

4.9.4.5 template<int nRows, int nCols, class T = double> Matrix operator/ (const Matrix< nRows, nCols, T > & *lhs*, const T & *scalar*) [friend]

Division of a matrix by a scalar.

Parameters:

lhs Left hand side matrix

scalar The scalar value to divide the Matrix by.

Returns:
The matrix with each element divided by the *scalar*. Example:

```
double s;
Matrix<3,2> m1, m2;
m1 = m2 / s;
```

4.9.4.6 template<int nRows, int nCols, class T = double> std::ostream& operator<< (std::ostream & *output*, const Matrix< nRows, nCols, T > & *matrix*) [friend]

This function overloads the ostream << operator to output the elements of the matrix *matrix* row-wise to the output stream separated by white spaces(e.g. spaces). Example:

```
cout << matrix;
```

4.9.4.7 template<int nRows, int nCols, class T = double> std::istream& operator>> (std::istream & *input*, Matrix< nRows, nCols, T > & *matrix*) [friend]

This function overloads the istream >> operator to read the elements of the matrix *matrix* from an input stream. The elements must be arranged row-wise in the input stream, separated by white spaces (e.g. spaces, tabs, etc). Example:

```
cin >> matrix;
```

4.9.4.8 template<int nRows, int nCols, class T = double> bool operator== (const Matrix< nRows, nCols, T > & *lhs*, const Matrix< nRows, nCols, T > & *rhs*) [friend]

Returns:
'true' if the *lhs* matrix is same as the *rhs* matrix, else FALSE.

4.9.4.9 template<int nRows, int nCols, class T = double> bool operator!= (const Matrix< nRows, nCols, T > & *lhs*, const Matrix< nRows, nCols, T > & *rhs*) [friend]

Returns:
'true' if the *lhs* matrix is not the same as the *rhs* matrix, else FALSE.

4.9.5 Member Data Documentation

4.9.5.1 template<int nRows, int nCols, class T = double> T Matrix< nRows, nCols, T >::d_element[nRows *nCols] [protected]

4.9.5.2 template<int nRows, int nCols, class T = double> int Matrix< nRows, nCols, T >::d_size [protected]

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

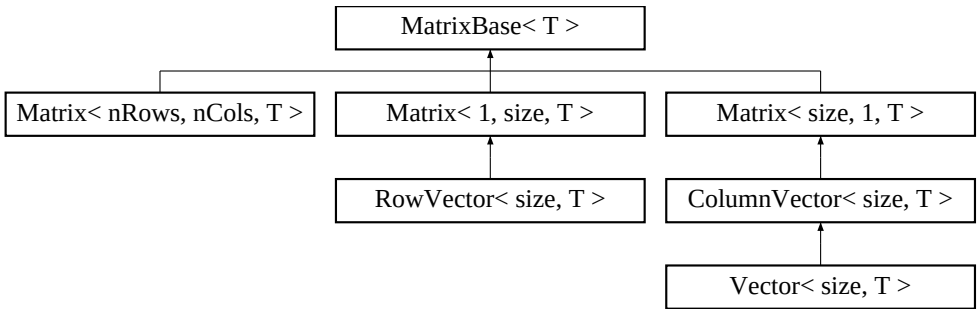
- Matrix.hpp

4.10 MatrixBase< T > Class Template Reference

This is a pure virtual base class for Matrix.

`#include <MatrixBase.hpp>`

Inheritance diagram for MatrixBase< T >::



Public Member Functions

- MatrixBase ()
- virtual ~MatrixBase ()
- virtual T * getElementsPointer () const =0
- virtual int getNumRows () const =0
- virtual int getNumColumns () const =0
- virtual T getElement (int row, int column) const =0
- virtual void setElement (int row, int column, T value)=0
- MatrixBase< T > & operator= (const MatrixBase< T > &m)
- virtual MatrixInitializer< T > operator= (const T &value)=0
- void output (std::ostream &outputStream=std::cout)

Protected Member Functions

- MatrixBase (const MatrixBase< T > &m)

4.10.1 Detailed Description

template<class T = double> class MatrixBase< T >

This is a pure virtual base class for Matrix.

4.10.2 Constructor & Destructor Documentation

4.10.2.1 template<class T = double> MatrixBase< T >::MatrixBase () [inline]

The default constructor.

4.10.2.2 template<class T = double> virtual MatrixBase< T >::~MatrixBase () [inline, virtual]

4.10.2.3 template<class T = double> MatrixBase< T >::MatrixBase (const MatrixBase< T > & *m*) [inline, protected]

4.10.3 Member Function Documentation

4.10.3.1 template<class T = double> virtual T* MatrixBase< T >::getElementsPointer () const [pure virtual]

The default destructor.

Returns:
The pointer to the first element in a matrix or vector.

Implemented in ColumnVector< size, T >, Matrix< nRows, nCols, T >, RowVector< size, T >, Matrix< size, 1, T >, Matrix< 1, size, T >, and Matrix< 4, 4, double >.

4.10.3.2 template<class T = double> virtual int MatrixBase< T >::getNumRows () const [pure virtual]

Returns:
The number of rows in the Matrix.

Implemented in Matrix< nRows, nCols, T >, Matrix< size, 1, T >, Matrix< 1, size, T >, and Matrix< 4, 4, double >.

4.10.3.3 template<class T = double> virtual int MatrixBase< T >::getNumColumns () const [pure virtual]

Returns:
The number of columns in the Matrix.

Implemented in Matrix< nRows, nCols, T >, Matrix< size, 1, T >, Matrix< 1, size, T >, and Matrix< 4, 4, double >.

4.10.3.4 template<class T = double> virtual T MatrixBase< T >::getElement (int *row*, int *column*) const [pure virtual]

Returns:
The element at the specified position.

Implemented in Matrix< nRows, nCols, T >, Matrix< size, 1, T >, Matrix< 1, size, T >, and Matrix< 4, 4, double >.

4.10.3.5 template<class T = double> virtual void MatrixBase< T >::setElement (int *row*, int *column*, T *value*) [pure virtual]

Sets an element to a value at the specified position.

Parameters:

row Row number of the desired element.

column Column number of the desired element.

value The desired element is set to this value.

Implemented in Matrix< nRows, nCols, T >, Matrix< size, 1, T >, Matrix< 1, size, T >, and Matrix< 4, 4, double >.

4.10.3.6 template<class T = double> MatrixBase<T>& MatrixBase< T >::operator= (const MatrixBase< T > & *m*)

Assignment operator between two `MatrixBase` types of same dimensions.

Reimplemented in Matrix< nRows, nCols, T >, Matrix< size, 1, T >, Matrix< 1, size, T >, and Matrix< 4, 4, double >.

4.10.3.7 template<class T = double> virtual MatrixInitializer<T> MatrixBase< T >::operator= (const T & *value*) [pure virtual]

Initialization of matrix.

Parameters:

value The value to which all elements in the matrix are initialized.

Implemented in ColumnVector< size, T >, Matrix< nRows, nCols, T >, RowVector< size, T >, Transform, Vector< size, T >, Matrix< size, 1, T >, Matrix< 1, size, T >, and Matrix< 4, 4, double >.

4.10.3.8 template<class T = double> void MatrixBase< T >::output (std::ostream & *outputStream* = `std::cout`)

Returns:

The elements in the referenced matrix to the output stream (by default the output is to the console)

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

- MatrixBase.hpp

4.11 MatrixInitializer< T > Class Template Reference

This class is used internally by the library to initialize the Matrix and its derived class objects.

`#include <MatrixInitializer.hpp>`

Public Member Functions

- MatrixInitializer (T num, int offset, T *firstElementPointer)
- ~MatrixInitializer ()
- MatrixInitializer< T > operator, (const T &elementValue)

4.11.1 Detailed Description

template<class T = double> class MatrixInitializer< T >

This class is used internally by the library to initialize the Matrix and its derived class objects.

4.11.2 Constructor & Destructor Documentation

4.11.2.1 template<class T = double> MatrixInitializer< T >::MatrixInitializer (T *num*, int *offset*, T * *firstElementPointer*) [inline]

The default constructor.

4.11.2.2 template<class T = double> MatrixInitializer< T >::~MatrixInitializer () [inline]

The default destructor.

4.11.3 Member Function Documentation

4.11.3.1 template<class T = double> MatrixInitializer<T> MatrixInitializer< T >::operator, (const T & *elementValue*) [inline]

This function provides a method to easily assign the elements of a Matrix object. A Matrix object is initialized in the following manner:

```
Matrix<2,2> myMatrix;
   myMatrix = 2.0, 5.0, 78.90, 20;
```

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

- MatrixInitializer.hpp

4.12 ODESolverRK4< T > Class Template Reference

Solver for ordinary differential equations using 4th order Runge Kutta method.

`#include <ODESolverRK4.hpp>`

Public Member Functions

- ODESolverRK4 ()
- ODESolverRK4 (double period, const T &init)
- virtual ~ODESolverRK4 ()
- void setSamplingPeriod (double period)
- void setODE (T(*diffFunc)(T &, double t))
- virtual void reset (const T &init)
- virtual T stepSolve ()

4.12.1 Detailed Description

template<class T = double> class ODESolverRK4< T >

Solver for ordinary differential equations using 4th order Runge Kutta method.

Use this class for solving differential equations of the type x'(t) = f(x,t). The user must provide a function that returns x'(t) for any given x(t) and t.

Example Program:

```
//============================================================================
// Package                      : The Math Library - Ex
// Authors                      : Vilas Kumar Chitrakaran
// Start Date           : Wed Oct 28 11:08:28 GMT 2004
// Compiler                     : GNU C++ 2.95.3 and above
// ---------------------------------------------------------------------------
// File: ODESolverRK4.t.cpp
// Example program for the ODESolverRK4 integrator.
//============================================================================

#include <stdio.h>
#include <math.h>
#include "ODESolverRK4.hpp"
#include "Vector.hpp"


//============================================================================
// This program computes solution of a 2nd order differential equation
// numerically and compares result with analytical solution.
//============================================================================


//============================================================================
// Physical system - an mass-spring-damper.
// This function implements physical system as x' = f(x,t)
//============================================================================
Vector<2> system(Vector<2> &y, double t)
{
 Vector<2> ydot;
 t = t;
 double M = 1.0;  // mass (kg)
 double K = 10.0; // spring stiffness coefficient (N/m)
```

```
 double f = 2.0;  // viscous friction coefficient (Ns/m)

 // system: mx'' + fx' + Kx = 0.
 // Write in state space form as
 // [y1dot; y2dot] = [-f/M, -K/M; 1, 0].[y1; y2]

 ydot(1) = -(f/M) * y(1)  - (K/M) * y(2);
 ydot(2) = y(1);

 return ydot;
}


//========================================================================
// main function
//========================================================================
int main()
{
 FILE *outfile;
 double dt = 0.001;                // sampling time (s)
 Vector<2> y;                      // state vector ([velocity; position])
 Vector<2> y_next_numerical;       // numerical soln at next time step
 double y_next_analytical;         // analytical soln at next time step
 ODESolverRK4< Vector<2> > solver; // numerical solver

 y = 2.0, 4.0; // initial velocity and position

 // set up the solver
 solver.setODE(system);
 solver.setSamplingPeriod(dt);
 solver.reset(y);

 outfile = fopen("ODESolverRK4.dat", "w+");

 // Call integrate() every time-step
 double t = 0;
 double error;

 y_next_numerical = y;
 y_next_analytical = y(2);
 fprintf(outfile, "%s\n%s %s %s\n", "%ODE Solver RK4 output file",
         "%analytical_output", "numerical_output", "error" );
 for(t = dt; t < 10.0; t +=dt)
 {
  // find error between analytical and numerical soln.
  error = y_next_analytical - y_next_numerical(2);

  // dump outputs in file
  fprintf(outfile, "%f %f %f\n", y_next_analytical, y_next_numerical(2), error);

  // analytical (actual) solution
  y_next_analytical = exp(-t)*(4*cos(3 * t) + 2 * sin(3 * t));

  // numerical solution
  y_next_numerical = solver.stepSolve();
 }

 fclose(outfile);
 return(0);
}
```

4.12.2 Constructor & Destructor Documentation

4.12.2.1 template<class T = double> ODESolverRK4< T >::ODESolverRK4 ()

The default constructor. The sampling period is set to default of 0.001 seconds The initial value is set to 0.

4.12.2.2 template<class T = double> ODESolverRK4< T >::ODESolverRK4 (double *period*, const T & *init*)

This constructor initializes the sampling period and initial Value.

Parameters:

period The sampling period in seconds.

init The initial value.

4.12.2.3 template<class T = double> virtual ODESolverRK4< T >::~ODESolverRK4 () [inline, virtual]

4.12.3 Member Function Documentation

4.12.3.1 template<class T = double> void ODESolverRK4< T >::setSamplingPeriod (double *period*) [inline]

The default destructor Sets the sampling period of the solver

4.12.3.2 template<class T = double> void ODESolverRK4< T >::setODE (T(∗)(T &, double t) *diffFunc*)

Set the differential equation of the form x'=f(x,t) to evaluate.

Parameters:

diffFunc The differential equation

4.12.3.3 template<class T = double> virtual void ODESolverRK4< T >::reset (const T & *init*) [virtual]

This function resets the output of the solver to *init* and further integration restarts from this initial value.

4.12.3.4 template<class T = double> virtual T ODESolverRK4< T >::stepSolve () [inline, virtual]

Find solution for the next time step.

Returns:

The integrated value for the next step

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

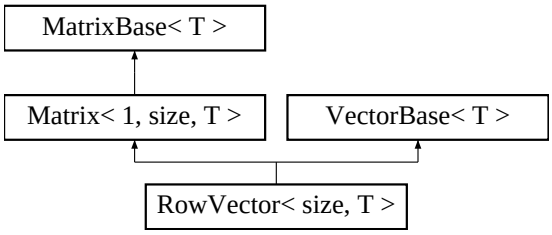
- ODESolverRK4.hpp

4.13 RowVector< size, T > Class Template Reference

A class for row vectors.

`#include <RowVector.hpp>`

Inheritance diagram for RowVector< size, T >::



Public Member Functions

- RowVector ()
- RowVector (const RowVector< size, T > &rowVector)
- RowVector (const Matrix< 1, size, T > &matrix)
- ~RowVector ()
- virtual T * getElementsPointer () const
- virtual T getElement (int index) const
- virtual void setElement (int index, T value)
- virtual bool isRowVector () const
- virtual int getNumElements () const
- T operator() (int index) const
- T & operator() (int index)
- RowVector< size, T > & operator= (const VectorBase< T > &vectorBase)
- MatrixInitializer< T > operator= (const T &value)

4.13.1 Detailed Description

template<int size, class T = double> class RowVector< size, T >

A class for row vectors.

The class `RowVector` is derived from the base classes `Matrix` and `VectorBase`, and provides methods for operations such as cross product, dot product and element-by-element multiplication.

Example Program: See the example of the class `ColumnVector`.

4.13.2 Constructor & Destructor Documentation

4.13.2.1 template<int size, class T = double> RowVector< size, T >::RowVector () [inline]

The default constructor. The elements are not initialized.

4.13.2.2 template<int size, class T = double> RowVector< size, T >::RowVector (const RowVector< size, T > & *rowVector*) [inline]

Copy Constructor.

4.13.2.3 template<int size, class T = double> RowVector< size, T >::RowVector (const Matrix< 1, size, T > & *matrix*) [inline]

The conversion constructor for conversion of a Matrix type of single row into type RowVector.

4.13.2.4 template<int size, class T = double> RowVector< size, T >::~RowVector () [inline]

The default destructor

4.13.3 Member Function Documentation

4.13.3.1 template<int size, class T = double> virtual T* RowVector< size, T >::getElementsPointer () const [inline, virtual]

Returns:
A pointer to the first element in the vector.

Implements VectorBase< T >.

4.13.3.2 template<int size, class T = double> virtual T RowVector< size, T >::getElement (int *index*) const [inline, virtual]

Returns:
The value at position specified by index (index = 1 is the first element).

Implements VectorBase< T >.

4.13.3.3 template<int size, class T = double> virtual void RowVector< size, T >::setElement (int *index*, T *value*) [inline, virtual]

Sets an element to a value at the specified position.

Parameters:
index Position of the desired element.
value The desired element is set to this value.

Implements VectorBase< T >.

4.13.3.4 template<int size, class T = double> virtual bool RowVector< size, T >::isRowVector () const [inline, virtual]

Returns:
true

Implements VectorBase< T >.

4.13.3.5 template<int size, class T = double> virtual int RowVector< size, T >::getNumElements () const [inline, virtual]

Returns:
The number of elements in the vector.

Implements VectorBase< T >.

4.13.3.6 template<int size, class T = double> T RowVector< size, T >::operator() (int *index*) const [inline]

4.13.3.7 template<int size, class T = double> T& RowVector< size, T >::operator() (int *index*) [inline]

Access or assign the element at the position specified by index. For example:

```
myVector(2)=12.65;
```

4.13.3.8 template<int size, class T = double> RowVector<size, T>& RowVector< size, T >::operator= (const VectorBase< T > & *vectorBase*) [inline]

Assign a VectorBase type to a RowVector type. Both objects must have the same dimensions.

Reimplemented from VectorBase< T >.

4.13.3.9 template<int size, class T = double> MatrixInitializer<T> RowVector< size, T >::operator= (const T & *value*) [virtual]

Initialize a vector object.

Parameters:
value The value to which all elements in the vector are initialized. The initialization of the vector object can also be done as a comma seperated list. For example:

```
ColumnVector<3> myVector;
myVector = 67.88, 45.89, 90;
```

Implements VectorBase< T >.

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

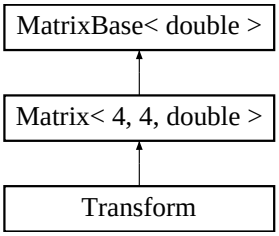
- RowVector.hpp

4.14 Transform Class Reference

The class `Transform` represents a 4x4 homogeneous transformation matrix.

`#include <Transform.hpp>`

Inheritance diagram for Transform::



Public Member Functions

- Transform ()
- Transform (const Transform &transform)
- ~Transform ()
- MatrixInitializer< double > operator= (const double &val)
- ColumnVector< 3, double > getTranslation () const
- void getRollPitchYaw (double &roll, double &pitch, double &yaw) const

4.14.1 Detailed Description

The class `Transform` represents a 4x4 homogeneous transformation matrix.

Example Program:

```
//============================================================================
// Package              : The Math Library - Ex
// Authors              : Vilas Kumar Chitrakaran
// Start Date           : Wed Dec 20 11:08:28 GMT 2000
// Compiler             : GNU C++ 2.95.3 and above
// ---------------------------------------------------------------------------
// File: Transform.t.cpp
// Example program for the Transform class.
//============================================================================

//============================================================================
// Transform.t.cpp
//----------------------------------------------------------------------------
// This program demonstrates how to find the values of the
// components of a vector in a fixed reference frame after the
// vector undergoes a series of transformations about the frame.
//============================================================================

#include "Transform.hpp"
#include <math.h>
#ifndef M_PI
 #define M_PI 3.14159265358979323846
#endif

using namespace std;
```

```
int main()
{
 ColumnVector<4, double> initialVector;
 ColumnVector<4, double> finalVector;
 Transform firstRotation;
 Transform secondRotation;
 Transform transform;

 // The initial position is [1 0 0 1] in homogeneous coordinates
 initialVector = 1, 0, 0, 1;

 // Finding the new co-ordinates. (Angles must be specified in radians.)
 transform = xRotation(0.5 * M_PI) * yRotation(0.5 * M_PI)  * translation(0,0,6);

 finalVector = transform * initialVector;
 cout << "* Position vector after transformation in fixed frame : "
      << transpose(finalVector) << endl;
 cout << "* Translation : " << transpose(transform.getTranslation()) << endl;

 // To get the roll/pitch/yaw angles of the new vector.
 double roll;
 double pitch;
 double yaw;

 transform.getRollPitchYaw(roll, pitch, yaw);
 roll *= 180.0/M_PI; pitch *= 180.0/M_PI; yaw *= 180.0/M_PI;
 cout << "* (yaw, pitch, roll) angles of the new vector (degrees) : ("
      << yaw << "," << pitch << "," << roll << ")" << endl;

 // You can get back the initial position vector by the inverse transformation..
 ColumnVector<4, double> initialVectorAgain;
 initialVectorAgain = inverse(transform) * finalVector;

 cout << "* After inverse transform : " << transpose(initialVectorAgain) << endl;
 cout << "* Should be the same as what we began with : " << transpose(initialVector) << endl;

 return 0;
}
```

4.14.2 Constructor & Destructor Documentation

4.14.2.1 Transform::Transform () [inline]

The default constructor for the Transform object. The Transform matrix is initialized to the following form.

```
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

4.14.2.2 Transform::Transform (const Transform & *transform*) [inline]

Copy Constructor.

4.14.2.3 Transform::~Transform () [inline]

The default destructor.

4.14.3 Member Function Documentation

4.14.3.1 MatrixInitializer<double> Transform::operator= (const double & *val*) [inline, virtual]

This function provides an overloaded assignment operator for initializing the elements of a Transform. The initialization of the Transform object can be done as a comma seperated list. For example:

```
Transform myTransform;
   myTransform = cos(x), sin(x),...so on ;
```

Parameters:

val The comma separated list of elements.

Reimplemented from Matrix< 4, 4, double >.

4.14.3.2 ColumnVector<3,double> Transform::getTranslation () const [inline]

Returns:

The position vector (last column) from the matrix.

4.14.3.3 void Transform::getRollPitchYaw (double & *roll*, double & *pitch*, double & *yaw*) const [inline]

This function returns the roll (z), pitch (y) and yaw (x) angles from the homogeneous transformation matrix. The angles are defined as follows: The rotational part of the Transform matrix is obtained by first defining a rotation about X axis by `yaw` radians, then a rotation about the Y axis by `pitch` radians and finally a rotation about the Z axis by `roll` radians, all rotations being relative to a fixed XYZ frame. The definition follows the description in the following textbook: M. W. Spong, and M. Vidyasagar, Robot Dynamics and Control, John Wiley and Sons, ISBN: 047161243, 1989.

NOTE: There are multiple solutions (combinations of angles) that result in the same rotation matrix, but are physically different orientations. This function returns a solution corresponding to cos(pitch) $>$ 0, i.e., -pi/2 $<$ pitch $<$ pi/2. If the assumption that cos(pitch) $>$ 0 does not hold, incorrect solutions are returned. Beware of gimbal lock that happens when cos(pitch) = 0, in which case, the roll and the yaw angles are indistinguishable.

Parameters:

roll,pitch,yaw The angles extracted from the transformation matrix.

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

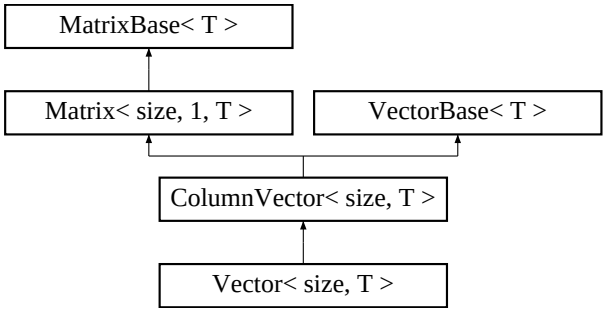
- Transform.hpp

4.15 Vector< size, T > Class Template Reference

The class `Vector` provides is equivalent to a `ColumnVector` object.

`#include <Vector.hpp>`

Inheritance diagram for Vector< size, T >::



Public Member Functions

- Vector ()
- Vector (const Vector< size, T > &vector)
- ~Vector ()
- Vector (const Matrix< size, 1, T > &matrix)
- Vector< size, T > & operator= (const VectorBase< T > &vectorBase)
- MatrixInitializer< T > operator= (const T &value)

4.15.1 Detailed Description

template<int size, class T = double> class Vector< size, T >

The class `Vector` provides is equivalent to a `ColumnVector` object.

Example Program: See the Example program for class ColumnVector.

4.15.2 Constructor & Destructor Documentation

4.15.2.1 template<int size, class T = double> Vector< size, T >::Vector () [inline]

The default constructor. No element initializations.

4.15.2.2 template<int size, class T = double> Vector< size, T >::Vector (const Vector< size, T > & *vector*) [inline]

Copy Constructor.

4.15.2.3 template<int size, class T = double> Vector< size, T >::~Vector () [inline]

The default Destructor.

4.15.2.4 template<int size, class T = double> Vector< size, T >::Vector (const Matrix< size, 1, T > & *matrix*) [inline]

The conversion constructor for conversion of a Matrix type of single column into type Vector.

4.15.3 Member Function Documentation

4.15.3.1 template<int size, class T = double> Vector<size, T>& Vector< size, T >::operator= (const VectorBase< T > & *vectorBase*) [inline]

Assign a VectorBase type to a Vector type. Both objects must have the same dimensions.

Reimplemented from ColumnVector< size, T >.

4.15.3.2 template<int size, class T = double> MatrixInitializer<T> Vector< size, T >::operator= (const T & *value*) [virtual]

Initialize a vector object.

Parameters:

value The value to which all elements in the vector are initialized. The initialization of the vector object can also be done as a comma seperated list. For example:

```
ColumnVector<3> myVector;
myVector = 67.88, 45.89, 90;
```

Reimplemented from ColumnVector< size, T >.

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

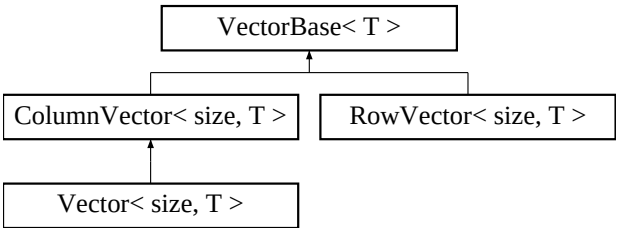
- Vector.hpp

4.16 VectorBase< T > Class Template Reference

The pure virtual base class for ColumnVector, RowVector and Vector classes.

`#include <VectorBase.hpp>`

Inheritance diagram for VectorBase< T >::



Public Member Functions

- VectorBase ()
- virtual ~VectorBase ()
- virtual T * getElementsPointer () const =0
- virtual T getElement (int i) const =0
- virtual void setElement (int index, T value)=0
- virtual int getNumElements () const =0
- T norm () const
- virtual bool isRowVector () const =0
- bool isColumnVector () const
- VectorBase< T > & operator= (const VectorBase< T > &vectorBase)
- virtual MatrixInitializer< T > operator= (const T &value)=0
- void output (std::ostream &outputStream=std::cout)

Protected Member Functions

- VectorBase (VectorBase< T > &m)

4.16.1 Detailed Description

template<class T = double> class VectorBase< T >

The pure virtual base class for ColumnVector, RowVector and Vector classes.

4.16.2 Constructor & Destructor Documentation

4.16.2.1 template<class T = double> VectorBase< T >::VectorBase () [inline]

The default constructor.

4.16.2.2 template<class T = double> virtual VectorBase< T >::~VectorBase () [inline, virtual]

The default destructor.

4.16.2.3 template<class T = double> VectorBase< T >::VectorBase (VectorBase< T > & *m*) [inline, protected]

4.16.3 Member Function Documentation

4.16.3.1 template<class T = double> virtual T* VectorBase< T >::getElementsPointer () const [pure virtual]

Returns:
A pointer to the first element in a vector.

Implemented in ColumnVector< size, T >, and RowVector< size, T >.

4.16.3.2 template<class T = double> virtual T VectorBase< T >::getElement (int *i*) const [pure virtual]

Returns:
The element at the index i.

Implemented in ColumnVector< size, T >, and RowVector< size, T >.

4.16.3.3 template<class T = double> virtual void VectorBase< T >::setElement (int *index*, T *value*) [pure virtual]

Sets an element to a value at the specified position.

Parameters:
index Position of the desired element.
value The desired element is set to this value.

Implemented in ColumnVector< size, T >, and RowVector< size, T >.

4.16.3.4 template<class T = double> virtual int VectorBase< T >::getNumElements () const [pure virtual]

Returns:
The number of elements in the vector.

Implemented in ColumnVector< size, T >, and RowVector< size, T >.

4.16.3.5 template<class T = double> T VectorBase< T >::norm () const [inline]

Returns:
2-norm of the vector.

4.16.3.6 template<class T = double> virtual bool VectorBase< T >::isRowVector () const [pure virtual]

Returns:
'true' if the vector instantiated is a RowVector.

Implemented in ColumnVector< size, T >, and RowVector< size, T >.

4.16.3.7 template<class T = double> bool VectorBase< T >::isColumnVector () const [inline]

Returns:
'true' if the vector instantiated is a ColumnVector.

4.16.3.8 template<class T = double> VectorBase<T>& VectorBase< T >::operator= (const VectorBase< T > & *vectorBase*)

Assignment operator between two VectorBase types.

Reimplemented in ColumnVector< size, T >, RowVector< size, T >, and Vector< size, T >.

4.16.3.9 template<class T = double> virtual MatrixInitializer<T> VectorBase< T >::operator= (const T & *value*) [pure virtual]

Initialize a vector object.

Implemented in ColumnVector< size, T >, RowVector< size, T >, and Vector< size, T >.

4.16.3.10 template<class T = double> void VectorBase< T >::output (std::ostream & *outputStream* = std::cout)

Returns:
The elements in the vector to the output stream (by default the output is to the console)

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

- VectorBase.hpp

Chapter 5

QMath Matrix Library File Documentation

5.1 Adams3Integrator.hpp File Reference

```
#include "Integrator.hpp"
```

Classes

- class Adams3Integrator< T >

  Numerical integration using Adam's 3'rd order method.

5.2 ColumnVector.hpp File Reference

```
#include "Matrix.hpp"
#include "VectorBase.hpp"
```

Classes

- class ColumnVector< size, T >

  A class for column vectors.

Functions

- template<class T> ColumnVector< 3, T > crossProduct (const ColumnVector< 3, T > &v1, const ColumnVector< 3, T > &v2)
- template<int size, class T> ColumnVector< size, T > elementProduct (const ColumnVector< size, T > &v1, const ColumnVector< size, T > &v2)
- template<int size, class T> T dotProduct (const ColumnVector< size, T > &v1, const ColumnVector< size, T > &v2)
- template<int size, class T> RowVector< size, T > transpose (const ColumnVector< size, T > &vector)

5.2.1 Function Documentation

5.2.1.1 template<class T> ColumnVector<3,T> crossProduct (const ColumnVector< 3, T > & *v1*, const ColumnVector< 3, T > & *v2*)

Generates the cross product of two 3 dimensional column vectors.

Parameters:

v1,v2 The 3D column-vector arguments.

Returns:

The cross product.

5.2.1.2 template<int size, class T> ColumnVector<size, T> elementProduct (const ColumnVector< size, T > & *v1*, const ColumnVector< size, T > & *v2*)

This function performs multiplication between two column vectors element-by-element.

Parameters:

v1,v2 The column-vector arguments

Returns:

The product.

5.2.1.3 template<int size, class T> T dotProduct (const ColumnVector< size, T > & *v1*, const ColumnVector< size, T > & *v2*)

Dot (inner) product between two column-vectors.

Parameters:
v1,v2 The column-vector arguments.

Returns:
The scalar product.

5.2.1.4 template<int size, class T> RowVector<size, T> transpose (const ColumnVector< size, T > & *vector*) [inline]

Returns:
The transpose of type RowVector.

5.3 Differentiator.hpp File Reference

```
#include "LowpassFilter.hpp"
#include <math.h>
```

Classes

- class Differentiator< T >

  This is the base class for differentiators.

5.4 Differentiator4O.hpp File Reference

```
#include "Differentiator.hpp"
```

Classes

- class Differentiator4O< T >

  Fourth order differentation followed by low-pass filtering.

5.5 GSLCompat.hpp File Reference

```
#include "Vector.hpp"
#include "gsl/gsl_matrix.h"
#include "gsl/gsl_vector.h"
```

Functions

- void GSLCompat_matrix (MatrixBase< double > *q, gsl_matrix *gsl)
- void GSLCompat_matrix (MatrixBase< char > *q, gsl_matrix_char *gsl)
- void GSLCompat_matrix (MatrixBase< unsigned char > *q, gsl_matrix_uchar *gsl)
- void GSLCompat_matrix (MatrixBase< short > *q, gsl_matrix_short *gsl)
- void GSLCompat_matrix (MatrixBase< unsigned short > *q, gsl_matrix_ushort *gsl)
- void GSLCompat_matrix (MatrixBase< int > *q, gsl_matrix_int *gsl)
- void GSLCompat_matrix (MatrixBase< unsigned int > *q, gsl_matrix_uint *gsl)
- void GSLCompat_matrix (MatrixBase< long > *q, gsl_matrix_long *gsl)
- void GSLCompat_matrix (MatrixBase< unsigned long > *q, gsl_matrix_ulong *gsl)
- void GSLCompat_matrix (MatrixBase< float > *q, gsl_matrix_float *gsl)
- void GSLCompat_matrix (MatrixBase< long double > *q, gsl_matrix_long_double *gsl)
- void GSLCompat_vector (VectorBase< double > *q, gsl_vector *gsl)
- void GSLCompat_vector (VectorBase< char > *q, gsl_vector_char *gsl)
- void GSLCompat_vector (VectorBase< unsigned char > *q, gsl_vector_uchar *gsl)
- void GSLCompat_vector (VectorBase< short > *q, gsl_vector_short *gsl)
- void GSLCompat_vector (VectorBase< unsigned short > *q, gsl_vector_ushort *gsl)
- void GSLCompat_vector (VectorBase< int > *q, gsl_vector_int *gsl)
- void GSLCompat_vector (VectorBase< unsigned int > *q, gsl_vector_uint *gsl)
- void GSLCompat_vector (VectorBase< long > *q, gsl_vector_long *gsl)
- void GSLCompat_vector (VectorBase< unsigned long > *q, gsl_vector_ulong *gsl)
- void GSLCompat_vector (VectorBase< float > *q, gsl_vector_float *gsl)
- void GSLCompat_vector (VectorBase< long double > *q, gsl_vector_long_double *gsl)

5.5.1 Function Documentation

5.5.1.1 void GSLCompat_matrix (MatrixBase< double > * *q*, gsl_matrix * *gsl*)

Prototypes for typecasting to GSL data types. Note that the functions do not copy data between objects. Hence, changes made to contents of one type will reflect in the contents of the converted type. Supported data types: char, unsigned char, short, unsigned short, int, unsigned int, long, unsigned long, float, double, long double. Obtain a gsl_matrix pointer from a Matrix pointer for data type double. Subsequent modification of one object will be reflected in the other.

Parameters:

q A pointer to QMath object

gsl A pointer to GSL object

5.5.1.2 void GSLCompat_matrix (MatrixBase< char > * *q*, gsl_matrix_char * *gsl*)

5.5.1.3 void GSLCompat_matrix (MatrixBase< unsigned char > * *q*, gsl_matrix_uchar * *gsl*)

5.5.1.4 void GSLCompat_matrix (MatrixBase< short > * *q*, gsl_matrix_short * *gsl*)

5.5.1.5 void GSLCompat_matrix (MatrixBase< unsigned short > * *q*, gsl_matrix_ushort * *gsl*)

5.5.1.6 void GSLCompat_matrix (MatrixBase< int > * *q*, gsl_matrix_int * *gsl*)

5.5.1.7 void GSLCompat_matrix (MatrixBase< unsigned int > * *q*, gsl_matrix_uint * *gsl*)

5.5.1.8 void GSLCompat_matrix (MatrixBase< long > * *q*, gsl_matrix_long * *gsl*)

5.5.1.9 void GSLCompat_matrix (MatrixBase< unsigned long > * *q*, gsl_matrix_ulong * *gsl*)

5.5.1.10 void GSLCompat_matrix (MatrixBase< float > * *q*, gsl_matrix_float * *gsl*)

5.5.1.11 void GSLCompat_matrix (MatrixBase< long double > * *q*, gsl_matrix_long_double * *gsl*)

5.5.1.12 void GSLCompat_vector (VectorBase< double > * *q*, gsl_vector * *gsl*)

Obtain an gsl_vector pointer from a Vector pointer for data type double. Subsequent modification of one object will be reflected in the other.

Parameters:

q A pointer to QMath object

gsl A pointer to GSL object

5.5.1.13 void GSLCompat_vector (VectorBase< char > * *q*, gsl_vector_char * *gsl*)

5.5.1.14 void GSLCompat_vector (VectorBase< unsigned char > * *q*, gsl_vector_uchar * *gsl*)

5.5.1.15 void GSLCompat_vector (VectorBase< short > * *q*, gsl_vector_short * *gsl*)

5.5.1.16 void GSLCompat_vector (VectorBase< unsigned short > * *q*, gsl_vector_ushort * *gsl*)

5.5.1.17 void GSLCompat_vector (VectorBase< int > * *q*, gsl_vector_int * *gsl*)

5.5.1.18 void GSLCompat_vector (VectorBase< unsigned int > * *q*, gsl_vector_uint * *gsl*)

5.5.1.19 void GSLCompat_vector (VectorBase< long > * *q*, gsl_vector_long * *gsl*)

5.5.1.20 void GSLCompat_vector (VectorBase< unsigned long > * *q*, gsl_vector_ulong * *gsl*)

5.5.1.21 void GSLCompat_vector (VectorBase< float > * *q*, gsl_vector_float * *gsl*)

5.5.1.22 void GSLCompat_vector (VectorBase< long double > * *q*, gsl_vector_long_double * *gsl*)

5.6 HighpassFilter.hpp File Reference

```
#include <iostream>
#include <math.h>
```

Classes

- class HighpassFilter< T >

  A high-pass second order butterworth filter.

Defines

- #define M_PI 3.14159265358979323846

5.6.1 Define Documentation

5.6.1.1 #define M_PI 3.14159265358979323846

5.7 Integrator.hpp File Reference

Classes

- class Integrator< T >

  The base class for integrators.

5.8 LowpassFilter.hpp File Reference

```
#include <math.h>
```

Classes

- class LowpassFilter< T >

  A second order butterworth lowpass filter.

Defines

- #define M_PI 3.14159265358979323846

5.8.1 Define Documentation

5.8.1.1 #define M_PI 3.14159265358979323846

5.9 MathException.hpp File Reference

Classes

- class MathException

  Run-time exception handling for the math library.

Typedefs

- typedef enum _QMathException QMathException_t

Enumerations

- enum _QMathException {

  QMathException_unknown = 0x00, QMathException_illegalIndex = 0x01, QMathException_-singular = 0x02, QMathException_divideByZero = 0x03,

  QMathException_incompatibleSize = 0x04, QMathException_typeMismatch = 0x05, QMath-Exception_dimensionTooLarge = 0x06 }

  Supported exception types.

Variables

- struct {
  QMathException_t error
  char * errorMsg
  } QMathExceptions [ ]

5.9.1 Typedef Documentation

5.9.1.1 typedef enum _QMathException QMathException_t

5.9.2 Enumeration Type Documentation

5.9.2.1 enum _QMathException

Supported exception types.

Enumerator:

QMathException_unknown Undocumented error.

QMathException_illegalIndex Illegal index.

QMathException_singular Singular matrix.

QMathException_divideByZero Division by 0.

QMathException_incompatibleSize Operation between two non-conformable matrices.

QMathException_typeMismatch Operation between incompatible data types (Ex: int and double).

QMathException_dimensionTooLarge Matrix dimensions too large for the library to handle (Ex: calling inverse() on matrices larger than 8 x 8.)

5.9.3 Variable Documentation

5.9.3.1 QMathException_t error

5.9.3.2 char* errorMsg

5.9.3.3 struct { ... } QMathExceptions[ ] [static]

5.10 Matrix.hpp File Reference

```
#include "MatrixBase.hpp"
#include "VectorBase.hpp"
#include <iostream>
#include <iomanip>
#include <math.h>
```

Classes

- class Matrix< nRows, nCols, T >

  Methods for mathematical operations on matrices.

Functions

- template<int r, int c, class T> Matrix< r, c, T > operator+ (const Matrix< r, c, T > &lhs, const Matrix< r, c, T > &rhs)
- template<int r, int c, class T> Matrix< r, c, T > operator- (const Matrix< r, c, T > &lhs, const Matrix< r, c, T > &rhs)
- template<int r, int c, class T> Matrix< r, c, T > operator * (const Matrix< r, c, T > &lhs, const T &s)
- template<int r, int c, class T> Matrix< r, c, T > operator * (const T &s, const Matrix< r, c, T > &rhs)
- template<int r, int c, class T> Matrix< r, c, T > operator/ (const Matrix< r, c, T > &lhs, const T &s)
- template<int r, int c, class T> std::ostream & operator<< (std::ostream &out, const Matrix< r, c, T > &m)
- template<int r, int c, class T> std::istream & operator>> (std::istream &in, Matrix< r, c, T > &m)
- template<int r, int c, class T> bool operator== (const Matrix< r, c, T > &lhs, const Matrix< r, c, T > &rhs)
- template<int r, int c, class T> bool operator!= (const Matrix< r, c, T > &lhs, const Matrix< r, c, T > &rhs)

5.10.1 Function Documentation

5.10.1.1 template<int r, int c, class T> Matrix<r,c,T> operator+ (const Matrix< r, c, T > & *lhs*, const Matrix< r, c, T > & *rhs*)

5.10.1.2 template<int r, int c, class T> Matrix<r,c,T> operator- (const Matrix< r, c, T > & *lhs*, const Matrix< r, c, T > & *rhs*)

5.10.1.3 template<int r, int c, class T> Matrix<r,c,T> operator * (const Matrix< r, c, T > & *lhs*, const T & *s*)

5.10.1.4 template<int r, int c, class T> Matrix<r, c, T> operator * (const T & *s*, const Matrix< r, c, T > & *rhs*)

5.10.1.5 template<int r, int c, class T> Matrix<r,c,T> operator/ (const Matrix< r, c, T > & *lhs*, const T & *s*)

5.10.1.6 template<int r, int c, class T> std::ostream& operator<< (std::ostream & *out*, const Matrix< r, c, T > & *m*)

5.10.1.7 template<int r, int c, class T> std::istream& operator>> (std::istream & *in*, Matrix< r, c, T > & *m*)

5.10.1.8 template<int r, int c, class T> bool operator== (const Matrix< r, c, T > & *lhs*, const Matrix< r, c, T > & *rhs*)

5.10.1.9 template<int r, int c, class T> bool operator!= (const Matrix< r, c, T > & *lhs*, const Matrix< r, c, T > & *rhs*)

5.11 MatrixBase.hpp File Reference

```
#include <iostream>
#include "MathException.hpp"
#include "MatrixInitializer.hpp"
```

Classes

- class MatrixBase< T >

  This is a pure virtual base class for `Matrix`.

5.12 MatrixInitializer.hpp File Reference

```
#include "MathException.hpp"
```

Classes

- class MatrixInitializer< T >

  This class is used internally by the library to initialize the `Matrix` *and its derived class objects.*

5.13 ODESolverRK4.hpp File Reference

```
#include "Vector.hpp"
```

Classes

- class ODESolverRK4< T >

  Solver for ordinary differential equations using 4th order Runge Kutta method.

5.14 RowVector.hpp File Reference

```
#include "VectorBase.hpp"
#include "Matrix.hpp"
```

Classes

- class RowVector< size, T >

  A class for row vectors.

Functions

- template<class T> RowVector< 3, T > crossProduct (const RowVector< 3, T > &v1, const RowVector< 3, T > &v2)
- template<int size, class T> RowVector< size, T > elementProduct (const RowVector< size, T > &v1, const RowVector< size, T > &v2)
- template<int size, class T> T dotProduct (const RowVector< size, T > &v1, const RowVector< size, T > &v2)
- template<int size, class T> ColumnVector< size, T > transpose (const RowVector< size, T > &vector)

5.14.1 Function Documentation

5.14.1.1 template<class T> RowVector<3,T> crossProduct (const RowVector< 3, T > & *v1*, const RowVector< 3, T > & *v2*)

Generates the cross product of two 3 dimensional row vectors.

Parameters:

v1,v2 The 3D row vector arguments.

Returns:

The cross product.

5.14.1.2 template<int size, class T> RowVector<size, T> elementProduct (const RowVector< size, T > & *v1*, const RowVector< size, T > & *v2*)

This function performs multiplication between two column vectors element-by-element.

Parameters:

v1,v2 The row vector arguments

Returns:

The product.

5.14.1.3 template<int size, class T> T dotProduct (const RowVector< size, T > & *v1*, const RowVector< size, T > & *v2*)

Dot (inner) product between two row vectors.

Parameters:
v1,v2 The row vector arguments.

Returns:
The scalar product.

5.14.1.4 template<int size, class T> ColumnVector<size, T> transpose (const RowVector< size, T > & *vector*)

return The transpose of the type `ColumnVector`.

5.15 Transform.hpp File Reference

```
#include "Matrix.hpp"
#include "ColumnVector.hpp"
#include "RowVector.hpp"
```

Classes

- class Transform

  The class `Transform` *represents a 4x4 homogeneous transformation matrix.*

Functions

- Transform operator ∗ (const Transform &firstTransfrom, const Transform &secondTransform)
- Transform inverse (const Transform &t)
- Transform translation (double x, double y, double z)
- Transform xRotation (double theta)
- Transform yRotation (double theta)
- Transform zRotation (double theta)
- Transform vectorRotation (const ColumnVector< 3, double > &vector, double theta)
- Transform rpyRotation (double roll, double pitch, double yaw)

5.15.1 Function Documentation

5.15.1.1 Transform operator ∗ (const Transform & *firstTransfrom*, const Transform & *secondTransform*) [inline]

Overloading binary operator ∗ for multiplication between two transforms.

5.15.1.2 Transform inverse (const Transform & *t*) [inline]

Returns:
The inverse of the transform *t*.

5.15.1.3 Transform translation (double *x*, double *y*, double *z*) [inline]

Returns:
A *Transform* representing a translation of *x*, *y* and *z* units in the X, Y and Z directions.

5.15.1.4 Transform xRotation (double *theta*)

Returns:
A *Transform* representing a rotation of angle *theta* radians about the X axis.

5.15.1.5 Transform yRotation (double *theta*)

Returns:
A *Transform* representing a rotation of angle *theta* about the Y axis.

5.15.1.6 Transform zRotation (double *theta*)

Returns:
A *Transform* representing a rotation of angle *theta* radians about the Z axis.

5.15.1.7 Transform vectorRotation (const ColumnVector< 3, double > & *vector*, double *theta*)

Returns:
A *Transform* representing a rotation of angle *theta* radians about an arbitrary vector *vector*.

5.15.1.8 Transform rpyRotation (double *roll*, double *pitch*, double *yaw*) [inline]

Sets the rotational part of the transform matrix from the `roll`, `pitch` and `yaw` angles as described in the following: M. W. Spong, and M. Vidyasagar, Robot Dynamics and Control, John Wiley and Sons, ISBN: 047161243, 1989. The rotational part of the Transform matrix is obtained by first defining a rotation about X axis by `yaw` radians, then a rotation about the Y axis by `pitch` radians and finally a rotation about the Z axis by `roll` radians, all rotations being relative to a fixed XYZ frame.

Parameters:
yaw Rotation about the X axis.

pitch Successive rotation about the Y axis.

roll Successive rotation about the Z axis.

Returns:
A transformation matrix with the rotation matrix set.

5.16 Vector.hpp File Reference

```
#include "ColumnVector.hpp"
```

Classes

- class Vector< size, T >

  The class `Vector` *provides is equivalent to a* `ColumnVector` *object.*

5.17 VectorBase.hpp File Reference

```
#include <iostream>
#include "MathException.hpp"
#include "MatrixInitializer.hpp"
```

Classes

- class VectorBase< T >

  The pure virtual base class for `ColumnVector`, `RowVector` *and* `Vector` *classes.*

Index