

Heidelberg Educational Numerics Library

Generated by Doxygen 1.9.3



<b>1 Hierarchical Index</b>	<b>1</b>
1.1 Class Hierarchy	1
<b>2 Class Index</b>	<b>3</b>
2.1 Class List	3
<b>3 File Index</b>	<b>5</b>
3.1 File List	5
<b>4 Class Documentation</b>	<b>7</b>
4.1 hdnum::Banach Class Reference	7
4.1.1 Detailed Description	7
4.2 hdnum::SparseMatrix< REAL >::builder Class Reference	8
4.3 hdnum::SparseMatrix< REAL >::column_index_iterator Class Reference	8
4.4 hdnum::SparseMatrix< REAL >::const_column_index_iterator Class Reference	9
4.5 hdnum::SparseMatrix< REAL >::const_row_iterator Class Reference	9
4.6 hdnum::oc::OpCounter< F >::Counters Struct Reference	10
4.6.1 Detailed Description	11
4.7 hdnum::DenseMatrix< REAL > Class Template Reference	11
4.7.1 Detailed Description	14
4.7.2 Member Function Documentation	14
4.7.2.1 colsize()	14
4.7.2.2 mm()	14
4.7.2.3 mv()	15
4.7.2.4 operator>()	16
4.7.2.5 operator*() [1/2]	17
4.7.2.6 operator*() [2/2]	17
4.7.2.7 operator*=( )	18
4.7.2.8 operator+( )	19
4.7.2.9 operator+=( )	19
4.7.2.10 operator-( )	20
4.7.2.11 operator-=( )	20
4.7.2.12 operator/=( )	21
4.7.2.13 operator=( ) [1/2]	21
4.7.2.14 operator=( ) [2/2]	22
4.7.2.15 rowsize()	22
4.7.2.16 sc()	22
4.7.2.17 scientific()	23
4.7.2.18 sr()	23
4.7.2.19 sub()	24
4.7.2.20 transpose()	24
4.7.2.21 umm()	25
4.7.2.22 umv() [1/2]	26

4.7.2.23 umv() [2/2]	26
4.7.2.24 update()	27
4.7.3 Friends And Related Function Documentation	28
4.7.3.1 identity()	28
4.7.3.2 readMatrixFromFileDat()	28
4.7.3.3 readMatrixFromFileMatrixMarket()	29
4.7.3.4 spd()	30
4.7.3.5 vandermonde()	30
4.8 hdnum::DIRK< M, S > Class Template Reference	31
4.8.1 Detailed Description	32
4.8.2 Constructor & Destructor Documentation	32
4.8.2.1 DIRK() [1/2]	32
4.8.2.2 DIRK() [2/2]	33
4.9 hdnum::EE< M > Class Template Reference	33
4.9.1 Detailed Description	34
4.10 hdnum::ErrorException Class Reference	34
4.10.1 Detailed Description	34
4.11 hdnum::Exception Class Reference	35
4.11.1 Detailed Description	35
4.12 hdnum::GenericNonlinearProblem< Lambda, Vec > Class Template Reference	35
4.12.1 Detailed Description	36
4.13 hdnum::Heun2< M > Class Template Reference	36
4.13.1 Detailed Description	37
4.14 hdnum::Heun3< M > Class Template Reference	37
4.14.1 Detailed Description	38
4.15 hdnum::IE< M, S > Class Template Reference	38
4.15.1 Detailed Description	39
4.16 hdnum::ImplicitRungeKuttaStepProblem< M > Class Template Reference	39
4.16.1 Detailed Description	40
4.17 hdnum::InvalidStateException Class Reference	40
4.17.1 Detailed Description	41
4.18 hdnum::IOError Class Reference	41
4.18.1 Detailed Description	41
4.19 hdnum::Kutta3< M > Class Template Reference	41
4.19.1 Detailed Description	42
4.20 hdnum::MathError Class Reference	42
4.20.1 Detailed Description	43
4.21 hdnum::ModifiedEuler< M > Class Template Reference	43
4.21.1 Detailed Description	44
4.22 hdnum::Newton Class Reference	44
4.22.1 Detailed Description	45
4.23 hdnum::NotImplemented Class Reference	45

4.23.1 Detailed Description	46
4.24 <code>hdnum::oc::OpCounter&lt; F &gt;</code> Class Template Reference	46
4.24.1 Detailed Description	47
4.25 <code>hdnum::OutOfMemoryError</code> Class Reference	47
4.25.1 Detailed Description	48
4.26 <code>hdnum::RangeError</code> Class Reference	48
4.26.1 Detailed Description	48
4.27 <code>hdnum::RE&lt; M, S &gt;</code> Class Template Reference	48
4.27.1 Detailed Description	49
4.28 <code>hdnum::RKF45&lt; M &gt;</code> Class Template Reference	50
4.28.1 Detailed Description	50
4.29 <code>hdnum::SparseMatrix&lt; REAL &gt;::row_iterator</code> Class Reference	51
4.30 <code>hdnum::RungeKutta&lt; M, S &gt;</code> Class Template Reference	51
4.30.1 Detailed Description	52
4.31 <code>hdnum::RungeKutta4&lt; M &gt;</code> Class Template Reference	53
4.31.1 Detailed Description	53
4.32 <code>hdnum::SGrid&lt; N, DF, dimension &gt;</code> Class Template Reference	54
4.32.1 Detailed Description	55
4.32.2 Constructor & Destructor Documentation	55
4.32.2.1 <code>SGrid()</code>	55
4.32.3 Member Function Documentation	55
4.32.3.1 <code>getNeighborIndex()</code>	56
4.33 <code>hdnum::SparseMatrix&lt; REAL &gt;</code> Class Template Reference	56
4.33.1 Detailed Description	58
4.33.2 Constructor & Destructor Documentation	58
4.33.2.1 <code>SparseMatrix()</code>	58
4.33.3 Member Function Documentation	59
4.33.3.1 <code>begin()</code> [1/2]	59
4.33.3.2 <code>begin()</code> [2/2]	59
4.33.3.3 <code>cbegin()</code>	59
4.33.3.4 <code>cend()</code>	59
4.33.3.5 <code>colsize()</code>	60
4.33.3.6 <code>end()</code> [1/2]	60
4.33.3.7 <code>end()</code> [2/2]	60
4.33.3.8 <code>identity()</code>	61
4.33.3.9 <code>matchingIdentity()</code>	61
4.33.3.10 <code>mv()</code>	61
4.33.3.11 <code>norm_infty()</code>	62
4.33.3.12 <code>operator*()</code>	62
4.33.3.13 <code>operator*=( )</code>	62
4.33.3.14 <code>operator/=( )</code>	63
4.33.3.15 <code>rowsize()</code>	63

4.33.3.16 scientific()	63
4.33.3.17 umv()	64
4.33.4 Friends And Related Function Documentation	64
4.33.4.1 identity()	64
4.34 hdnum::SquareRootProblem< N > Class Template Reference	65
4.34.1 Detailed Description	65
4.35 hdnum::StationarySolver< M > Class Template Reference	66
4.35.1 Detailed Description	66
4.36 hdnum::SystemError Class Reference	66
4.36.1 Detailed Description	67
4.37 hdnum::Timer Class Reference	67
4.37.1 Detailed Description	67
4.38 hdnum::TimerError Class Reference	68
4.38.1 Detailed Description	68
4.39 hdnum::Vector< REAL > Class Template Reference	68
4.39.1 Detailed Description	70
4.39.2 Member Function Documentation	70
4.39.2.1 operator*()	70
4.39.2.2 operator+()	71
4.39.2.3 operator-()	71
4.39.2.4 operator=()	72
4.39.2.5 scientific()	72
4.39.2.6 sub()	73
4.39.2.7 two_norm()	73
4.39.3 Friends And Related Function Documentation	74
4.39.3.1 fill()	74
4.39.3.2 gnuplot()	74
4.39.3.3 operator<<()	75
4.39.3.4 readVectorFromFile()	75
4.39.3.5 unitvector()	76
<b>5 File Documentation</b>	<b>77</b>
5.1 densematrix.hh	77
5.2 src/exceptions.hh File Reference	83
5.2.1 Detailed Description	84
5.2.2 Macro Definition Documentation	84
5.2.2.1 HDNUM_ERROR	84
5.2.2.2 HDNUM_THROW	85
5.3 exceptions.hh	86
5.4 src/lr.hh File Reference	87
5.4.1 Detailed Description	88
5.5 lr.hh	88

5.6 src/newton.hh File Reference	91
5.6.1 Detailed Description	91
5.6.2 Function Documentation	91
5.6.2.1 getNonlinearProblem()	91
5.7 newton.hh	92
5.8 src/ode.hh File Reference	96
5.8.1 Detailed Description	97
5.9 ode.hh	97
5.10 src/opcounter.hh File Reference	113
5.10.1 Detailed Description	116
5.11 opcounter.hh	116
5.12 src/pde.hh File Reference	126
5.12.1 Detailed Description	127
5.13 pde.hh	127
5.14 src/precision.hh File Reference	128
5.14.1 Detailed Description	128
5.15 precision.hh	128
5.16 src/qr.hh File Reference	129
5.16.1 Detailed Description	129
5.16.2 Function Documentation	130
5.16.2.1 gram_schmidt()	130
5.16.2.2 modified_gram_schmidt()	130
5.16.2.3 permute_forward()	131
5.16.2.4 qr_gram_schmidt()	131
5.16.2.5 qr_gram_schmidt_pivoting()	132
5.16.2.6 qr_gram_schmidt_simple()	133
5.17 qr.hh	134
5.18 src/qrhousholder.hh File Reference	138
5.18.1 Detailed Description	138
5.18.2 Function Documentation	138
5.18.2.1 qrhousholder()	139
5.18.2.2 qrhousholderexplicitQ()	140
5.19 qrhousholder.hh	140
5.20 src/rungekutta.hh File Reference	142
5.20.1 Detailed Description	143
5.20.2 Function Documentation	143
5.20.2.1 ordertest()	143
5.21 rungekutta.hh	144
5.22 sgrid.hh	149
5.23 sparsematrix.hh	151
5.24 src/timer.hh File Reference	161
5.24.1 Detailed Description	162

5.25 timer.hh . . . . .	162
5.26 vector.hh . . . . .	163
<b>Index</b>	<b>169</b>



# Chapter 1

## Hierarchical Index

### 1.1 Class Hierarchy

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

hdnum::Banach	7
hdnum::SparseMatrix< REAL >::builder	8
hdnum::SparseMatrix< REAL >::column_index_iterator	8
hdnum::SparseMatrix< REAL >::const_column_index_iterator	9
hdnum::SparseMatrix< REAL >::const_row_iterator	9
hdnum::oc::OpCounter< F >::Counters	10
hdnum::DenseMatrix< REAL >	11
hdnum::DenseMatrix< number_type >	11
hdnum::DIRK< M, S >	31
hdnum::EE< M >	33
hdnum::Exception	35
hdnum::ErrorException	34
hdnum::IOError	41
hdnum::InvalidStateException	40
hdnum::MathError	42
hdnum::NotImplemented	45
hdnum::RangeError	48
hdnum::SystemError	66
hdnum::OutOfMemoryError	47
hdnum::TimerError	68
hdnum::GenericNonlinearProblem< Lambda, Vec >	35
hdnum::Heun2< M >	36
hdnum::Heun3< M >	37
hdnum::IE< M, S >	38
hdnum::ImplicitRungeKuttaStepProblem< M >	39
hdnum::Kutta3< M >	41
hdnum::ModifiedEuler< M >	43
hdnum::Newton	44
hdnum::oc::OpCounter< F >	46
hdnum::RE< M, S >	48
hdnum::RK45< M >	50
hdnum::SparseMatrix< REAL >::row_iterator	51
hdnum::RungeKutta< M, S >	51
hdnum::RungeKutta4< M >	53
hdnum::SGrid< N, DF, dimension >	54

hdnum::SparseMatrix< REAL > . . . . .	56
hdnum::SquareRootProblem< N > . . . . .	65
hdnum::StationarySolver< M > . . . . .	66
hdnum::Timer . . . . .	67
std::vector	
hdnum::Vector< number_type > . . . . .	68
hdnum::Vector< hdnum::Vector< number_type > > . . . . .	68
hdnum::Vector< size_type > . . . . .	68
hdnum::Vector< REAL > . . . . .	68

## Chapter 2

# Class Index

### 2.1 Class List

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

<a href="#">hdnum::Banach</a>	7
Solve nonlinear problem using a fixed point iteration	
<a href="#">hdnum::SparseMatrix&lt; REAL &gt;::builder</a>	8
<a href="#">hdnum::SparseMatrix&lt; REAL &gt;::column_index_iterator</a>	8
<a href="#">hdnum::SparseMatrix&lt; REAL &gt;::const_column_index_iterator</a>	9
<a href="#">hdnum::SparseMatrix&lt; REAL &gt;::const_row_iterator</a>	9
<a href="#">hdnum::oc::OpCounter&lt; F &gt;::Counters</a>	
Struct storing the number of operations	10
<a href="#">hdnum::DenseMatrix&lt; REAL &gt;</a>	
Class with mathematical matrix operations	11
<a href="#">hdnum::DIRK&lt; M, S &gt;</a>	
Implementation of a general Diagonal Implicit Runge-Kutta method	31
<a href="#">hdnum::EE&lt; M &gt;</a>	
Explicit Euler method as an example for an ODE solver	33
<a href="#">hdnum::ErrorException</a>	
General Error	34
<a href="#">hdnum::Exception</a>	
Base class for Exceptions	35
<a href="#">hdnum::GenericNonlinearProblem&lt; Lambda, Vec &gt;</a>	
A generic problem class that can be set up with a lambda defining $F(x)=0$	35
<a href="#">hdnum::Heun2&lt; M &gt;</a>	
Heun method (order 2 with 2 stages)	36
<a href="#">hdnum::Heun3&lt; M &gt;</a>	
Heun method (order 3 with 3 stages)	37
<a href="#">hdnum::IE&lt; M, S &gt;</a>	
Implicit Euler using <a href="#">Newton's</a> method to solve nonlinear system	38
<a href="#">hdnum::ImplicitRungeKuttaStepProblem&lt; M &gt;</a>	
Nonlinear problem we need to solve to do one step of an implicit Runge Kutta method	39
<a href="#">hdnum::InvalidStateException</a>	
Default exception if a function was called while the object is not in a valid state for that function	40
<a href="#">hdnum::IOError</a>	
Default exception class for I/O errors	41
<a href="#">hdnum::Kutta3&lt; M &gt;</a>	
Kutta method (order 3 with 3 stages)	41
<a href="#">hdnum::MathError</a>	
Default exception class for mathematical errors	42

<a href="#">hdnum::ModifiedEuler&lt; M &gt;</a>	
Modified Euler method (order 2 with 2 stages) . . . . .	43
<a href="#">hdnum::Newton</a>	
Solve nonlinear problem using a damped <a href="#">Newton</a> method . . . . .	44
<a href="#">hdnum::NotImplemented</a>	
Default exception for dummy implementations . . . . .	45
<a href="#">hdnum::oc::OpCounter&lt; F &gt;</a> . . . . .	46
<a href="#">hdnum::OutOfMemoryError</a>	
Default exception if memory allocation fails . . . . .	47
<a href="#">hdnum::RangeError</a>	
Default exception class for range errors . . . . .	48
<a href="#">hdnum::RE&lt; M, S &gt;</a>	
Adaptive one-step method using Richardson extrapolation . . . . .	48
<a href="#">hdnum::RKF45&lt; M &gt;</a>	
Adaptive Runge-Kutta-Fehlberg method . . . . .	50
<a href="#">hdnum::SparseMatrix&lt; REAL &gt;::row_iterator</a> . . . . .	51
<a href="#">hdnum::RungeKutta&lt; M, S &gt;</a>	
Classical Runge-Kutta method (order n with n stages) . . . . .	51
<a href="#">hdnum::RungeKutta4&lt; M &gt;</a>	
Classical Runge-Kutta method (order 4 with 4 stages) . . . . .	53
<a href="#">hdnum::SGrid&lt; N, DF, dimension &gt;</a>	
Structured Grid for Finite Differences . . . . .	54
<a href="#">hdnum::SparseMatrix&lt; REAL &gt;</a>	
Sparse matrix Class with mathematical matrix operations . . . . .	56
<a href="#">hdnum::SquareRootProblem&lt; N &gt;</a>	
Example class for a nonlinear model $F(x) = 0$ ; . . . . .	65
<a href="#">hdnum::StationarySolver&lt; M &gt;</a>	
Stationary problem solver. E.g. for elliptic problems . . . . .	66
<a href="#">hdnum::SystemError</a>	
Default exception class for OS errors . . . . .	66
<a href="#">hdnum::Timer</a>	
A simple stop watch . . . . .	67
<a href="#">hdnum::TimerError</a>	
Exception thrown by the <a href="#">Timer</a> class . . . . .	68
<a href="#">hdnum::Vector&lt; REAL &gt;</a>	
Class with mathematical vector operations . . . . .	68

## Chapter 3

# File Index

### 3.1 File List

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

src/ <a href="#">densematrix.hh</a>	77
src/ <a href="#">exceptions.hh</a>	
A few common exception classes	83
src/ <a href="#">lr.hh</a>	
This file implements LU decomposition	87
src/ <a href="#">newton.hh</a>	
Newton's method with line search	91
src/ <a href="#">ode.hh</a>	
Solvers for ordinary differential equations	96
src/ <a href="#">opcounter.hh</a>	
This file implements an operator counting class	113
src/ <a href="#">pde.hh</a>	
Solvers for partial differential equations	126
src/ <a href="#">precision.hh</a>	
Find machine precision for given float type	128
src/ <a href="#">qr.hh</a>	
This file implements QR decomposition using Gram-Schmidt method	129
src/ <a href="#">qrhousholder.hh</a>	
This file implements QR decomposition using housholder transformation	138
src/ <a href="#">rungekutta.hh</a>	142
src/ <a href="#">sgrid.hh</a>	149
src/ <a href="#">sparsematrix.hh</a>	151
src/ <a href="#">timer.hh</a>	
A simple timing class	161
src/ <a href="#">vector.hh</a>	163



## Chapter 4

# Class Documentation

### 4.1 hdnum::Banach Class Reference

Solve nonlinear problem using a fixed point iteration.

```
#include <newton.hh>
```

#### Public Member Functions

- **Banach** ()  
*constructor stores reference to the model*
- void **set\_maxit** (size\_type n)  
*maximum number of iterations before giving up*
- void **set\_sigma** (double sigma\_)  
*damping parameter*
- void **set\_linesearchsteps** (size\_type n)  
*maximum number of steps in linesearch before giving up*
- void **set\_verbosity** (size\_type n)  
*control output given 0=nothing, 1=summary, 2=every step, 3=include line search*
- void **set\_abslimit** (double l)  
*basolute limit for defect*
- void **set\_reduction** (double l)  
*reduction factor*
- template<class M >  
void **solve** (const M &model, [Vector](#)< typename M::number\_type > &x) const  
*do one step*
- bool **has\_converged** () const

#### 4.1.1 Detailed Description

Solve nonlinear problem using a fixed point iteration.

solve  $F(x) = 0$ .

$$x = x - \sigma * F(x)$$

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

- [src/newton.hh](#)

## 4.2 `hdnum::SparseMatrix< REAL >::builder` Class Reference

### Public Member Functions

- **builder** ([size\\_type](#) new\_m\_rows, [size\\_type](#) new\_m\_cols)
- **builder** (const std::initializer\_list< std::initializer\_list< REAL > > &v)
- std::pair< typename std::map< [size\\_type](#), REAL >::iterator, bool > **addEntry** ([size\\_type](#) i, [size\\_type](#) j, REAL value)
- std::pair< typename std::map< [size\\_type](#), REAL >::iterator, bool > **addEntry** ([size\\_type](#) i, [size\\_type](#) j)
- bool **operator==** (const [SparseMatrix::builder](#) &other) const
- bool **operator!=** (const [SparseMatrix::builder](#) &other) const
- [size\\_type](#) **colsize** () const noexcept
- [size\\_type](#) **rowsize** () const noexcept
- [size\\_type](#) **setNumCols** ([size\\_type](#) new\_m\_cols) noexcept
- [size\\_type](#) **setNumRows** ([size\\_type](#) new\_m\_rows)
- void **clear** () noexcept
- std::string **to\_string** () const
- [SparseMatrix](#) **build** ()

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

- src/sparsematrix.hh

## 4.3 `hdnum::SparseMatrix< REAL >::column_index_iterator` Class Reference

### Public Types

- using **self\_type** = [column\\_index\\_iterator](#)
- using **difference\_type** = std::ptrdiff\_t
- using **value\_type** = std::pair< REAL &, [size\\_type](#) const & >
- using **pointer** = value\_type \*
- using **reference** = value\_type &
- using **iterator\_category** = std::bidirectional\_iterator\_tag

### Public Member Functions

- **column\_index\_iterator** (typename std::vector< REAL >::iterator vallter, std::vector< [size\\_type](#) >::iterator colIndicesIter)
- [self\\_type](#) & **operator++** ()
- [self\\_type](#) & **operator++** (int junk)
- value\_type **operator\*** ()
- value\_type::first\_type **value** ()
- value\_type::second\_type **index** ()
- bool **operator==** (const [self\\_type](#) &other)
- bool **operator!=** (const [self\\_type](#) &other)

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

- src/sparsematrix.hh



## 4.4 `hdnum::SparseMatrix< REAL >::const_column_index_iterator` Class Reference

### Public Types

- using **self\_type** = `const_column_index_iterator`
- using **difference\_type** = `std::ptrdiff_t`
- using **value\_type** = `std::pair< REAL const &, size_type const & >`
- using **pointer** = `value_type *`
- using **reference** = `value_type &`
- using **iterator\_category** = `std::bidirectional_iterator_tag`

### Public Member Functions

- **const\_column\_index\_iterator** (typename `std::vector< REAL >::const_iterator` `vallter`, `std::vector< size_type >::const_iterator` `colIndicesIter`)
- **self\_type & operator++** ()
- **self\_type operator++** (int `junk`)
- `value_type operator*` ()
- `value_type::first_type value` ()
- `value_type::second_type index` ()
- `bool operator==` (const **self\_type** &`other`)
- `bool operator!=` (const **self\_type** &`other`)

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

- `src/sparsematrix.hh`

## 4.5 `hdnum::SparseMatrix< REAL >::const_row_iterator` Class Reference

### Public Types

- using **self\_type** = `const_row_iterator`
- using **difference\_type** = `std::ptrdiff_t`
- using **value\_type** = **self\_type**
- using **pointer** = **self\_type** \*
- using **reference** = **self\_type** &
- using **iterator\_category** = `std::bidirectional_iterator_tag`

## Public Member Functions

- **const\_row\_iterator** (std::vector< [size\\_type](#) >::const\_iterator rowPtrIter, std::vector< [size\\_type](#) >::const\_iterator colIndicesIter, typename std::vector< REAL >::const\_iterator valIter)
- **const\_column\_iterator** **begin** () const
- **const\_column\_iterator** **end** () const
- **const\_column\_index\_iterator** **ibegin** () const
- **const\_column\_index\_iterator** **iend** () const
- **const\_column\_iterator** **cbegin** () const
- **const\_column\_iterator** **cend** () const
- **self\_type** & **operator++** ()
- **self\_type** & **operator++** (int junk)
- **self\_type** & **operator+=** (difference\_type offset)
- **self\_type** & **operator-=** (difference\_type offset)
- **self\_type** **operator-** (difference\_type offset)
- **self\_type** **operator+** (difference\_type offset)
- **reference** **operator[]** (difference\_type offset)
- bool **operator<** (const **self\_type** &other)
- bool **operator>** (const **self\_type** &other)
- **self\_type** & **operator\*** ()
- bool **operator==** (const **self\_type** &rhs)
- bool **operator!=** (const **self\_type** &rhs)

## Friends

- **self\_type** **operator+** (const difference\_type &offset, const **self\_type** &sec)

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

- src/sparsematrix.hh

## 4.6 hdnum::oc::OpCounter< F >::Counters Struct Reference

Struct storing the number of operations.

```
#include <opcounter.hh>
```

## Public Member Functions

- void **reset** ()
- template<typename Stream >  
void **reportOperations** (Stream &os, bool doReset=false)  
*Report operations to stream object.*
- size\_type **totalOperationCount** (bool doReset=false)  
*Get total number of operations.*
- **Counters** & **operator+=** (const **Counters** &rhs)
- **Counters** **operator-** (const **Counters** &rhs)

## Public Attributes

- size\_type **addition\_count**
- size\_type **multiplication\_count**
- size\_type **division\_count**
- size\_type **exp\_count**
- size\_type **pow\_count**
- size\_type **sin\_count**
- size\_type **sqrt\_count**
- size\_type **comparison\_count**

### 4.6.1 Detailed Description

```
template<typename F>
struct hdnum::oc::OpCounter< F >::Counters
```

Struct storing the number of operations.

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

- [src/opcounter.hh](#)

## 4.7 hdnum::DenseMatrix< REAL > Class Template Reference

Class with mathematical matrix operations.

```
#include <densematrix.hh>
```

## Public Types

- typedef std::size\_t **size\_type**  
*Type used for array indices.*
- typedef std::vector< REAL > **VType**
- typedef VType::const\_iterator **ConstVectorIterator**
- typedef VType::iterator **VectorIterator**

## Public Member Functions

- **DenseMatrix** ()  
*default constructor (empty Matrix)*
- **DenseMatrix** (const std::size\_t \_rows, const std::size\_t \_cols, const REAL def\_val=0)  
*constructor*
- **DenseMatrix** (const std::initializer\_list< std::initializer\_list< REAL > > &v)  
*constructor from initializer list*
- **DenseMatrix** (const [hdnum::SparseMatrix](#)< REAL > &other)  
*constructor from [hdnum::SparseMatrix](#)*
- void **addNewRow** (const [hdnum::Vector](#)< REAL > &rowvector)
- size\_t **rowsize** () const  
*get number of rows of the matrix*
- size\_t **colsize** () const  
*get number of columns of the matrix*
- bool **scientific** () const
- void **scientific** (bool b) const  
*Switch between floating point (default=true) and fixed point (false) display.*
- std::size\_t **iwidth** () const  
*get index field width for pretty-printing*
- std::size\_t **width** () const  
*get data field width for pretty-printing*
- std::size\_t **precision** () const  
*get data precision for pretty-printing*
- void **iwidth** (std::size\_t i) const  
*set index field width for pretty-printing*
- void **width** (std::size\_t i) const  
*set data field width for pretty-printing*
- void **precision** (std::size\_t i) const  
*set data precision for pretty-printing*
- REAL & **operator()** (const std::size\_t row, const std::size\_t col)  
*(i,j)-operator for accessing entries of a (m x n)-matrix directly*
- const REAL & **operator()** (const std::size\_t row, const std::size\_t col) const  
*read-access on matrix element A\_ij using A(i,j)*
- const ConstVectorIterator **operator[]** (const std::size\_t row) const  
*read-access on matrix element A\_ij using A[i][j]*
- VectorIterator **operator[]** (const std::size\_t row)  
*write-access on matrix element A\_ij using A[i][j]*
- [DenseMatrix](#) & **operator=** (const [DenseMatrix](#) &A)  
*assignment operator*
- [DenseMatrix](#) & **operator=** (const REAL value)  
*assignment from a scalar value*
- [DenseMatrix](#) sub (size\_type i, size\_type j, size\_type rows, size\_type cols)  
*Submatrix extraction.*
- [DenseMatrix](#) **transpose** () const  
*Transposition.*
- [DenseMatrix](#) & **operator+=** (const [DenseMatrix](#) &B)  
*Addition assignment.*
- [DenseMatrix](#) & **operator-=** (const [DenseMatrix](#) &B)  
*Subtraction assignment.*
- [DenseMatrix](#) & **operator\*=** (const REAL s)

- *Scalar multiplication assignment.*
- `DenseMatrix` & `operator/=` (const `REAL` s)
- *Scalar division assignment.*
- void `update` (const `REAL` s, const `DenseMatrix` &B)
- *Scaled update of a Matrix.*
- template<class V >
- void `mv` (`Vector`< V > &y, const `Vector`< V > &x) const
- *matrix vector product  $y = A*x$*
- template<class V >
- void `umv` (`Vector`< V > &y, const `Vector`< V > &x) const
- *update matrix vector product  $y += A*x$*
- template<class V >
- void `umv` (`Vector`< V > &y, const V &s, const `Vector`< V > &x) const
- *update matrix vector product  $y += sA*x$*
- void `mm` (const `DenseMatrix`< `REAL` > &A, const `DenseMatrix`< `REAL` > &B)
- *assign to matrix product  $C = A*B$  to matrix C*
- void `umm` (const `DenseMatrix`< `REAL` > &A, const `DenseMatrix`< `REAL` > &B)
- *add matrix product  $A*B$  to matrix C*
- void `sc` (const `Vector`< `REAL` > &x, std::size\_t k)
- *set column: make x the k'th column of A*
- void `sr` (const `Vector`< `REAL` > &x, std::size\_t k)
- *set row: make x the k'th row of A*
- `REAL norm_inf` () const
- *compute row sum norm*
- `REAL norm_1` () const
- *compute column sum norm*
- `Vector`< `REAL` > `operator*` (const `Vector`< `REAL` > &x) const
- *vector = matrix \* vector*
- `DenseMatrix operator*` (const `DenseMatrix` &x) const
- *matrix = matrix \* matrix*
- `DenseMatrix operator+` (const `DenseMatrix` &x) const
- *matrix = matrix + matrix*
- `DenseMatrix operator-` (const `DenseMatrix` &x) const
- *matrix = matrix - matrix*

## Related Functions

(Note that these are not member functions.)

- template<class T >
- void `identity` (`DenseMatrix`< T > &A)
- template<typename `REAL` >
- void `spd` (`DenseMatrix`< `REAL` > &A)
- template<typename `REAL` >
- void `vandermonde` (`DenseMatrix`< `REAL` > &A, const `Vector`< `REAL` > x)
- template<typename `REAL` >
- void `readMatrixFromFileDat` (const std::string &filename, `DenseMatrix`< `REAL` > &A)
- *Read matrix from a text file.*
- template<typename `REAL` >
- void `readMatrixFromFileMatrixMarket` (const std::string &filename, `DenseMatrix`< `REAL` > &A)
- *Read matrix from a matrix market file.*

### 4.7.1 Detailed Description

```
template<typename REAL>
class hdnum::DenseMatrix< REAL >
```

Class with mathematical matrix operations.

### 4.7.2 Member Function Documentation

#### 4.7.2.1 colsize()

```
template<typename REAL >
size_t hdnum::DenseMatrix< REAL >::colsize ( ) const [inline]
```

get number of columns of the matrix

##### Example:

```
hdnum::DenseMatrix<double> A(4,5);
size_t nColumns = A.colsize();
std::cout << "Matrix A has " << nColumns << " columns." << std::endl;
```

##### Output:

Matrix A has 5 columns.

#### 4.7.2.2 mm()

```
template<typename REAL >
void hdnum::DenseMatrix< REAL >::mm (
    const DenseMatrix< REAL > & A,
    const DenseMatrix< REAL > & B ) [inline]
```

assign to matrix product  $C = A*B$  to matrix C

Implements  $C = A*B$  where A and B are matrices

##### Parameters

in	A	constant reference to a <a href="#">DenseMatrix</a>
in	B	constant reference to a <a href="#">DenseMatrix</a>

##### Example:

```
hdnum::DenseMatrix<double> A(2,6,1.0);
hdnum::DenseMatrix<double> B(6,3,-1.0);
A.scientific(false); // fixed point representation for all DenseMatrix
objects A.width(6); // use at least 6 columns for displaying
matrix entries A.precision(3); // display 3 digits behind the point
```

```
std::cout << "A =" << A << std::endl;
std::cout << "B =" << B << std::endl;
hdnum::DenseMatrix<double> C(2,3);
C.mm(A,B);
std::cout << "C = A*B =" << C << std::endl;
```

**Output:**

```
A =
0      1      2      3      4      5
0  1.000  1.000  1.000  1.000  1.000  1.000
1  1.000  1.000  1.000  1.000  1.000  1.000

B =
0      1      2
0 -1.000 -1.000 -1.000
1 -1.000 -1.000 -1.000
2 -1.000 -1.000 -1.000
3 -1.000 -1.000 -1.000
4 -1.000 -1.000 -1.000
5 -1.000 -1.000 -1.000

C = A*B =
0      1      2
0 -6.000 -6.000 -6.000
1 -6.000 -6.000 -6.000
```

**4.7.2.3 mv()**

```
template<typename REAL >
template<class V >
void hdnun::DenseMatrix< REAL >::mv (
    Vector< V > & y,
    const Vector< V > & x ) const [inline]
```

matrix vector product  $y = A*x$

Implements  $y = A*x$  where  $x$  and  $y$  are a vectors and  $A$  is a matrix

**Parameters**

in	$y$	reference to the resulting <a href="#">Vector</a>
in	$x$	constant reference to a <a href="#">Vector</a>

**Example:**

```
hdnum::Vector<double> x(3,10.0);
hdnum::Vector<double> y(2);
hdnum::DenseMatrix<double> A(2,3,1.0);
x.scientific(false); // fixed point representation for all Vector objects
A.scientific(false); // fixed point representation for all DenseMatrix
objects
std::cout << "A =" << A << std::endl;
std::cout << "x =" << x << std::endl;
A.mv(y,x);
std::cout << "y = A*x =" << y << std::endl;
```

**Output:**

```
A =
0      1      2
```

```

0      1.000      1.000      1.000
1      1.000      1.000      1.000

```

```

x =
[ 0]      10.0000000
[ 1]      10.0000000
[ 2]      10.0000000

```

```

y = A*x =
[ 0]      30.0000000
[ 1]      30.0000000

```

#### 4.7.2.4 operator()(i,j)

```

template<typename REAL >
REAL & hdnum::DenseMatrix< REAL >::operator() (
    const std::size_t row,
    const std::size_t col ) [inline]

```

(i,j)-operator for accessing entries of a (m x n)-matrix directly

##### Parameters

in	<i>row</i>	row index (0...m-1)
in	<i>col</i>	column index (0...n-1)

##### Example:

```

hdnum::DenseMatrix<double> A(4,4);
A.scientific(false); // fixed point representation for all DenseMatrix
objects A.width(8); A.precision(3);
identity(A); // Defines the identity matrix of the same dimension
std::cout << "A=" << A << std::endl;
std::cout << "reading A(0,0)=" << A(0,0) << std::endl;
std::cout << "resetting A(0,0) and A(2,3)..." << std::endl;
A(0,0) = 1.234;
A(2,3) = 432.1;
std::cout << "A=" << A << std::endl;

```

##### Output:

```

A=
0      1      2      3
0      1.000      0.000      0.000      0.000
1      0.000      1.000      0.000      0.000
2      0.000      0.000      1.000      0.000
3      0.000      0.000      0.000      1.000

```

```

reading A(0,0)=1.000
resetting A(0,0) and A(2,3)...

```

```

A=
0      1      2      3
0      1.234      0.000      0.000      0.000
1      0.000      1.000      0.000      0.000
2      0.000      0.000      1.000      432.100
3      0.000      0.000      0.000      1.000

```



## 4.7.2.5 operator\*() [1/2]

```
template<typename REAL >
DenseMatrix<hdnum::DenseMatrix< REAL >>::operator* (
    const DenseMatrix< REAL > & x ) const [inline]
```

matrix = matrix \* matrix

## Parameters

in	x	constant reference to a <a href="#">DenseMatrix</a>
----	---	---

## Example:

```
hdnum::DenseMatrix<double> A(3,3,2.0);
hdnum::DenseMatrix<double> B(3,3,4.0);
hdnum::DenseMatrix<double> C(3,3);
A.scientific(false); // fixed point representation for all DenseMatrix
objects A.width(8); A.precision(1);
std::cout << "A=" << A << std::endl;
std::cout << "B=" << B << std::endl;
C=A*B;
std::cout << "C=A*B=" << C << std::endl;
```

## Output:

```
A=
0      1      2
0      2.0    2.0    2.0
1      2.0    2.0    2.0
2      2.0    2.0    2.0

B=
0      1      2
0      4.0    4.0    4.0
1      4.0    4.0    4.0
2      4.0    4.0    4.0

C=A*B=
0      1      2
0      24.0   24.0   24.0
1      24.0   24.0   24.0
2      24.0   24.0   24.0
```

## 4.7.2.6 operator\*() [2/2]

```
template<typename REAL >
Vector< REAL > hdnum::DenseMatrix< REAL >::operator* (
    const Vector< REAL > & x ) const [inline]
```

vector = matrix \* vector

## Parameters

in	x	constant reference to a <a href="#">Vector</a>
----	---	--

## Example:

```
hdnum::Vector<double> x(3,4.0);
```

```

hdnum::DenseMatrix<double> A(3,3,2.0);
hdnum::Vector<double> y(3);
A.scientific(false); // fixed point representation for all DenseMatrix
objects A.width(8); A.precision(1);
x.scientific(false); // fixed point representation for all Vector objects
x.width(8);
x.precision(1);
std::cout << "A=" << A << std::endl;
std::cout << "x=" << x << std::endl;
y=A*x;
std::cout << "y=A*x" << y << std::endl;

```

### Output:

```

A=
0      1      2
0      2.0    2.0    2.0
1      2.0    2.0    2.0
2      2.0    2.0    2.0

x=
[ 0]    4.0
[ 1]    4.0
[ 2]    4.0

y=A*x
[ 0]    24.0
[ 1]    24.0
[ 2]    24.0

```

#### 4.7.2.7 operator\*=( )

```

template<typename REAL >
DenseMatrix & hdnum::DenseMatrix< REAL >::operator*= (
    const REAL s ) [inline]

```

Scalar multiplication assignment.

Implements  $A *= s$  where  $s$  is a scalar

#### Parameters

in	s	scalar value to multiply with
----	---	-------------------------------

#### Example:

```

double s = 0.5;
hdnum::DenseMatrix<double> A(2,3,1.0);
std::cout << "A=" << A << std::endl;
A *= s;
std::cout << "A=" << A << std::endl;

```

### Output:

```

A=
0      1      2
0      1.000e+00  1.000e+00  1.000e+00
1      1.000e+00  1.000e+00  1.000e+00

0.5*A =
0      1      2
0      5.000e-01  5.000e-01  5.000e-01
1      5.000e-01  5.000e-01  5.000e-01

```

## 4.7.2.8 operator+()

```
template<typename REAL >
DenseMatrix hdnum::DenseMatrix< REAL >::operator+ (
    const DenseMatrix< REAL > & x ) const [inline]
```

matrix = matrix + matrix

## Parameters

in	x	constant reference to a <a href="#">DenseMatrix</a>
----	---	---

## Example:

```
hdnum::DenseMatrix<double> A(3,3,2.0);
hdnum::DenseMatrix<double> B(3,3,4.0);
hdnum::DenseMatrix<double> C(3,3);
A.scientific(false); // fixed point representation for all DenseMatrix
objects A.width(8); A.precision(1);
std::cout << "A=" << A << std::endl;
std::cout << "B=" << B << std::endl;
C=A+B;
std::cout << "C=A+B=" << C << std::endl;
```

## Output:

```
A=
0      1      2
0      2.0    2.0    2.0
1      2.0    2.0    2.0
2      2.0    2.0    2.0

B=
0      1      2
0      4.0    4.0    4.0
1      4.0    4.0    4.0
2      4.0    4.0    4.0

C=A+B=
0      1      2
0      6.0    6.0    6.0
1      6.0    6.0    6.0
2      6.0    6.0    6.0
```

## 4.7.2.9 operator+=()

```
template<typename REAL >
DenseMatrix & hdnum::DenseMatrix< REAL >::operator+= (
    const DenseMatrix< REAL > & B ) [inline]
```

Addition assignment.

Implements  $A += B$  matrix addition

## Parameters

in	B	another Matrix
----	---	----------------

#### 4.7.2.10 operator-()

```
template<typename REAL >
DenseMatrix hdnum::DenseMatrix< REAL >::operator- (
    const DenseMatrix< REAL > & x ) const [inline]
```

matrix = matrix - matrix

##### Parameters

in	x	constant reference to a <a href="#">DenseMatrix</a>
----	---	---

##### Example:

```
hdnum::DenseMatrix<double> A(3,3,2.0);
hdnum::DenseMatrix<double> B(3,3,4.0);
hdnum::DenseMatrix<double> C(3,3);
A.scientific(false); // fixed point representation for all DenseMatrix
objects A.width(8); A.precision(1);
std::cout << "A=" << A << std::endl;
std::cout << "B=" << B << std::endl;
C=A-B;
std::cout << "C=A-B=" << C << std::endl;
```

##### Output:

```
A=
0      1      2
0      2.0      2.0      2.0
1      2.0      2.0      2.0
2      2.0      2.0      2.0

B=
0      1      2
0      4.0      4.0      4.0
1      4.0      4.0      4.0
2      4.0      4.0      4.0

C=A-B=
0      1      2
0      -2.0      -2.0      -2.0
1      -2.0      -2.0      -2.0
2      -2.0      -2.0      -2.0
```

#### 4.7.2.11 operator-=( )

```
template<typename REAL >
DenseMatrix & hdnum::DenseMatrix< REAL >::operator-= (
    const DenseMatrix< REAL > & B ) [inline]
```

Subtraction assignment.

Implements  $A -= B$  matrix subtraction

## Parameters

in	<i>B</i>	another matrix
----	----------	----------------

## 4.7.2.12 operator/=()

```
template<typename REAL >
DenseMatrix & hdnum::DenseMatrix< REAL >::operator/= (
    const REAL s ) [inline]
```

Scalar division assignment.

Implements  $A /= s$  where  $s$  is a scalar

## Parameters

in	<i>s</i>	scalar value to multiply with
----	----------	-------------------------------

## Example:

```
double s = 0.5;
hdnum::DenseMatrix<double> A(2,3,1.0);
std::cout << "A=" << A << std::endl;
A /= s;
std::cout << "A=" << A << std::endl;
```

## Output:

```
A=
0          1          2
0  1.000e+00  1.000e+00  1.000e+00
1  1.000e+00  1.000e+00  1.000e+00

A/0.5 =
0          1          2
0  2.000e+00  2.000e+00  2.000e+00
1  2.000e+00  2.000e+00  2.000e+00
```

## 4.7.2.13 operator=() [1/2]

```
template<typename REAL >
DenseMatrix & hdnum::DenseMatrix< REAL >::operator= (
    const DenseMatrix< REAL > & A ) [inline]
```

assignment operator

## Example:

```
hdnum::DenseMatrix<double> A(4,4);
spd(A);
hdnum::DenseMatrix<double> B(4,4);
B = A;
std::cout << "B=" << B << std::endl;
```

## Output:

```

B=
0          1          2          3
0  4.000e+00 -1.000e+00 -2.500e-01 -1.111e-01
1 -1.000e+00  4.000e+00 -1.000e+00 -2.500e-01
2 -2.500e-01 -1.000e+00  4.000e+00 -1.000e+00
3 -1.111e-01 -2.500e-01 -1.000e+00  4.000e+00

```

#### 4.7.2.14 operator=() [2/2]

```

template<typename REAL >
DenseMatrix & hdnum::DenseMatrix< REAL >::operator= (
    const REAL value ) [inline]

```

assignment from a scalar value

##### Example:

```

hdnum::DenseMatrix<double> A(2,3);
A = 5.432;
A.scientific(false); // fixed point representation for all DenseMatrix
objects A.width(8); A.precision(3); std::cout << "A=" << A << std::endl;

```

##### Output:

```

A=
0          1          2
0  5.432  5.432  5.432
1  5.432  5.432  5.432

```

#### 4.7.2.15 rowsize()

```

template<typename REAL >
size_t hdnum::DenseMatrix< REAL >::rowsize ( ) const [inline]

```

get number of rows of the matrix

##### Example:

```

hdnum::DenseMatrix<double> A(4,5);
size_t nRows = A.rowsize();
std::cout << "Matrix A has " << nRows << " rows." << std::endl;

```

##### Output:

```

Matrix A has 4 rows.

```

#### 4.7.2.16 sc()

```

template<typename REAL >
void hdnum::DenseMatrix< REAL >::sc (
    const Vector< REAL > & x,
    std::size_t k ) [inline]

```

set column: make x the k'th column of A

## Parameters

in	$x$	constant reference to a <a href="#">Vector</a>
in	$k$	number of the column of A to be set

## Example:

```

hdnum::Vector<double> x(2,434.0);
hdnum::DenseMatrix<double> A(2,6);
A.scientific(false); // fixed point representation for all DenseMatrix
objects A.width(8); A.precision(1);
std::cout << "original A=" << A << std::endl;
A.sc(x,3); // redefine fourth column of the matrix
std::cout << "modified A=" << A << std::endl;

```

## Output:

```

original A=
0      1      2      3      4      5
0      0.0     0.0     0.0     0.0     0.0     0.0
1      0.0     0.0     0.0     0.0     0.0     0.0

modified A=
0      1      2      3      4      5
0      0.0     0.0     0.0    434.0     0.0     0.0
1      0.0     0.0     0.0    434.0     0.0     0.0

```

## 4.7.2.17 scientific()

```

template<typename REAL >
void hdnnum::DenseMatrix< REAL >::scientific (
    bool b ) const [inline]

```

Switch between floating point (default=true) and fixed point (false) display.

## Example:

```

hdnum::DenseMatrix<double> A(4,4);
A.scientific(false); // fixed point representation for all DenseMatrix
objects A.width(8); A.precision(3); identity(A); // Defines the identity
matrix of the same dimension std::cout << "A=" << A << std::endl;

```

## Output:

```

A=
0      1      2      3
0    1.000    0.000    0.000    0.000
1    0.000    1.000    0.000    0.000
2    0.000    0.000    1.000    0.000
3    0.000    0.000    0.000    1.000

```

## 4.7.2.18 sr()

```

template<typename REAL >
void hdnnum::DenseMatrix< REAL >::sr (
    const Vector< REAL > & x,
    std::size_t k ) [inline]

```

set row: make x the k'th row of A

## Parameters

in	$x$	constant reference to a <a href="#">Vector</a>
in	$k$	number of the row of A to be set

## Example:

```

hdnum::Vector<double> x(3,434.0);
hdnum::DenseMatrix<double> A(3,3);
A.scientific(false); // fixed point representation for all DenseMatrix
objects A.width(8); A.precision(1);
std::cout << "original A=" << A << std::endl;
A.sr(x,1); // redefine second row of the matrix
std::cout << "modified A=" << A << std::endl;

```

## Output:

```

original A=
0      1      2
0      0.0    0.0    0.0
1      0.0    0.0    0.0
2      0.0    0.0    0.0

modified A=
0      1      2
0      0.0    0.0    0.0
1      434.0  434.0  434.0
2      0.0    0.0    0.0

```

## 4.7.2.19 sub()

```

template<typename REAL >
DenseMatrix hdnum::DenseMatrix< REAL >::sub (
    size_type i,
    size_type j,
    size_type rows,
    size_type cols ) [inline]

```

Submatrix extraction.

Returns a new matrix that is a subset of the components of the given matrix.

## Parameters

in	$i$	first row index of the new matrix
in	$j$	first column index of the new matrix
in	$rows$	row size of the new matrix, i.e. it has components $[i,i+rows-1]$
in	$cols$	column size of the new matrix, i.e. it has components $[j,j+cols-1]$

## 4.7.2.20 transpose()

```

template<typename REAL >
DenseMatrix hdnum::DenseMatrix< REAL >::transpose ( ) const [inline]

```



Transposition.

Return the transposed as a new matrix.

#### 4.7.2.21 umm()

```
template<typename REAL >
void hdnnum::DenseMatrix< REAL >::umm (
    const DenseMatrix< REAL > & A,
    const DenseMatrix< REAL > & B ) [inline]
```

add matrix product A\*B to matrix C

Implements  $C += A*B$  where A, B and C are matrices

##### Parameters

in	<i>A</i>	constant reference to a <a href="#">DenseMatrix</a>
in	<i>B</i>	constant reference to a <a href="#">DenseMatrix</a>

##### Example:

```
hdnnum::DenseMatrix<double> A(2,6,1.0);
hdnnum::DenseMatrix<double> B(6,3,-1.0);
hdnnum::DenseMatrix<double> C(2,3,0.5);
A.scientific(false); // fixed point representation for all DenseMatrix
objects A.width(6); A.precision(3);
std::cout << "C =" << C << std::endl;
std::cout << "A =" << A << std::endl;
std::cout << "B =" << B << std::endl;
C.umm(A,B);
std::cout << "C + A*B =" << C << std::endl;
```

##### Output:

```
C =
0      1      2
0  0.500  0.500  0.500
1  0.500  0.500  0.500
```

```
A =
0      1      2      3      4      5
0  1.000  1.000  1.000  1.000  1.000  1.000
1  1.000  1.000  1.000  1.000  1.000  1.000
```

```
B =
0      1      2
0 -1.000 -1.000 -1.000
1 -1.000 -1.000 -1.000
2 -1.000 -1.000 -1.000
3 -1.000 -1.000 -1.000
4 -1.000 -1.000 -1.000
5 -1.000 -1.000 -1.000
```

```
C + A*B =
0      1      2
0 -5.500 -5.500 -5.500
1 -5.500 -5.500 -5.500
```

**4.7.2.22 umv()** [1/2]

```
template<typename REAL >
template<class V >
void hdnum::DenseMatrix< REAL >::umv (
    Vector< V > & y,
    const V & s,
    const Vector< V > & x ) const [inline]
```

update matrix vector product  $y += sA*x$

Implements  $y += sA*x$  where  $s$  is a scalar value,  $x$  and  $y$  are a vectors and  $A$  is a matrix

**Parameters**

in	$y$	reference to the resulting <a href="#">Vector</a>
in	$s$	constant reference to a number type
in	$x$	constant reference to a <a href="#">Vector</a>

**Example:**

```
double s=0.5;
hdnum::Vector<double> x(3,10.0);
hdnum::Vector<double> y(2,5.0);
hdnum::DenseMatrix<double> A(2,3,1.0);
x.scientific(false); // fixed point representation for all Vector objects
A.scientific(false); // fixed point representation for all DenseMatrix
objects
std::cout << "y =" << y << std::endl;
std::cout << "A =" << A << std::endl;
std::cout << "x =" << x << std::endl;
A.umv(y,s,x);
std::cout << "y = s*A*x =" << y << std::endl;
```

**Output:**

```
y =
[ 0]      5.00000000
[ 1]      5.00000000

A =
0          1          2
0      1.000      1.000      1.000
1      1.000      1.000      1.000

x =
[ 0]      10.00000000
[ 1]      10.00000000
[ 2]      10.00000000

y = s*A*x =
[ 0]      20.00000000
[ 1]      20.00000000
```

**4.7.2.23 umv()** [2/2]

```
template<typename REAL >
template<class V >
void hdnum::DenseMatrix< REAL >::umv (
    Vector< V > & y,
    const Vector< V > & x ) const [inline]
```

update matrix vector product  $y += A*x$

Implements  $y += A*x$  where  $x$  and  $y$  are a vectors and  $A$  is a matrix

## Parameters

in	$y$	reference to the resulting <a href="#">Vector</a>
in	$x$	constant reference to a <a href="#">Vector</a>

## Example:

```

hdnum::Vector<double> x(3,10.0);
hdnum::Vector<double> y(2,5.0);
hdnum::DenseMatrix<double> A(2,3,1.0);
x.scientific(false); // fixed point representation for all Vector objects
A.scientific(false); // fixed point representation for all DenseMatrix
objects
std::cout << "y =" << y << std::endl;
std::cout << "A =" << A << std::endl;
std::cout << "x =" << x << std::endl;
A.umv(y,x);
std::cout << "y = A*x =" << y << std::endl;

```

## Output:

```

y =
[ 0]      5.0000000
[ 1]      5.0000000

A =
0          1          2
0      1.000      1.000      1.000
1      1.000      1.000      1.000

x =
[ 0]      10.0000000
[ 1]      10.0000000
[ 2]      10.0000000

y + A*x =
[ 0]      35.0000000
[ 1]      35.0000000

```

## 4.7.2.24 update()

```

template<typename REAL >
void hdnum::DenseMatrix< REAL >::update (
    const REAL s,
    const DenseMatrix< REAL > & B ) [inline]

```

Scaled update of a Matrix.

Implements  $A += s*B$  where  $s$  is a scalar and  $B$  a matrix

## Parameters

in	$s$	scalar value to multiply with
in	$B$	another matrix

## Example:

```

double s = 0.5;
hdnum::DenseMatrix<double> A(2,3,1.0);
hdnum::DenseMatrix<double> B(2,3,2.0);

```

```
A.update(s,B);
std::cout << "A + s*B =" << A << std::endl;
```

#### Output:

```
A + s*B =
0          1          2
0      1.500      1.500      1.500
1      1.500      1.500      1.500
```

## 4.7.3 Friends And Related Function Documentation

### 4.7.3.1 identity()

```
template<class T >
void identity (
    DenseMatrix< T > & A ) [related]
```

#### Function: make identity matrix

```
template<class T>
inline void identity (DenseMatrix<T> &A)
```

#### Parameters

in	A	reference to a <a href="#">DenseMatrix</a> that shall be filled with entries
----	---	--

#### Example:

```
hdnum::DenseMatrix<double> A(4,4);
identity(A);
A.scientific(false); // fixed point representation for all DenseMatrix objects
A.width(10);
A.precision(5);
std::cout << "A=" << A << std::endl;
```

#### Output:

```
A=
0          1          2          3
0      1.00000      0.00000      0.00000      0.00000
1      0.00000      1.00000      0.00000      0.00000
2      0.00000      0.00000      1.00000      0.00000
3      0.00000      0.00000      0.00000      1.00000
```

### 4.7.3.2 readMatrixFromFileDat()

```
template<typename REAL >
void readMatrixFromFileDat (
    const std::string & filename,
    DenseMatrix< REAL > & A ) [related]
```

Read matrix from a text file.

## Parameters

in	<i>filename</i>	name of the text file
in, out	<i>A</i>	reference to a <a href="#">DenseMatrix</a>

## Example:

```

hdnum::DenseMatrix<number> L;
readMatrixFromFile("matrixL.dat", L );
std::cout << "L=" << L << std::endl;

```

## Output:

Contents of "matrixL.dat":

```

1.000e+00  0.000e+00  0.000e+00
2.000e+00  1.000e+00  0.000e+00
3.000e+00  2.000e+00  1.000e+00

```

would give:

```

L=
      0      1      2
0  1.000e+00  0.000e+00  0.000e+00
1  2.000e+00  1.000e+00  0.000e+00
2  3.000e+00  2.000e+00  1.000e+00

```

## 4.7.3.3 readMatrixFromFileMatrixMarket()

```

template<typename REAL >
void readMatrixFromFileMatrixMarket (
    const std::string & filename,
    DenseMatrix< REAL > & A ) [related]

```

Read matrix from a matrix market file.

## Parameters

in	<i>filename</i>	name of the text file
in, out	<i>A</i>	reference to a <a href="#">DenseMatrix</a>

## Example:

```

hdnum::DenseMatrix<number> L;
readMatrixFromFile("matrixL.mtx", L );
std::cout << "L=" << L << std::endl;

```

## Output:

Contents of "matrixL.mtx":

```

3 3 6
1 1 1
2 1 2
2 2 1
3 1 3
3 2 2
3 3 1

```

would give:

```

L=
      0      1      2
0  1.000e+00  0.000e+00  0.000e+00
1  2.000e+00  1.000e+00  0.000e+00
2  3.000e+00  2.000e+00  1.000e+00

```

```

0  1.000e+00  0.000e+00  0.000e+00
1  2.000e+00  1.000e+00  0.000e+00
2  3.000e+00  2.000e+00  1.000e+00

```

#### 4.7.3.4 spd()

```

template<typename REAL >
void spd (
    DenseMatrix< REAL > & A ) [related]

```

**Function:** make a symmetric and positive definite matrix

```

template<typename REAL>
inline void spd (DenseMatrix<REAL> &A)

```

##### Parameters

in	A	reference to a <a href="#">DenseMatrix</a> that shall be filled with entries
----	---	--

##### Example:

```

hdnum::DenseMatrix<double> A(4,4);
spd(A);
A.scientific(false); // fixed point representation for all DenseMatrix objects
A.width(10);
A.precision(5);
std::cout << "A=" << A << std::endl;

```

##### Output:

```

A=
0      1      2      3
0  4.00000 -1.00000 -0.25000 -0.11111
1 -1.00000  4.00000 -1.00000 -0.25000
2 -0.25000 -1.00000  4.00000 -1.00000
3 -0.11111 -0.25000 -1.00000  4.00000

```

#### 4.7.3.5 vandermonde()

```

template<typename REAL >
void vandermonde (
    DenseMatrix< REAL > & A,
    const Vector< REAL > x ) [related]

```

**Function:** make a vandermonde matrix

```

template<typename REAL>
inline void vandermonde (DenseMatrix<REAL> &A, const Vector<REAL> x)

```

##### Parameters

in	A	reference to a <a href="#">DenseMatrix</a> that shall be filled with entries
in	x	constant reference to a <a href="#">Vector</a>

**Example:**

```

hdnum::Vector<double> x(4);
fill(x,2.0,1.0);
hdnum::DenseMatrix<double> A(4,4);
vandermonde(A,x);
A.scientific(false); // fixed point representation for all DenseMatrix objects
A.width(10);
A.precision(5);
x.scientific(false); // fixed point representation for all Vector objects
x.width(10);
x.precision(5);
std::cout << "x=" << x << std::endl;
std::cout << "A=" << A << std::endl;

```

**Output:**

```

x=
[ 0]  2.00000
[ 1]  3.00000
[ 2]  4.00000
[ 3]  5.00000

A=
      1      2      3
0  1.00000  2.00000  4.00000  8.00000
1  1.00000  3.00000  9.00000 27.00000
2  1.00000  4.00000 16.00000 64.00000
3  1.00000  5.00000 25.00000 125.00000

```

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

- `src/densematrix.hh`

## 4.8 `hdnum::DIRK< M, S >` Class Template Reference

Implementation of a general Diagonal Implicit Runge-Kutta method.

```
#include <ode.hh>
```

**Public Types**

- `typedef M::size_type size_type`  
*export size\_type*
- `typedef M::time_type time_type`  
*export time\_type*
- `typedef M::number_type number_type`  
*export number\_type*
- `typedef DenseMatrix< number_type > ButcherTableau`  
*the type of a Butcher tableau*

## Public Member Functions

- [DIRK](#) (const M &model\_, const S &newton\_, const [ButcherTableau](#) &butcher\_, const int order\_)
- [DIRK](#) (const M &model\_, const S &newton\_, const std::string method)
- void [set\\_dt](#) ([time\\_type](#) dt\_)  
*set time step for subsequent steps*
- void [set\\_verbosity](#) ([size\\_type](#) verbosity\_)  
*set verbosity level*
- void [step](#) ()  
*do one step*
- bool [get\\_error](#) () const  
*get current state*
- void [set\\_state](#) ([time\\_type](#) t\_, const [Vector](#)< [number\\_type](#) > &u\_)  
*set current state*
- const [Vector](#)< [number\\_type](#) > & [get\\_state](#) () const  
*get current state*
- [time\\_type](#) [get\\_time](#) () const  
*get current time*
- [time\\_type](#) [get\\_dt](#) () const  
*get dt used in last step (i.e. to compute current state)*
- [size\\_type](#) [get\\_order](#) () const  
*return consistency order of the method*
- void [get\\_info](#) () const  
*print some information*

### 4.8.1 Detailed Description

```
template<class M, class S>
class hdnum::DIRK< M, S >
```

Implementation of a general Diagonal Implicit Runge-Kutta method.

The ODE solver is parametrized by a model. The model also exports all relevant types for time and states. The ODE solver encapsulates the states needed for the computation.

#### Template Parameters

<i>M</i>	the model type
<i>S</i>	nonlinear solver

### 4.8.2 Constructor & Destructor Documentation

#### 4.8.2.1 [DIRK\(\)](#) [1/2]

```
template<class M , class S >
hdnum::DIRK< M, S >::DIRK (
```



```

    const M & model_,
    const S & newton_,
    const ButcherTableau & butcher_,
    const int order_ ) [inline]

```

constructor stores reference to the model and requires a butcher tableau

#### 4.8.2.2 DIRK() [2/2]

```

template<class M , class S >
hdnum::DIRK< M, S >::DIRK (
    const M & model_,
    const S & newton_,
    const std::string method ) [inline]

```

constructor stores reference to the model and sets the default butcher tableau corresponding to the given order

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

- src/ode.hh

## 4.9 hdnum::EE< M > Class Template Reference

Explicit Euler method as an example for an ODE solver.

```
#include <ode.hh>
```

### Public Types

- typedef M::size\_type **size\_type**  
*export size\_type*
- typedef M::time\_type **time\_type**  
*export time\_type*
- typedef M::number\_type **number\_type**  
*export number\_type*

### Public Member Functions

- **EE** (const M &model\_)  
*constructor stores reference to the model*
- void **set\_dt** (time\_type dt\_)  
*set time step for subsequent steps*
- void **step** ()  
*do one step*
- void **set\_state** (time\_type t\_, const Vector< number\_type > &u\_)  
*set current state*
- const Vector< number\_type > &**get\_state** () const  
*get current state*
- time\_type **get\_time** () const  
*get current time*
- time\_type **get\_dt** () const  
*get dt used in last step (i.e. to compute current state)*
- size\_type **get\_order** () const  
*return consistency order of the method*

### 4.9.1 Detailed Description

```
template<class M>
class hdnum::EE< M >
```

Explicit Euler method as an example for an ODE solver.

The ODE solver is parametrized by a model. The model also exports all relevant types for time and states. The ODE solver encapsulates the states needed for the computation.

#### Template Parameters

<i>M</i>	the model type
----------	----------------

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

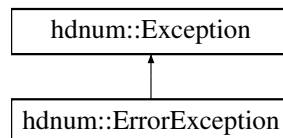
- [src/ode.hh](#)

## 4.10 hdnum::ErrorException Class Reference

General Error.

```
#include <exceptions.hh>
```

Inheritance diagram for hdnum::ErrorException:



### Additional Inherited Members

#### 4.10.1 Detailed Description

General Error.

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

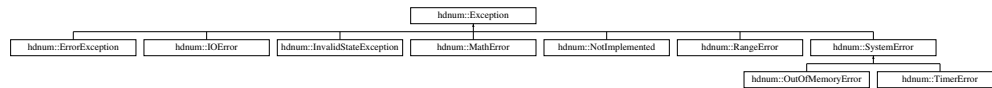
- [src/exceptions.hh](#)

## 4.11 hdnum::Exception Class Reference

Base class for Exceptions.

```
#include <exceptions.hh>
```

Inheritance diagram for hdnum::Exception:



### Public Member Functions

- void **message** (const std::string &message)  
*store string in internal message buffer*
- const std::string & **what** () const  
*output internal message buffer*

#### 4.11.1 Detailed Description

Base class for Exceptions.

all HDNUM exceptions are derived from this class via trivial subclassing:

```
class MyException : public Dune::Exception {};
```

You should not `throw` a `Dune::Exception` directly but use the macro `DUNE_THROW()` instead which fills the message-buffer of the exception in a standard way and features a way to pass the result in the operator<<-style

See also

[HDNUM\\_THROW](#), [IOError](#), [MathError](#)

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

- [src/exceptions.hh](#)

## 4.12 hdnum::GenericNonlinearProblem< Lambda, Vec > Class Template Reference

A generic problem class that can be set up with a lambda defining  $F(x)=0$ .

```
#include <newton.hh>
```

## Public Types

- typedef std::size\_t **size\_type**  
*export size\_type*
- typedef Vec::value\_type **number\_type**  
*export number\_type*

## Public Member Functions

- **GenericNonlinearProblem** (const Lambda &l\_, const Vec &x\_, [number\\_type](#) eps\_=1e-7)  
*constructor stores parameter lambda*
- std::size\_t **size** () const  
*return number of componentes for the model*
- void **F** (const Vec &x, Vec &result) const  
*model evaluation*
- void **F\_x** (const Vec &x, [DenseMatrix](#)< [number\\_type](#) > &result) const  
*jacobian evaluation needed for implicit solvers*

### 4.12.1 Detailed Description

```
template<typename Lambda, typename Vec>
class hdnum::GenericNonlinearProblem< Lambda, Vec >
```

A generic problem class that can be set up with a lambda defining  $F(x)=0$ .

#### Template Parameters

<i>Lambda</i>	mapping a <a href="#">Vector</a> to a <a href="#">Vector</a>
<i>Vec</i>	the type for the <a href="#">Vector</a>

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

- src/[newton.hh](#)

## 4.13 hdnum::Heun2< M > Class Template Reference

Heun method (order 2 with 2 stages)

```
#include <ode.hh>
```

## Public Types

- typedef M::size\_type **size\_type**  
*export size\_type*
- typedef M::time\_type **time\_type**  
*export time\_type*
- typedef M::number\_type **number\_type**  
*export number\_type*

## Public Member Functions

- **Heun2** (const M &model\_)  
*constructor stores reference to the model*
- void **set\_dt** (time\_type dt\_)  
*set time step for subsequent steps*
- void **step** ()  
*do one step*
- void **set\_state** (time\_type t\_, const Vector< number\_type > &u\_)  
*set current state*
- const Vector< number\_type > & **get\_state** () const  
*get current state*
- time\_type **get\_time** () const  
*get current time*
- time\_type **get\_dt** () const  
*get dt used in last step (i.e. to compute current state)*
- size\_type **get\_order** () const  
*return consistency order of the method*

### 4.13.1 Detailed Description

```
template<class M>
class hdnum::Heun2< M >
```

Heun method (order 2 with 2 stages)

The ODE solver is parametrized by a model. The model also exports all relevant types for time and states. The ODE solver encapsulates the states needed for the computation.

Template Parameters

<i>M</i>	the model type
----------	----------------

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

- src/ode.hh

## 4.14 hdnum::Heun3< M > Class Template Reference

Heun method (order 3 with 3 stages)

```
#include <ode.hh>
```

### Public Types

- typedef M::size\_type **size\_type**  
*export size\_type*
- typedef M::time\_type **time\_type**  
*export time\_type*
- typedef M::number\_type **number\_type**  
*export number\_type*

## Public Member Functions

- **Heun3** (const M &model\_)  
*constructor stores reference to the model*
- void **set\_dt** (time\_type dt\_)  
*set time step for subsequent steps*
- void **step** ()  
*do one step*
- void **set\_state** (time\_type t\_, const Vector< number\_type > &u\_)  
*set current state*
- const Vector< number\_type > & **get\_state** () const  
*get current state*
- time\_type **get\_time** () const  
*get current time*
- time\_type **get\_dt** () const  
*get dt used in last step (i.e. to compute current state)*
- size\_type **get\_order** () const  
*return consistency order of the method*

### 4.14.1 Detailed Description

```
template<class M>
class hdnum::Heun3< M >
```

Heun method (order 3 with 3 stages)

The ODE solver is parametrized by a model. The model also exports all relevant types for time and states. The ODE solver encapsulates the states needed for the computation.

Template Parameters

<i>M</i>	the model type
----------	----------------

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

- src/ode.hh

## 4.15 hdnum::IE< M, S > Class Template Reference

Implicit Euler using [Newton](#)'s method to solve nonlinear system.

```
#include <ode.hh>
```

### Public Types

- typedef M::size\_type **size\_type**  
*export size\_type*
- typedef M::time\_type **time\_type**  
*export time\_type*
- typedef M::number\_type **number\_type**  
*export number\_type*

## Public Member Functions

- **IE** (const M &model\_, const S &newton\_)  
*constructor stores reference to the model*
- void **set\_dt** (time\_type dt\_)  
*set time step for subsequent steps*
- void **set\_verbosity** (size\_type verbosity\_)  
*set verbosity level*
- void **step** ()  
*do one step*
- bool **get\_error** () const  
*get current state*
- void **set\_state** (time\_type t\_, const Vector< number\_type > &u\_)  
*set current state*
- const Vector< number\_type > &**get\_state** () const  
*get current state*
- time\_type **get\_time** () const  
*get current time*
- time\_type **get\_dt** () const  
*get dt used in last step (i.e. to compute current state)*
- size\_type **get\_order** () const  
*return consistency order of the method*
- void **get\_info** () const  
*print some information*

### 4.15.1 Detailed Description

```
template<class M, class S>
class hdnum::IE< M, S >
```

Implicit Euler using [Newton](#)'s method to solve nonlinear system.

The ODE solver is parametrized by a model. The model also exports all relevant types for time and states. The ODE solver encapsulates the states needed for the computation.

#### Template Parameters

<i>M</i>	the model type
<i>S</i>	nonlinear solver

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

- [src/ode.hh](#)

## 4.16 hdnum::ImplicitRungeKuttaStepProblem< M > Class Template Reference

Nonlinear problem we need to solve to do one step of an implicit Runge Kutta method.

```
#include <rungekutta.hh>
```

## Public Types

- typedef M::size\_type **size\_type**  
*export size\_type*
- typedef M::time\_type **time\_type**  
*export time\_type*
- typedef M::number\_type **number\_type**  
*export number\_type*

## Public Member Functions

- **ImplicitRungeKuttaStepProblem** (const M &model\_, DenseMatrix< number\_type > A\_, Vector< number\_type > b\_, Vector< number\_type > c\_, time\_type t\_, Vector< number\_type > u\_, time\_type dt\_)  
*constructor stores parameter lambda*
- std::size\_t **size** () const  
*return number of componentes for the model*
- void **F** (const Vector< number\_type > &x, Vector< number\_type > &result) const  
*model evaluation*
- void **F\_x** (const Vector< number\_type > &x, DenseMatrix< number\_type > &result) const  
*jacobian evaluation needed for newton in impicite solvers*

### 4.16.1 Detailed Description

```
template<class M>
class hdnum::ImplicitRungeKuttaStepProblem< M >
```

Nonlinear problem we need to solve to do one step of an implicit Runge Kutta method.

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

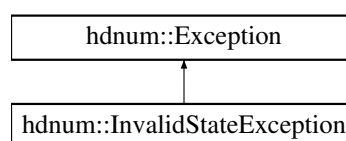
- src/[rungekutta.hh](#)

## 4.17 hdnum::InvalidStateException Class Reference

Default exception if a function was called while the object is not in a valid state for that function.

```
#include <exceptions.hh>
```

Inheritance diagram for hdnum::InvalidStateException:





## Additional Inherited Members

### 4.17.1 Detailed Description

Default exception if a function was called while the object is not in a valid state for that function.

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

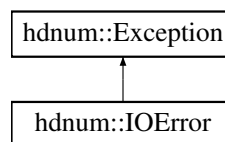
- [src/exceptions.hh](#)

## 4.18 hdnum::IOException Class Reference

Default exception class for I/O errors.

```
#include <exceptions.hh>
```

Inheritance diagram for hdnum::IOException:



## Additional Inherited Members

### 4.18.1 Detailed Description

Default exception class for I/O errors.

This is a superclass for any errors dealing with file/socket I/O problems like

- file not found
- could not write file
- could not connect to remote socket

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

- [src/exceptions.hh](#)

## 4.19 hdnum::Kutta3< M > Class Template Reference

Kutta method (order 3 with 3 stages)

```
#include <ode.hh>
```

## Public Types

- typedef M::size\_type **size\_type**  
*export size\_type*
- typedef M::time\_type **time\_type**  
*export time\_type*
- typedef M::number\_type **number\_type**  
*export number\_type*

## Public Member Functions

- **Kutta3** (const M &model\_)  
*constructor stores reference to the model*
- void **set\_dt** (time\_type dt\_)  
*set time step for subsequent steps*
- void **step** ()  
*do one step*
- void **set\_state** (time\_type t\_, const Vector< number\_type > &u\_)  
*set current state*
- const Vector< number\_type > &**get\_state** () const  
*get current state*
- time\_type **get\_time** () const  
*get current time*
- time\_type **get\_dt** () const  
*get dt used in last step (i.e. to compute current state)*
- size\_type **get\_order** () const  
*return consistency order of the method*

### 4.19.1 Detailed Description

```
template<class M>
class hdnum::Kutta3< M >
```

Kutta method (order 3 with 3 stages)

The ODE solver is parametrized by a model. The model also exports all relevant types for time and states. The ODE solver encapsulates the states needed for the computation.

#### Template Parameters

<i>M</i>	the model type
----------	----------------

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

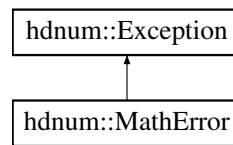
- src/ode.hh

## 4.20 hdnum::MathError Class Reference

Default exception class for mathematical errors.

```
#include <exceptions.hh>
```

Inheritance diagram for `hdnum::MathError`:



## Additional Inherited Members

### 4.20.1 Detailed Description

Default exception class for mathematical errors.

This is the superclass for all errors which are caused by mathematical problems like

- matrix not invertible
- not convergent

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

- [src/exceptions.hh](#)

## 4.21 `hdnum::ModifiedEuler< M >` Class Template Reference

Modified Euler method (order 2 with 2 stages)

```
#include <ode.hh>
```

### Public Types

- `typedef M::size_type size_type`  
*export size\_type*
- `typedef M::time_type time_type`  
*export time\_type*
- `typedef M::number_type number_type`  
*export number\_type*

## Public Member Functions

- **ModifiedEuler** (const M &model\_)  
*constructor stores reference to the model*
- void **set\_dt** (time\_type dt\_)  
*set time step for subsequent steps*
- void **step** ()  
*do one step*
- void **set\_state** (time\_type t\_, const Vector< number\_type > &u\_)  
*set current state*
- const Vector< number\_type > & **get\_state** () const  
*get current state*
- time\_type **get\_time** () const  
*get current time*
- time\_type **get\_dt** () const  
*get dt used in last step (i.e. to compute current state)*
- size\_type **get\_order** () const  
*return consistency order of the method*

### 4.21.1 Detailed Description

```
template<class M>
class hdnum::ModifiedEuler< M >
```

Modified Euler method (order 2 with 2 stages)

The ODE solver is parametrized by a model. The model also exports all relevant types for time and states. The ODE solver encapsulates the states needed for the computation.

#### Template Parameters

<i>M</i>	the model type
----------	----------------

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

- src/ode.hh

## 4.22 hdnum::Newton Class Reference

Solve nonlinear problem using a damped [Newton](#) method.

```
#include <newton.hh>
```

## Public Member Functions

- **Newton** ()  
*constructor stores reference to the model*
- void **set\_maxit** (size\_type n)  
*maximum number of iterations before giving up*
- void **set\_sigma** (double sigma\_)
- void **set\_linesearchsteps** (size\_type n)  
*maximum number of steps in linesearch before giving up*
- void **set\_verbosity** (size\_type n)  
*control output given 0=nothing, 1=summary, 2=every step, 3=include line search*
- void **set\_abslimit** (double l)  
*basolute limit for defect*
- void **set\_reduction** (double l)  
*reduction factor*
- template<class M >  
void **solve** (const M &model, [Vector](#)< typename M::number\_type > &x) const  
*do one step*
- bool **has\_converged** () const
- size\_type **iterations** () const

### 4.22.1 Detailed Description

Solve nonlinear problem using a damped [Newton](#) method.

The [Newton](#) solver is parametrized by a model. The model also exports all relevant types for types.

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

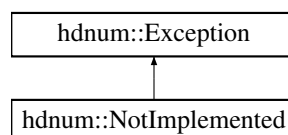
- [src/newton.hh](#)

## 4.23 hdnum::NotImplemented Class Reference

Default exception for dummy implementations.

```
#include <exceptions.hh>
```

Inheritance diagram for hdnum::NotImplemented:



## Additional Inherited Members

### 4.23.1 Detailed Description

Default exception for dummy implementations.

This exception can be used for functions/methods

- that have to be implemented but should never be called
- that are missing

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

- [src/exceptions.hh](#)

## 4.24 hdnum::oc::OpCounter< F > Class Template Reference

```
#include <opcounter.hh>
```

### Classes

- struct [Counters](#)  
*Struct storing the number of operations.*

### Public Types

- using **size\_type** = std::size\_t
- using **value\_type** = F

### Public Member Functions

- template<typename T >  
**OpCounter** (const T &t, typename std::enable\_if< std::is\_same< T, int >::value and !std::is\_same< F, int >::value >::type \* = nullptr)
- **OpCounter** (const F &f)
- **OpCounter** (F &&f)
- **OpCounter** (const char \*s)
- [OpCounter](#) & **operator=** (const char \*s)
- **operator F** () const
- [OpCounter](#) & **operator=** (const F &f)
- [OpCounter](#) & **operator=** (F &&f)
- F \* **data** ()
- const F \* **data** () const

## Static Public Member Functions

- static void **additions** (std::size\_t n)
- static void **multiplications** (std::size\_t n)
- static void **divisions** (std::size\_t n)
- static void **reset** ()
- template<typename Stream >  
static void **reportOperations** (Stream &os, bool doReset=false)  
*Report operations to stream object.*
- static size\_type **totalOperationCount** (bool doReset=false)  
*Return total number of operations.*

## Public Attributes

- F\_v

## Static Public Attributes

- static [Counters](#) counters

## Friends

- std::ostream & **operator**<< (std::ostream &os, const [OpCounter](#) &f)
- std::istream & **operator**>> (std::istream &iss, [OpCounter](#) &f)

### 4.24.1 Detailed Description

```
template<typename F>
class hdnum::oc::OpCounter< F >
```

Class counting operations

This is done by overloading operations and storing the numbers in a static class member.

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

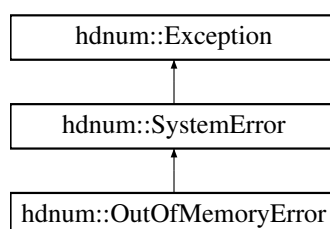
- [src/opcounter.hh](#)

## 4.25 hdnum::OutOfMemoryError Class Reference

Default exception if memory allocation fails.

```
#include <exceptions.hh>
```

Inheritance diagram for hdnum::OutOfMemoryError:



## Additional Inherited Members

### 4.25.1 Detailed Description

Default exception if memory allocation fails.

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

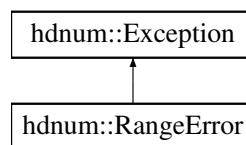
- [src/exceptions.hh](#)

## 4.26 hdnum::RangeError Class Reference

Default exception class for range errors.

```
#include <exceptions.hh>
```

Inheritance diagram for hdnum::RangeError:



## Additional Inherited Members

### 4.26.1 Detailed Description

Default exception class for range errors.

This is the superclass for all errors which are caused because the user tries to access data that was not allocated before. These can be problems like

- accessing array entries behind the last entry
- adding the fourth non zero entry in a sparse matrix with only three non zero entries per row

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

- [src/exceptions.hh](#)

## 4.27 hdnum::RE< M, S > Class Template Reference

Adaptive one-step method using Richardson extrapolation.

```
#include <ode.hh>
```



## Public Types

- typedef M::size\_type **size\_type**  
*export size\_type*
- typedef M::time\_type **time\_type**  
*export time\_type*
- typedef M::number\_type **number\_type**  
*export number\_type*

## Public Member Functions

- **RE** (const M &model\_, S &solver\_)  
*constructor stores reference to the model*
- void **set\_dt** (time\_type dt\_)  
*set time step for subsequent steps*
- void **set\_TOL** (time\_type TOL\_)  
*set tolerance for adaptive computation*
- void **step** ()  
*do one step*
- const Vector< number\_type > &**get\_state** () const  
*get current state*
- time\_type **get\_time** () const  
*get current time*
- time\_type **get\_dt** () const  
*get dt used in last step (i.e. to compute current state)*
- size\_type **get\_order** () const  
*return consistency order of the method*
- void **get\_info** () const  
*print some information*

### 4.27.1 Detailed Description

```
template<class M, class S>
class hdnum::RE< M, S >
```

Adaptive one-step method using Richardson extrapolation.

#### Template Parameters

<i>M</i>	a model
<i>S</i>	any of the (non-adaptive) one step methods (solving model M)

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

- src/ode.hh

## 4.28 hdnum::RKF45< M > Class Template Reference

Adaptive Runge-Kutta-Fehlberg method.

```
#include <ode.hh>
```

### Public Types

- typedef M::size\_type **size\_type**  
*export size\_type*
- typedef M::time\_type **time\_type**  
*export time\_type*
- typedef M::number\_type **number\_type**  
*export number\_type*

### Public Member Functions

- **RKF45** (const M &model\_)  
*constructor stores reference to the model*
- void **set\_dt** (time\_type dt\_)  
*set time step for subsequent steps*
- void **set\_TOL** (time\_type TOL\_)  
*set tolerance for adaptive computation*
- void **step** ()  
*do one step*
- const Vector< number\_type > & **get\_state** () const  
*get current state*
- time\_type **get\_time** () const  
*get current time*
- time\_type **get\_dt** () const  
*get dt used in last step (i.e. to compute current state)*
- size\_type **get\_order** () const  
*return consistency order of the method*
- void **get\_info** () const  
*print some information*

### 4.28.1 Detailed Description

```
template<class M>
class hdnum::RKF45< M >
```

Adaptive Runge-Kutta-Fehlberg method.

#### Template Parameters

<i>M</i>	the model type
----------	----------------

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

- [src/ode.hh](#)

## 4.29 `hdnum::SparseMatrix< REAL >::row_iterator` Class Reference

### Public Types

- using **self\_type** = [row\\_iterator](#)
- using **difference\_type** = `std::ptrdiff_t`
- using **value\_type** = [self\\_type](#)
- using **pointer** = [self\\_type](#) \*
- using **reference** = [self\\_type](#) &
- using **iterator\_category** = `std::random_access_iterator_tag`

### Public Member Functions

- **row\_iterator** (`std::vector< size\_type >::iterator rowPtrIter, std::vector< size\_type >::iterator colIndicesIter, typename std::vector< REAL >::iterator valIter`)
- **column\_iterator** **begin** ()
- **column\_iterator** **end** ()
- **column\_index\_iterator** **ibegin** ()
- **column\_index\_iterator** **iend** ()
- [self\\_type](#) & **operator++** ()
- [self\\_type](#) **operator++** (int junk)
- [self\\_type](#) & **operator+=** (difference\_type offset)
- [self\\_type](#) & **operator-=** (difference\_type offset)
- [self\\_type](#) **operator-** (difference\_type offset)
- [self\\_type](#) **operator+** (difference\_type offset)
- [reference](#) **operator[]** (difference\_type offset)
- bool **operator<** (const [self\\_type](#) &other)
- bool **operator>** (const [self\\_type](#) &other)
- [self\\_type](#) & **operator\*** ()
- bool **operator==** (const [self\\_type](#) &rhs)
- bool **operator!=** (const [self\\_type](#) &rhs)

### Friends

- [self\\_type](#) **operator+** (const difference\_type &offset, const [self\\_type](#) &sec)

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

- [src/sparsematrix.hh](#)

## 4.30 `hdnum::RungeKutta< M, S >` Class Template Reference

classical Runge-Kutta method (order n with n stages)

```
#include <rungekutta.hh>
```

## Public Types

- typedef M::size\_type **size\_type**  
*export size\_type*
- typedef M::time\_type **time\_type**  
*export time\_type*
- typedef M::number\_type **number\_type**  
*export number\_type*

## Public Member Functions

- **RungeKutta** (const M &model\_, DenseMatrix< number\_type > A\_, Vector< number\_type > b\_, Vector< number\_type > c\_)  
*constructor stores reference to the model*
- **RungeKutta** (const M &model\_, DenseMatrix< number\_type > A\_, Vector< number\_type > b\_, Vector< number\_type > c\_, number\_type sigma\_)  
*constructor stores reference to the model*
- void **set\_dt** (time\_type dt\_)  
*set time step for subsequent steps*
- bool **check\_explicit** ()  
*test if method is explicit*
- void **step** ()  
*do one step*
- void **set\_state** (time\_type t\_, const Vector< number\_type > &u\_)  
*set current state*
- const Vector< number\_type > &**get\_state** () const  
*get current state*
- time\_type **get\_time** () const  
*get current time*
- time\_type **get\_dt** () const  
*get dt used in last step (i.e. to compute current state)*
- void **set\_verbosity** (int verbosity\_)  
*how much should the ODE solver talk*

### 4.30.1 Detailed Description

```
template<class M, class S = Newton>
class hdnum::RungeKutta< M, S >
```

classical Runge-Kutta method (order n with n stages)

The ODE solver is parametrized by a model. The model also exports all relevant types for time and states. The ODE solver encapsulates the states needed for the computation.

#### Template Parameters

<i>M</i>	The model type
<i>S</i>	(Nonlinear) solver (default is <a href="#">Newton</a> )

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

- `src/rungekutta.hh`

## 4.31 `hdnum::RungeKutta4< M >` Class Template Reference

classical Runge-Kutta method (order 4 with 4 stages)

```
#include <ode.hh>
```

### Public Types

- `typedef M::size_type size_type`  
*export size\_type*
- `typedef M::time_type time_type`  
*export time\_type*
- `typedef M::number_type number_type`  
*export number\_type*

### Public Member Functions

- **`RungeKutta4`** (`const M &model_`)  
*constructor stores reference to the model*
- `void set_dt (time_type dt_)`  
*set time step for subsequent steps*
- `void step ()`  
*do one step*
- `void set_state (time_type t_, const Vector< number_type > &u_)`  
*set current state*
- `const Vector< number_type > & get_state () const`  
*get current state*
- `time_type get_time () const`  
*get current time*
- `time_type get_dt () const`  
*get dt used in last step (i.e. to compute current state)*
- `size_type get_order () const`  
*return consistency order of the method*

#### 4.31.1 Detailed Description

```
template<class M>
class hdnum::RungeKutta4< M >
```

classical Runge-Kutta method (order 4 with 4 stages)

The ODE solver is parametrized by a model. The model also exports all relevant types for time and states. The ODE solver encapsulates the states needed for the computation.

## Template Parameters

<i>M</i>	the model type
----------	----------------

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

- [src/ode.hh](#)

## 4.32 `hdnum::SGrid< N, DF, dimension >` Class Template Reference

Structured Grid for Finite Differences.

```
#include <sgrid.hh>
```

### Public Types

- enum { **dim** = dimension }
- typedef std::size\_t **size\_type**  
*Export size type.*
- typedef N **number\_type**  
*Export number type.*
- typedef DF **DomainFunction**  
*Type of the function defining the domain.*

### Public Member Functions

- **SGrid** (const [Vector](#)< [number\\_type](#) > extent\_, const [Vector](#)< [size\\_type](#) > size\_, const [DomainFunction](#) &df\_)  
*Constructor.*
- [size\\_type](#) **getNeighborIndex** (const [size\\_type](#) ln, const [size\\_type](#) n\_dim, const int n\_side, const int k=1) const  
*Provides the index of the k-th neighbor of the node with index ln.*
- bool **isBoundaryNode** (const [size\\_type](#) ln) const  
*Returns true if the node is on the boundary of the discrete computational domain.*
- [size\\_type](#) **getNumberOfNodes** () const  
*Returns the number of nodes which are in the computational domain.*
- [Vector](#)< [size\\_type](#) > **getGridSize** () const
- [Vector](#)< [number\\_type](#) > **getCellWidth** () const  
*Returns the cell width h of the structured grid.*
- [Vector](#)< [number\\_type](#) > **getCoordinates** (const [size\\_type](#) ln) const  
*Returns the world coordinates of the node with the given node index.*
- std::vector< [Vector](#)< [number\\_type](#) > > **getNodeCoordinates** () const

### Public Attributes

- const [size\\_type](#) **invalid\_node**  
*The value which is returned to indicate an invalid node.*

## Static Public Attributes

- static const int **positive** = 1  
*Side definitions for usage in getNeighborIndex(..)*
- static const int **negative** = -1

### 4.32.1 Detailed Description

```
template<class N, class DF, int dimension>
class hdnum::SGrid< N, DF, dimension >
```

Structured Grid for Finite Differences.

#### Template Parameters

<i>N</i>	A continuous type representing coordinate values.
<i>DF</i>	A boolean function which defines the domain.
<i>dimension</i>	The grid dimension.

### 4.32.2 Constructor & Destructor Documentation

#### 4.32.2.1 SGrid()

```
template<class N , class DF , int dimension>
hdnum::SGrid< N, DF, dimension >::SGrid (
    const Vector< number_type > extent_,
    const Vector< size_type > size_,
    const DomainFunction & df_ ) [inline]
```

Constructor.

#### Parameters

in	<i>extent_↔</i> —	The extent of the grid domain. The actual computational domain may be smaller and is defined by the domain function df_.
in	<i>size_↔</i> —	The number of nodes in each grid dimension.
in	<i>df_</i>	The domain function. It has to provide a boolean function evaluate(Vector<number_type> x) which returns true if the node which is positioned at the coordinates of x is within the computational domain.

### 4.32.3 Member Function Documentation

#### 4.32.3.1 getNeighborIndex()

```
template<class N , class DF , int dimension>
size_type hdnum::SGrid< N, DF, dimension >::getNeighborIndex (
    const size_type ln,
    const size_type n_dim,
    const int n_side,
    const int k = 1 ) const [inline]
```

Provides the index of the k-th neighbor of the node with index ln.

##### Parameters

in	<i>ln</i>	Index of the node whose neighbor is to be determined.
in	<i>n_dim</i>	The axes which connects the node and its neighbor (e.g. n_dim = 0 for a neighbor in the direction of the x-axes)
in	<i>n_side</i>	Determines whether the neighbor is in positive or negative direction of the given axes. Should be either <a href="#">SGrid::positive</a> or <a href="#">SGrid::negative</a> .
in	<i>k</i>	For k=1 it will return the direct neighbor. Higher values will give distant nodes in the given direction. If the indicated node is not within the grid any more, then invalid_node will be returned. For k=0 it will simply return ln.

##### Returns

size\_type The index of the neighbor node.

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

- src/sgrid.hh

## 4.33 hdnum::SparseMatrix< REAL > Class Template Reference

Sparse matrix Class with mathematical matrix operations.

```
#include <sparsematrix.hh>
```

### Classes

- class [builder](#)
- class [column\\_index\\_iterator](#)
- class [const\\_column\\_index\\_iterator](#)
- class [const\\_row\\_iterator](#)
- class [row\\_iterator](#)

### Public Types

- using **size\_type** = std::size\_t  
*Types used for array indices.*
- using **column\_iterator** = typename std::vector< REAL >::iterator  
*type of a regular column iterator (no access to indices)*
- using **const\_column\_iterator** = typename std::vector< REAL >::const\_iterator  
*type of a const regular column iterator (no access to indices)*



## Public Member Functions

- `SparseMatrix` ()=default  
*default constructor (empty `SparseMatrix`)*
- `SparseMatrix` (const `size_type` \_rows, const `size_type` \_cols)  
*constructor with added dimensions and columns*
- `size_type` rowsize () const  
*get number of rows of the matrix*
- `size_type` colsize () const  
*get number of columns of the matrix*
- bool **scientific** () const  
*pretty-print output properties*
- `row_iterator` begin ()  
*get a (possibly modifying) row iterator for the sparse matrix*
- `row_iterator` end ()  
*get a (possibly modifying) row iterator for the sparse matrix*
- `const_row_iterator` cbegin () const  
*get a (non modifying) row iterator for the sparse matrix*
- `const_row_iterator` cend () const  
*get a (non modifying) row iterator for the sparse matrix*
- `const_row_iterator` begin () const
- `const_row_iterator` end () const
- void `scientific` (bool b) const  
*Switch between floating point (default=true) and fixed point (false) display.*
- `size_type` **iwidth** () const  
*get index field width for pretty-printing*
- `size_type` **width** () const  
*get data field width for pretty-printing*
- `size_type` **precision** () const  
*get data precision for pretty-printing*
- void **iwidth** (`size_type` i) const  
*set index field width for pretty-printing*
- void **width** (`size_type` i) const  
*set data field width for pretty-printing*
- void **precision** (`size_type` i) const  
*set data precision for pretty-printing*
- `column_iterator` **find** (const `size_type` row\_index, const `size_type` col\_index) const
- bool **exists** (const `size_type` row\_index, const `size_type` col\_index) const
- REAL & **get** (const `size_type` row\_index, const `size_type` col\_index)  
*write access on matrix element  $A_{ij}$  using  $A.get(i,j)$*
- const REAL & **operator()** (const `size_type` row\_index, const `size_type` col\_index) const  
*read-access on matrix element  $A_{ij}$  using  $A(i,j)$*
- bool **operator==** (const `SparseMatrix` &other) const  
*checks whether two matrices are equal based on values and dimension*
- bool **operator!=** (const `SparseMatrix` &other) const  
*checks whether two matrices are unequal based on values and dimension*
- bool **operator<** (const `SparseMatrix` &other)=delete
- bool **operator>** (const `SparseMatrix` &other)=delete
- bool **operator<=** (const `SparseMatrix` &other)=delete
- bool **operator>=** (const `SparseMatrix` &other)=delete
- `SparseMatrix` **transpose** () const
- `SparseMatrix` **operator\*=** (const REAL scalar)

- `SparseMatrix operator/=` (const REAL scalar)
- `template<class V >`  
`void mv (Vector< V > &result, const Vector< V > &x) const`  
*matrix vector product  $y = A*x$*
- `Vector< REAL > operator*` (const Vector< REAL > &x) const  
*matrix vector product  $A*x$*
- `template<class V >`  
`void umv (Vector< V > &result, const Vector< V > &x) const`  
*update matrix vector product  $y += A*x$*
- `auto norm_infty` () const  
*calculate row sum norm*
- `std::string to_string` () const noexcept
- `void print` () const noexcept
- `SparseMatrix< REAL > matchingIdentity` () const  
*creates a matching identity*

## Static Public Member Functions

- static `SparseMatrix identity` (const `size_type` dimN)  
*identity for the matrix*

## Related Functions

(Note that these are not member functions.)

- `template<class REAL >`  
`void identity (SparseMatrix< REAL > &A)`

### 4.33.1 Detailed Description

```
template<typename REAL>
class hdnum::SparseMatrix< REAL >
```

Sparse matrix Class with mathematical matrix operations.

### 4.33.2 Constructor & Destructor Documentation

#### 4.33.2.1 SparseMatrix()

```
template<typename REAL >
hdnum::SparseMatrix< REAL >::SparseMatrix ( ) [default]
```

default constructor (empty `SparseMatrix`)

#### Example:

```
hdnum::SparseMatrix<double> A();
auto nRows = A.rowsize();
std::cout << "Matrix A has " << nRows << " rows." << std::endl;
```

#### Output:

```
Matrix A has 0 rows.
```

### 4.33.3 Member Function Documentation

#### 4.33.3.1 begin() [1/2]

```
template<typename REAL >
row_iterator hdnum::SparseMatrix< REAL >::begin ( ) [inline]
```

get a (possibly modifying) row iterator for the sparse matrix

The iterator points to the first row in the matrix.

**Example:**

```
// A is of type hdnum::SparseMatrix<int> and contains some values
// the deduced variable type for row_it is
// hdnum::SparseMatrix<int>::row_iterator
// but thats way to long to type out ;)
for(auto row_it = A.begin(); row_it != A.end(); row_it++) {
    for(auto val_it = row_it.begin(); val_it != row_it.end(); val_it++) {
        *val_it = 1;
    }
}
```

#### 4.33.3.2 begin() [2/2]

```
template<typename REAL >
const_row_iterator hdnum::SparseMatrix< REAL >::begin ( ) const [inline]
```

See also

[cbegin\(\) const](#)

#### 4.33.3.3 cbegin()

```
template<typename REAL >
const_row_iterator hdnum::SparseMatrix< REAL >::cbegin ( ) const [inline]
```

get a (non modifying) row iterator for the sparse matrix

The iterator points to the first row in the matrix.

#### 4.33.3.4 cend()

```
template<typename REAL >
const_row_iterator hdnum::SparseMatrix< REAL >::cend ( ) const [inline]
```

get a (non modifying) row iterator for the sparse matrix

The iterator points to the row one after the last one.

#### 4.33.3.5 colsize()

```
template<typename REAL >
size_type hdnum::SparseMatrix< REAL >::colsize ( ) const [inline]
```

get number of columns of the matrix

##### Example:

```
hdnum::SparseMatrix<double> A(4,5);
auto nRows = A.colsize();
std::cout << "Matrix A has " << nRows << " rows." << std::endl;
```

##### Output:

```
Matrix A has 4 rows.
```

#### 4.33.3.6 end() [1/2]

```
template<typename REAL >
row_iterator hdnum::SparseMatrix< REAL >::end ( ) [inline]
```

get a (possibly modifying) row iterator for the sparse matrix

The iterator points to the row one after the last one.

##### Example:

```
// A is of type hdnum::SparseMatrix<int> and contains some values
// the deduced variable type for row_it is
// hdnum::SparseMatrix<int>::row_iterator
// but thats way to long to type out ;)
for(auto row_it = A.begin(); row_it != A.end(); row_it++) {
    for(auto val_it = row_it.begin(); val_it != row_it.end(); val_it++) {
        *val_it = 1;
    }
}
```

#### 4.33.3.7 end() [2/2]

```
template<typename REAL >
const_row_iterator hdnum::SparseMatrix< REAL >::end ( ) const [inline]
```

See also

[cend\(\) const](#)

## 4.33.3.8 identity()

```
template<typename REAL >
static SparseMatrix<REAL> hdnum::SparseMatrix<REAL>::identity (
    const size_type dimN ) [inline], [static]
```

identity for the matrix

**Example:**

```
auto A = hdnum::SparseMatrix<double>::identity(4);
// fixed point representation for all SparseMatrix objects
A.scientific(false);
A.width(8);
A.precision(3);
std::cout << "A=" << A << std::endl;
```

**Output:**

```
A=
0      1      2      3
0  1.000  0.000  0.000  0.000
1  0.000  1.000  0.000  0.000
2  0.000  0.000  1.000  0.000
3  0.000  0.000  0.000  1.000
```

## 4.33.3.9 matchingIdentity()

```
template<typename REAL >
SparseMatrix<REAL> hdnum::SparseMatrix<REAL>::matchingIdentity ( ) const [inline]
```

creates a matching identity

**Example:**

```
auto A = hdnum::SparseMatrix<double>(4, 5);
auto B = A.matchingIdentity();
// fixed point representation for all SparseMatrix objects
A.scientific(false);
A.width(8);
A.precision(3);
std::cout << "A=" << A << std::endl;
```

**Output:**

```
A=
0      1      2      3
0  1.000  0.000  0.000  0.000
1  0.000  1.000  0.000  0.000
2  0.000  0.000  1.000  0.000
3  0.000  0.000  0.000  1.000
```

## 4.33.3.10 mv()

```
template<typename REAL >
template<class V >
void hdnum::SparseMatrix<REAL>::mv (
    Vector<V> & result,
    const Vector<V> & x ) const [inline]
```

matrix vector product  $y = A*x$

Implements  $y = A*x$  where  $x$  and  $y$  are a vectors and  $A$  is a matrix

## Parameters

in	<i>result</i>	reference to the resulting <a href="#">Vector</a>
in	<i>x</i>	constant reference to a <a href="#">Vector</a>

**4.33.3.11 norm\_infty()**

```
template<typename REAL >
auto hdnum::SparseMatrix< REAL >::norm_infty ( ) const [inline]
```

calculate row sum norm

$$||A||_{\infty} = \max_{i=1 \dots m} \sum_{j=1}^n |a_{ij}|$$

**4.33.3.12 operator\*()**

```
template<typename REAL >
Vector< REAL > hdnum::SparseMatrix< REAL >::operator* (
    const Vector< REAL > & x ) const [inline]
```

matrix vector product A\*x

Implements A\*x where x is a vectors and A is a matrix

## Parameters

in	<i>x</i>	constant reference to a <a href="#">Vector</a>
----	----------	--

**4.33.3.13 operator\*=( )**

```
template<typename REAL >
SparseMatrix hdnum::SparseMatrix< REAL >::operator*= (
    const REAL scalar ) [inline]
```

Element-wise multiplication of the matrix

## Parameters

in	<i>scalar</i>	with same type as the matrix elements
----	---------------	---------------------------------------

#### 4.33.3.14 operator/=( )

```
template<typename REAL >
SparseMatrix< REAL >::operator/= (
    const REAL scalar ) [inline]
```

Element-wise division of the matrix

##### Parameters

in	<i>scalar</i>	with same type as the matrix elements
----	---------------	---------------------------------------

#### 4.33.3.15 rowsize()

```
template<typename REAL >
size_type hdnum::SparseMatrix< REAL >::rowsize ( ) const [inline]
```

get number of rows of the matrix

##### Example:

```
hdnum::SparseMatrix<double> A(4,5);
auto nRows = A.rowsize();
std::cout << "Matrix A has " << nRows << " rows." << std::endl;
```

##### Output:

Matrix A has 4 rows.

#### 4.33.3.16 scientific()

```
template<typename REAL >
void hdnum::SparseMatrix< REAL >::scientific (
    bool b ) const [inline]
```

Switch between floating point (default=true) and fixed point (false) display.

##### Example:

```
hdnum::SparseMatrix<double> A(4,4);
// fixed point representation for all SparseMatrix objects
A.scientific(false);
A.width(8); A.precision(3); identity(A);
// Defines the identity matrix of the same dimension
std::cout << "A=" << A << std::endl;
```

##### Output:

```
A=
0      1      2      3
0  1.000  0.000  0.000  0.000
1  0.000  1.000  0.000  0.000
2  0.000  0.000  1.000  0.000
3  0.000  0.000  0.000  1.000
```

#### 4.33.3.17 umv()

```
template<typename REAL >
template<class V >
void hdnum::SparseMatrix< REAL >::umv (
    Vector< V > & result,
    const Vector< V > & x ) const [inline]
```

update matrix vector product  $y \leftarrow y + A \cdot x$

Implements  $y \leftarrow y + A \cdot x$  where  $x$  and  $y$  are vectors and  $A$  is a matrix

##### Parameters

in	<i>result</i>	reference to the resulting <a href="#">Vector</a>
in	<i>x</i>	constant reference to a <a href="#">Vector</a>

### 4.33.4 Friends And Related Function Documentation

#### 4.33.4.1 identity()

```
template<class REAL >
void identity (
    SparseMatrix< REAL > & A ) [related]
```

**Function:** make identity matrix

```
template<class T>
inline void identity (SparseMatrix<T> &A)
```

##### Parameters

in	<i>A</i>	reference to a <a href="#">SparseMatrix</a> that shall be filled with entries
----	----------	---

##### Example:

```
hdnum::SparseMatrix<double> A(4,4);
identity(A);
// fixed point representation for all DenseMatrix objects
A.scientific(false);
A.width(10);
A.precision(5);
std::cout << "A=" << A << std::endl;
```

##### Output:

```
A=
0      1      2      3
0  1.00000  0.00000  0.00000  0.00000
1  0.00000  1.00000  0.00000  0.00000
2  0.00000  0.00000  1.00000  0.00000
3  0.00000  0.00000  0.00000  1.00000
```

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



- `src/densematrix.hh`
- `src/sparsematrix.hh`

## 4.34 `hdnum::SquareRootProblem< N >` Class Template Reference

Example class for a nonlinear model  $F(x) = 0$ ;

```
#include <newton.hh>
```

### Public Types

- `typedef std::size_t size_type`  
*export size\_type*
- `typedef N number_type`  
*export number\_type*

### Public Member Functions

- `SquareRootProblem (number_type a_)`  
*constructor stores parameter lambda*
- `std::size_t size () const`  
*return number of componentes for the model*
- `void F (const Vector< N > &x, Vector< N > &result) const`  
*model evaluation*
- `void F_x (const Vector< N > &x, DenseMatrix< N > &result) const`  
*jacobian evaluation needed for implicit solvers*

#### 4.34.1 Detailed Description

```
template<class N>
class hdnum::SquareRootProblem< N >
```

Example class for a nonlinear model  $F(x) = 0$ ;

This example solves  $F(x) = x*x - a = 0$

#### Template Parameters

<code>N</code>	a type representing x and F components
----------------	--

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

- `src/newton.hh`

## 4.35 hdnum::StationarySolver< M > Class Template Reference

Stationary problem solver. E.g. for elliptic problems.

```
#include <pde.hh>
```

### Public Types

- typedef M::size\_type **size\_type**  
*export size\_type*
- typedef M::time\_type **time\_type**  
*export time\_type*
- typedef M::number\_type **number\_type**  
*export number\_type*

### Public Member Functions

- **StationarySolver** (const M &model\_)  
*constructor stores reference to the model*
- void **solve** ()  
*do one step*
- const [Vector](#)< [number\\_type](#) > & **get\_state** () const  
*get current state*
- [size\\_type](#) **get\_order** () const  
*return consistency order of the method*

#### 4.35.1 Detailed Description

```
template<class M>
class hdnum::StationarySolver< M >
```

Stationary problem solver. E.g. for elliptic problems.

The PDE solver is parametrized by a model. The model also exports all relevant types for the solution. The PDE solver encapsulates the states needed for the computation.

#### Template Parameters

<i>M</i>	the model type
----------	----------------

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

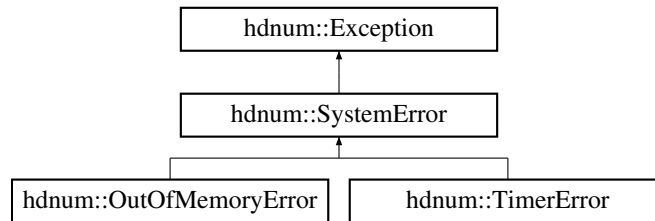
- [src/pde.hh](#)

## 4.36 hdnum::SystemError Class Reference

Default exception class for OS errors.

```
#include <exceptions.hh>
```

Inheritance diagram for hdnum::SystemError:



## Additional Inherited Members

### 4.36.1 Detailed Description

Default exception class for OS errors.

This class is thrown when a system-call is used and returns an error.

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

- [src/exceptions.hh](#)

## 4.37 hdnum::Timer Class Reference

A simple stop watch.

```
#include <timer.hh>
```

### Public Member Functions

- **Timer** ()  
*A new timer, start immediately.*
- void **reset** ()  
*Reset timer.*
- double **elapsed** () const  
*Get elapsed user-time in seconds.*

### 4.37.1 Detailed Description

A simple stop watch.

This class reports the elapsed user-time, i.e. time spent computing, after the last call to [Timer::reset\(\)](#). The results are seconds and fractional seconds. Note that the resolution of the timing depends on your OS kernel which should be somewhere in the millisecond range.

The class is basically a wrapper for the libc-function `getrusage()`

Taken from the DUNE project [www.dune-project.org](http://www.dune-project.org)

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

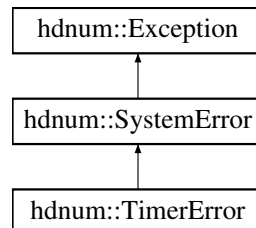
- [src/timer.hh](#)

## 4.38 hdnum::TimerError Class Reference

Exception thrown by the [Timer](#) class

```
#include <timer.hh>
```

Inheritance diagram for hdnum::TimerError:



### Additional Inherited Members

#### 4.38.1 Detailed Description

Exception thrown by the [Timer](#) class

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

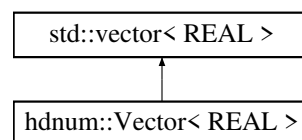
- [src/timer.hh](#)

## 4.39 hdnum::Vector< REAL > Class Template Reference

Class with mathematical vector operations.

```
#include <vector.hh>
```

Inheritance diagram for hdnum::Vector< REAL >:



### Public Types

- `typedef std::size_t size_type`  
*Type used for array indices.*

## Public Member Functions

- **Vector** ()  
*default constructor, also inherited from the STL vector default constructor*
- **Vector** (const `size_t` size, const `REAL` defaultvalue\_=0)  
*another constructor, with arguments, setting the default value for all entries of the vector of given size*
- **Vector** (const `std::initializer_list< REAL >` &v)  
*constructor from initializer list*
- **Vector** & **operator=** (const `REAL` value)  
*Assign all values of the **Vector** from one scalar value:  $x = \text{value}$ .*
- **Vector** sub (`size_type` i, `size_type` m)  
*Subvector extraction.*
- **Vector** & **operator\*=** (const `REAL` value)  
*Multiplication by a scalar value ( $x *= \text{value}$ )*
- **Vector** & **operator/=** (const `REAL` value)  
*Division by a scalar value ( $x /= \text{value}$ )*
- **Vector** & **operator+=** (const **Vector** &y)  
*Add another vector ( $x += y$ )*
- **Vector** & **operator-=** (const **Vector** &y)  
*Subtract another vector ( $x -= y$ )*
- **Vector** & **update** (const `REAL` alpha, const **Vector** &y)  
*Update vector by addition of a scaled vector ( $x += a y$ )*
- `REAL` **operator\*** (**Vector** &x) const  
*Inner product with another vector.*
- **Vector** **operator+** (**Vector** &x) const  
*Adding two vectors  $x+y$ .*
- **Vector** **operator-** (**Vector** &x) const  
*vector subtraction  $x-y$*
- `REAL` **two\_norm\_2** () const  
*Square of the Euclidean norm.*
- `REAL` **two\_norm** () const  
*Euclidean norm of a vector.*
- `bool` **scientific** () const  
*pretty-print output property: true = scientific, false = fixed point representation*
- `void` **scientific** (`bool` b) const  
*scientific(true) is the default, scientific(false) switches to the fixed point representation*
- `std::size_t` **iwidth** () const  
*get index field width for pretty-printing*
- `std::size_t` **width** () const  
*get data field width for pretty-printing*
- `std::size_t` **precision** () const  
*get data precision for pretty-printing*
- `void` **iwidth** (`std::size_t` i) const  
*set index field width for pretty-printing*
- `void` **width** (`std::size_t` i) const  
*set data field width for pretty-printing*
- `void` **precision** (`std::size_t` i) const  
*set data precision for pretty-printing*

## Related Functions

(Note that these are not member functions.)

- `template<typename REAL >`  
`std::ostream & operator<< (std::ostream &os, const Vector< REAL > &x)`  
*Output operator for Vector.*
- `template<typename REAL >`  
`void gnuplot (const std::string &fname, const Vector< REAL > x)`  
*Output contents of a Vector x to a text file named fname.*
- `template<typename REAL >`  
`void readVectorFromFile (const std::string &filename, Vector< REAL > &vector)`  
*Read vector from a text file.*
- `template<class REAL >`  
`void fill (Vector< REAL > &x, const REAL &t, const REAL &dt)`  
*Fill vector, with entries starting at t, consecutively shifted by dt.*
- `template<class REAL >`  
`void unitvector (Vector< REAL > &x, std::size_t j)`  
*Defines j-th unitvector (j=0,...,n-1) where n = length of the vector.*

### 4.39.1 Detailed Description

```
template<typename REAL>
class hdnum::Vector< REAL >
```

Class with mathematical vector operations.

### 4.39.2 Member Function Documentation

#### 4.39.2.1 operator\*()

```
template<typename REAL >
REAL hdnum::Vector< REAL >::operator* (
    Vector< REAL > & x ) const [inline]
```

Inner product with another vector.

#### Example:

```
hdnum::Vector<double> x(2);
x.scientific(false); // set fixed point display mode
x[0] = 12.0;
x[1] = 3.0;
std::cout << "x=" << x << std::endl;
hdnum::Vector<double> y(2);
y[0] = 4.0;
y[1] = -1.0;
std::cout << "y=" << y << std::endl;
double s = x*y;
std::cout << "s = x*y = " << s << std::endl;
```

#### Output:

```

x=
[ 0]      12.00000000
[ 1]       3.00000000

y=
[ 0]       4.00000000
[ 1]      -1.00000000

s = x*y = 45.00000000

```

#### 4.39.2.2 operator+()

```

template<typename REAL >
Vector hdnum::Vector< REAL >::operator+ (
    Vector< REAL > & x ) const [inline]

```

Adding two vectors x+y.

##### Example:

```

hdnum::Vector<double> x(2);
x.scientific(false); // set fixed point display mode
x[0] = 12.0;
x[1] = 3.0;
std::cout << "x=" << x << std::endl;
hdnum::Vector<double> y(2);
y[0] = 4.0;
y[1] = -1.0;
std::cout << "y=" << y << std::endl;
std::cout << "x+y = " << x+y << std::endl;

```

##### Output:

```

x=
[ 0]      12.00000000
[ 1]       3.00000000

y=
[ 0]       4.00000000
[ 1]      -1.00000000

x+y =
[ 0]      16.00000000
[ 1]       2.00000000

```

#### 4.39.2.3 operator-()

```

template<typename REAL >
Vector hdnum::Vector< REAL >::operator- (
    Vector< REAL > & x ) const [inline]

```

vector subtraction x-y

##### Example:

```

hdnum::Vector<double> x(2);
x.scientific(false); // set fixed point display mode
x[0] = 12.0;
x[1] = 3.0;
std::cout << "x=" << x << std::endl;
hdnum::Vector<double> y(2);
y[0] = 4.0;
y[1] = -1.0;
std::cout << "y=" << y << std::endl;
std::cout << "x-y = " << x-y << std::endl;

```

##### Output:

```

x=
[ 0]      12.00000000
[ 1]       3.00000000

y=
[ 0]       4.00000000
[ 1]      -1.00000000

x-y =
[ 0]       8.00000000
[ 1]       4.00000000

```

#### 4.39.2.4 operator=()

```

template<typename REAL >
Vector & hdnum::Vector< REAL >::operator= (
    const REAL value ) [inline]

```

Assign all values of the [Vector](#) from one scalar value: `x = value`.

##### Parameters

in	value	constant value which should be assigned
----	-------	---

##### Example:

```

hdnum::Vector<double> x(4);
x = 1.23;
std::cout << "x=" << x << std::endl;

```

##### Output:

```

x=
[ 0]  1.2340000e+00
[ 1]  1.2340000e+00
[ 2]  1.2340000e+00
[ 3]  1.2340000e+00

```

#### 4.39.2.5 scientific()

```

template<typename REAL >
void hdnum::Vector< REAL >::scientific (
    bool b ) const [inline]

```

`scientific(true)` is the default, `scientific(false)` switches to the fixed point representation

##### Example:

```

hdnum::Vector<double> x(3);
x[0] = 2.0;
x[1] = 2.0;
x[2] = 1.0;
std::cout << "x=" << x << std::endl;
x.scientific(false); // set fixed point display mode
std::cout << "x=" << x << std::endl;

```

##### Output:



```
x=
[ 0]  2.0000000e+00
[ 1]  2.0000000e+00
[ 2]  1.0000000e+00
```

```
x=
[ 0]      2.0000000
[ 1]      2.0000000
[ 2]      1.0000000
```

#### 4.39.2.6 sub()

```
template<typename REAL >
Vector hdnum::Vector< REAL >::sub (
    size_type i,
    size_type m ) [inline]
```

Subvector extraction.

Returns a new vector that is a subset of the components of the given vector.

##### Parameters

in	<i>i</i>	first index of the new vector
in	<i>m</i>	size of the new vector, i.e. it has components [i,i+m-1]

#### 4.39.2.7 two\_norm()

```
template<typename REAL >
REAL hdnum::Vector< REAL >::two_norm ( ) const [inline]
```

Euclidean norm of a vector.

##### Example:

```
hdnum::Vector<double> x(3);
x.scientific(false); // set fixed point display mode
x[0] = 2.0;
x[1] = 2.0;
x[2] = 1.0;
std::cout << "x=" << x << std::endl;
std::cout << "euclidean norm of x = " << x.two_norm() << std::endl;
```

##### Output:

```
x=
[ 0]      2.0000000
[ 1]      2.0000000
[ 2]      1.0000000
```

```
euclidean norm of x = 3.0000000
```

### 4.39.3 Friends And Related Function Documentation

#### 4.39.3.1 fill()

```
template<class REAL >
void fill (
    Vector< REAL > & x,
    const REAL & t,
    const REAL & dt ) [related]
```

Fill vector, with entries starting at t, consecutively shifted by dt.

##### Example:

```
hdnum::Vector<double> x(5);
fill(x,2.01,0.1);
x.scientific(false); // set fixed point display mode
std::cout << "x=" << x << std::endl;
```

##### Output:

```
x=
[ 0]      2.0100000
[ 1]      2.1100000
[ 2]      2.2100000
[ 3]      2.3100000
[ 4]      2.4100000
```

#### 4.39.3.2 gnuplot()

```
template<typename REAL >
void gnuplot (
    const std::string & fname,
    const Vector< REAL > x ) [related]
```

Output contents of a [Vector](#) x to a text file named fname.

##### Example:

```
hdnum::Vector<double> x(5);
unitvector(x,3);
x.scientific(false); // set fixed point display mode
gnuplot("test.dat",x);
```

##### Output:

```
Contents of 'test.dat':
0      0.0000000
1      0.0000000
2      0.0000000
3      1.0000000
4      0.0000000
```

### 4.39.3.3 operator<<()

```
template<typename REAL >
std::ostream & operator<< (
    std::ostream & os,
    const Vector< REAL > & x ) [related]
```

Output operator for [Vector](#).

#### Example:

```
hdnum::Vector<double> x(3);
x[0] = 2.0;
x[1] = 2.0;
x[2] = 1.0;
std::cout << "x=" << x << std::endl;
```

#### Output:

```
x=
[ 0]  2.0000000e+00
[ 1]  2.0000000e+00
[ 2]  1.0000000e+00
```

### 4.39.3.4 readVectorFromFile()

```
template<typename REAL >
void readVectorFromFile (
    const std::string & filename,
    Vector< REAL > & vector ) [related]
```

Read vector from a text file.

#### Parameters

in	<i>filename</i>	name of the text file
in, out	<i>vector</i>	reference to a <a href="#">Vector</a>

#### Example:

```
hdnum::Vector<number> x;
readVectorFromFile("x.dat", x );
std::cout << "x=" << x << std::endl;
```

#### Output:

```
Contents of "x.dat":
1.0
2.0
3.0
```

would give:

```
x=
[ 0]  1.0000000e+00
[ 1]  2.0000000e+00
[ 2]  3.0000000e+00
```

#### 4.39.3.5 unitvector()

```
template<class REAL >
void unitvector (
    Vector< REAL > & x,
    std::size_t j ) [related]
```

Defines j-th unitvector (j=0,...,n-1) where n = length of the vector.

##### Example:

```
hdnum::Vector<double> x(5);
unitvector(x,3);
x.scientific(false); // set fixed point display mode
std::cout << "x=" << x << std::endl;
```

##### Output:

```
x=
[ 0]      0.0000000
[ 1]      0.0000000
[ 2]      0.0000000
[ 3]      1.0000000
[ 4]      0.0000000
```

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

- src/vector.hh

## Chapter 5

# File Documentation

### 5.1 densematrix.hh

```
1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 /*
3  * File:    densematrix.hh
4  * Author:  ngo
5  *
6  * Created on April 15, 2011
7  */
8
9 #ifndef DENSEMATRIX_HH
10 #define DENSEMATRIX_HH
11
12 #include <cstdlib>
13 #include <fstream>
14 #include <iomanip>
15 #include <iostream>
16 #include <sstream>
17 #include <string>
18
19 #include "exceptions.hh"
20 #include "sparsematrix.hh"
21 #include "vector.hh"
22
23 namespace hdnum {
24
25 // forward-declare the sparse matrix template to make the transforming
26 // constructor from hdnum::SparseMatrix -> hdnum::DenseMatrix working
27 template <typename REAL>
28 class SparseMatrix;
29
30 template <typename REAL>
31 class DenseMatrix {
32 public:
33     typedef std::size_t size_type;
34     typedef typename std::vector<REAL> VType;
35     typedef typename VType::const_iterator ConstVectorIterator;
36     typedef typename VType::iterator VectorIterator;
37
38 private:
39     VType m_data; // Matrix data is stored in an STL vector!
40     std::size_t m_rows; // Number of Matrix rows
41     std::size_t m_cols; // Number of Matrix columns
42
43     static bool bScientific;
44     static std::size_t nIndexWidth;
45     static std::size_t nValueWidth;
46     static std::size_t nValuePrecision;
47
48     REAL myabs(REAL x) const {
49         if (x >= REAL(0))
50             return x;
51         else
52             return -x;
53     }
54
55     inline REAL& at(const std::size_t row, const std::size_t col) {
56         return m_data[row * m_cols + col];
57     }
58 }
59
60
61
62
63
```

```

65     inline const REAL& at(const std::size_t row, const std::size_t col) const {
66         return m_data[row * m_cols + col];
67     }
68
69 public:
70     DenseMatrix() : m_data(0, 0), m_rows(0), m_cols(0) {}
71
72     DenseMatrix(const std::size_t _rows, const std::size_t _cols,
73                 const REAL def_val = 0)
74         : m_data(_rows * _cols, def_val), m_rows(_rows), m_cols(_cols) {}
75
76     DenseMatrix(const std::initializer_list<std::initializer_list<REAL>>& v) {
77         m_rows = v.size();
78         m_cols = v.begin()->size();
79         for (auto row : v) {
80             if (row.size() != m_cols) {
81                 std::cout << "Zeilen der Matrix nicht gleich lang" << std::endl;
82                 exit(1);
83             }
84             for (auto elem : row) m_data.push_back(elem);
85         }
86     }
87
88     DenseMatrix(const hdnum::SparseMatrix<REAL>& other)
89         : m_data(other.rowsize() * other.colsize(), m_rows(other.rowsize()),
90                 m_cols(other.colsize())) {
91         using counter_type = typename hdnum::SparseMatrix<REAL>::size_type;
92         counter_type row_index {};
93         for (auto& row : other) {
94             for (auto it = row.ibegin(); it != row.iend(); it++) {
95                 this->operator[](row_index)[it.index()] = it.value();
96             }
97             row_index++;
98         }
99     }
100
101     void addNewRow(const hdnum::Vector<REAL>& rowvector) {
102         m_rows++;
103         m_cols = rowvector.size();
104         for (std::size_t i = 0; i < m_cols; i++) m_data.push_back(rowvector[i]);
105     }
106
107     /*
108     // copy constructor (not needed, since it inherits from the STL vector)
109     DenseMatrix( const DenseMatrix& A )
110     {
111         this->m_data = A.m_data;
112         m_rows = A.m_rows;
113         m_cols = A.m_cols;
114     }
115     */
116
117     size_t rowsize() const { return m_rows; }
118
119     size_t colsize() const { return m_cols; }
120
121     // pretty-print output properties
122     bool scientific() const { return bScientific; }
123
124     void scientific(bool b) const { bScientific = b; }
125
126     std::size_t iwidth() const { return nIndexWidth; }
127
128     std::size_t width() const { return nValueWidth; }
129
130     std::size_t precision() const { return nValuePrecision; }
131
132     void iwidth(std::size_t i) const { nIndexWidth = i; }
133
134     void width(std::size_t i) const { nValueWidth = i; }
135
136     void precision(std::size_t i) const { nValuePrecision = i; }
137
138     // overloaded element access operators
139     // write access on matrix element A_ij using A(i,j)
140     inline REAL& operator()(const std::size_t row, const std::size_t col) {
141         assert(row < m_rows || col < m_cols);
142         return at(row, col);
143     }
144
145     inline const REAL& operator()(const std::size_t row,
146                                   const std::size_t col) const {
147         assert(row < m_rows || col < m_cols);
148         return at(row, col);
149     }
150
151     const ConstVectorIterator operator[](const std::size_t row) const {

```

```

258     assert(row < m_rows);
259     return m_data.begin() + row * m_cols;
260 }
261
263 VectorIterator operator[](const std::size_t row) {
264     assert(row < m_rows);
265     return m_data.begin() + row * m_cols;
266 }
267
290 DenseMatrix& operator=(const DenseMatrix& A) {
291     m_data = A.m_data;
292     m_rows = A.m_rows;
293     m_cols = A.m_cols;
294     return *this;
295 }
296
316 DenseMatrix& operator=(const REAL value) {
317     for (std::size_t i = 0; i < rowsize(); i++)
318         for (std::size_t j = 0; j < colsize(); j++) (*this)(i, j) = value;
319     return *this;
320 }
321
334 DenseMatrix sub(size_type i, size_type j, size_type rows, size_type cols) {
335     DenseMatrix A(rows, cols);
336     DenseMatrix& self = *this;
337     for (size_type k1 = 0; k1 < rows; k1++) {
338         for (size_type k2 = 0; k2 < cols; k2++) {
339             A[k1][k2] = self[k1 + i][k2 + j];
340         }
341     }
342     return A;
343 }
344
350 DenseMatrix transpose() const {
351     DenseMatrix A(m_cols, m_rows);
352     for (size_type i = 0; i < m_rows; i++) {
353         for (size_type j = 0; j < m_cols; j++) {
354             A[j][i] = this->operator[](i)[j];
355         }
356     }
357     return A;
358 }
359
360 // Basic Matrix Operations
361
369 DenseMatrix& operator+=(const DenseMatrix& B) {
370     for (size_type i = 0; i < rowsize(); ++i) {
371         for (size_type j = 0; j < colsize(); ++j) {
372             (*this)(i, j) += B(i, j);
373         }
374     }
375     return *this;
376 }
377
385 DenseMatrix& operator-=(const DenseMatrix& B) {
386     for (std::size_t i = 0; i < rowsize(); ++i)
387         for (std::size_t j = 0; j < colsize(); ++j)
388             (*this)(i, j) -= B(i, j);
389     return *this;
390 }
391
421 DenseMatrix& operator*=(const REAL s) {
422     for (std::size_t i = 0; i < rowsize(); ++i)
423         for (std::size_t j = 0; j < colsize(); ++j) (*this)(i, j) *= s;
424     return *this;
425 }
426
457 DenseMatrix& operator/=(const REAL s) {
458     for (std::size_t i = 0; i < rowsize(); ++i)
459         for (std::size_t j = 0; j < colsize(); ++j) (*this)(i, j) /= s;
460     return *this;
461 }
462
489 void update(const REAL s, const DenseMatrix& B) {
490     for (std::size_t i = 0; i < rowsize(); ++i)
491         for (std::size_t j = 0; j < colsize(); ++j)
492             (*this)(i, j) += s * B(i, j);
493 }
494
537 template <class V>
538 void mv(Vector<V>& y, const Vector<V>& x) const {
539     if (this->rowsize() != y.size())
540         HDNUM_ERROR("mv: size of A and y do not match");
541     if (this->colsize() != x.size())
542         HDNUM_ERROR("mv: size of A and x do not match");
543     for (std::size_t i = 0; i < rowsize(); ++i) {
544         y[i] = 0;

```

```

545         for (std::size_t j = 0; j < colsize(); ++j)
546             y[i] += (*this)(i, j) * x[j];
547     }
548 }
549
550 template <class V>
551 void umv(Vector<V>& y, const Vector<V>& x) const {
552     if (this->rowsize() != y.size())
553         HDNUM_ERROR("mv: size of A and y do not match");
554     if (this->colsize() != x.size())
555         HDNUM_ERROR("mv: size of A and x do not match");
556     for (std::size_t i = 0; i < rowsize(); ++i) {
557         for (std::size_t j = 0; j < colsize(); ++j)
558             y[i] += (*this)(i, j) * x[j];
559     }
560 }
561
562 template <class V>
563 void umv(Vector<V>& y, const V& s, const Vector<V>& x) const {
564     if (this->rowsize() != y.size())
565         HDNUM_ERROR("mv: size of A and y do not match");
566     if (this->colsize() != x.size())
567         HDNUM_ERROR("mv: size of A and x do not match");
568     for (std::size_t i = 0; i < rowsize(); ++i) {
569         for (std::size_t j = 0; j < colsize(); ++j)
570             y[i] += s * (*this)(i, j) * x[j];
571     }
572 }
573
574 void mm(const DenseMatrix<REAL>& A, const DenseMatrix<REAL>& B) {
575     if (this->rowsize() != A.rowsize())
576         HDNUM_ERROR("mm: size incompatible");
577     if (this->colsize() != B.colsize())
578         HDNUM_ERROR("mm: size incompatible");
579     if (A.colsize() != B.rowsize()) HDNUM_ERROR("mm: size incompatible");
580
581     for (std::size_t i = 0; i < rowsize(); i++)
582         for (std::size_t j = 0; j < colsize(); j++) {
583             (*this)(i, j) = 0;
584             for (std::size_t k = 0; k < A.colsize(); k++)
585                 (*this)(i, j) += A(i, k) * B(k, j);
586         }
587 }
588
589 void umm(const DenseMatrix<REAL>& A, const DenseMatrix<REAL>& B) {
590     if (this->rowsize() != A.rowsize())
591         HDNUM_ERROR("mm: size incompatible");
592     if (this->colsize() != B.colsize())
593         HDNUM_ERROR("mm: size incompatible");
594     if (A.colsize() != B.rowsize()) HDNUM_ERROR("mm: size incompatible");
595
596     for (std::size_t i = 0; i < rowsize(); i++)
597         for (std::size_t j = 0; j < colsize(); j++)
598             for (std::size_t k = 0; k < A.colsize(); k++)
599                 (*this)(i, j) += A(i, k) * B(k, j);
600 }
601
602 void sc(const Vector<REAL>& x, std::size_t k) {
603     if (this->rowsize() != x.size()) HDNUM_ERROR("cc: size incompatible");
604
605     for (std::size_t i = 0; i < rowsize(); i++) (*this)(i, k) = x[i];
606 }
607
608 void sr(const Vector<REAL>& x, std::size_t k) {
609     if (this->colsize() != x.size()) HDNUM_ERROR("cc: size incompatible");
610
611     for (std::size_t i = 0; i < colsize(); i++) (*this)(k, i) = x[i];
612 }
613
614 REAL norm_infty() const {
615     REAL norm(0.0);
616     for (std::size_t i = 0; i < rowsize(); i++) {
617         REAL sum(0.0);
618         for (std::size_t j = 0; j < colsize(); j++)
619             sum += myabs((*this)(i, j));
620         if (sum > norm) norm = sum;
621     }
622     return norm;
623 }
624
625 REAL norm_1() const {
626     REAL norm(0.0);
627     for (std::size_t j = 0; j < colsize(); j++) {
628         REAL sum(0.0);
629         for (std::size_t i = 0; i < rowsize(); i++)
630             sum += myabs((*this)(i, j));
631         if (sum > norm) norm = sum;
632     }
633 }

```



```

900     }
901     return norm;
902 }
903
949 Vector<REAL> operator*(const Vector<REAL>& x) const {
950     assert(x.size() == colsize());
951     Vector<REAL> y(rowsize());
952     for (std::size_t r = 0; r < rowsize(); ++r) {
953         for (std::size_t c = 0; c < colsize(); ++c) {
954             y[r] += at(r, c) * x[c];
955         }
956     }
957     return y;
958 }
959
1003 DenseMatrix operator*(const DenseMatrix& x) const {
1004     assert(colsize() == x.rowsize());
1005
1006     const std::size_t out_rows = rowsize();
1007     const std::size_t out_cols = x.colsize();
1008     DenseMatrix y(out_rows, out_cols, 0.0);
1009     for (std::size_t r = 0; r < out_rows; ++r)
1010         for (std::size_t c = 0; c < out_cols; ++c)
1011             for (std::size_t i = 0; i < colsize(); ++i)
1012                 y(r, c) += at(r, i) * x(i, c);
1013
1014     return y;
1015 }
1016
1059 DenseMatrix operator+(const DenseMatrix& x) const {
1060     assert(colsize() == x.colsize());
1061     assert(rowsize() == x.rowsize());
1062
1063     const std::size_t out_rows = rowsize();
1064     const std::size_t out_cols = x.colsize();
1065     DenseMatrix y(out_rows, out_cols, 0.0);
1066     y = *this;
1067     y += x;
1068     return y;
1069 }
1070
1113 DenseMatrix operator-(const DenseMatrix& x) const {
1114     assert(colsize() == x.colsize());
1115     assert(rowsize() == x.rowsize());
1116
1117     const std::size_t out_rows = rowsize();
1118     const std::size_t out_cols = x.colsize();
1119     DenseMatrix y(out_rows, out_cols, 0.0);
1120     y = *this;
1121     y -= x;
1122     return y;
1123 }
1124 };
1125
1126 template <typename REAL>
1127 bool DenseMatrix<REAL>::bScientific = true;
1128 template <typename REAL>
1129 std::size_t DenseMatrix<REAL>::nIndexWidth = 10;
1130 template <typename REAL>
1131 std::size_t DenseMatrix<REAL>::nValueWidth = 10;
1132 template <typename REAL>
1133 std::size_t DenseMatrix<REAL>::nValuePrecision = 3;
1134
1158 template <typename REAL>
1159 inline std::ostream& operator<<(std::ostream& s, const DenseMatrix<REAL>& A) {
1160     s << std::endl;
1161     s << " " << std::setw(A.iwidth()) << " "
1162       << " ";
1163     for (typename DenseMatrix<REAL>::size_type j = 0; j < A.colsize(); ++j)
1164         s << std::setw(A.width()) << j << " ";
1165     s << std::endl;
1166
1167     for (typename DenseMatrix<REAL>::size_type i = 0; i < A.rowsize(); ++i) {
1168         s << " " << std::setw(A.iwidth()) << i << " ";
1169         for (typename DenseMatrix<REAL>::size_type j = 0; j < A.colsize();
1170              ++j) {
1171             if (A.scientific()) {
1172                 s << std::setw(A.width()) << std::scientific << std::showpoint
1173                   << std::setprecision(A.precision()) << A[i][j] << " ";
1174             } else {
1175                 s << std::setw(A.width()) << std::fixed << std::showpoint
1176                   << std::setprecision(A.precision()) << A[i][j] << " ";
1177             }
1178         }
1179         s << std::endl;
1180     }

```

```

1181     return s;
1182 }
1183
1190 template <typename REAL>
1191 inline void fill(DenseMatrix<REAL>& A, const REAL& t) {
1192     for (typename DenseMatrix<REAL>::size_type i = 0; i < A.rowsize(); ++i)
1193         for (typename DenseMatrix<REAL>::size_type j = 0; j < A.colsize(); ++j)
1194             A[i][j] = t;
1195 }
1196
1198 template <typename REAL>
1199 inline void zero(DenseMatrix<REAL>& A) {
1200     for (std::size_t i = 0; i < A.rowsize(); ++i)
1201         for (std::size_t j = 0; j < A.colsize(); ++j) A(i, j) = REAL(0);
1202 }
1203
1237 template <class T>
1238 inline void identity(DenseMatrix<T>& A) {
1239     for (typename DenseMatrix<T>::size_type i = 0; i < A.rowsize(); ++i)
1240         for (typename DenseMatrix<T>::size_type j = 0; j < A.colsize(); ++j)
1241             if (i == j)
1242                 A[i][i] = T(1);
1243             else
1244                 A[i][j] = T(0);
1245 }
1246
1282 template <typename REAL>
1283 inline void spd(DenseMatrix<REAL>& A) {
1284     if (A.rowsize() != A.colsize() || A.rowsize() == 0)
1285         HDNUM_ERROR("need square and nonempty matrix");
1286     for (std::size_t i = 0; i < A.rowsize(); ++i)
1287         for (std::size_t j = 0; j < A.colsize(); ++j)
1288             if (i == j)
1289                 A(i, i) = REAL(4.0);
1290             else
1291                 A(i, j) = -REAL(1.0) / ((i - j) * (i - j));
1292 }
1293
1343 template <typename REAL>
1344 inline void vandermonde(DenseMatrix<REAL>& A, const Vector<REAL> x) {
1345     if (A.rowsize() != A.colsize() || A.rowsize() == 0)
1346         HDNUM_ERROR("need square and nonempty matrix");
1347     if (A.rowsize() != x.size()) HDNUM_ERROR("need A and x of same size");
1348     for (typename DenseMatrix<REAL>::size_type i = 0; i < A.rowsize(); ++i) {
1349         REAL p(1.0);
1350         for (typename DenseMatrix<REAL>::size_type j = 0; j < A.colsize();
1351              ++j) {
1352             A[i][j] = p;
1353             p *= x[i];
1354         }
1355     }
1356 }
1357
1359 template <typename REAL>
1360 inline void gnuplot(const std::string& fname, const DenseMatrix<REAL>& A) {
1361     std::fstream f(fname.c_str(), std::ios::out);
1362     for (typename DenseMatrix<REAL>::size_type i = 0; i < A.rowsize(); ++i) {
1363         for (typename DenseMatrix<REAL>::size_type j = 0; j < A.colsize();
1364              ++j) {
1365             if (A.scientific()) {
1366                 f << std::setw(A.width()) << std::scientific << std::showpoint
1367                   << std::setprecision(A.precision()) << A[i][j];
1368             } else {
1369                 f << std::setw(A.width()) << std::fixed << std::showpoint
1370                   << std::setprecision(A.precision()) << A[i][j];
1371             }
1372         }
1373         f << std::endl;
1374     }
1375     f.close();
1376 }
1377
1407 template <typename REAL>
1408 inline void readMatrixFromFileDat(const std::string& filename,
1409                                   DenseMatrix<REAL>& A) {
1410     std::string buffer;
1411     std::ifstream fin(filename.c_str());
1412     std::size_t i = 0;
1413     std::size_t j = 0;
1414     if (fin.is_open()) {
1415         while (std::getline(fin, buffer)) {
1416             std::istringstream iss(buffer);
1417             hdnun::Vector<REAL> rowvector;
1418             while (iss) {
1419                 std::string sub;
1420                 iss >> sub;
1421                 // std::cout << " sub = " << sub.c_str() << ": ";

```

```

1422         if (sub.length() > 0) {
1423             REAL a = atof(sub.c_str());
1424             // std::cout << std::fixed << std::setw(10) <<
1425             // std::setprecision(5) << a;
1426             rowvector.push_back(a);
1427         }
1428         j++;
1429     }
1430     if (rowvector.size() > 0) {
1431         A.addNewRow(rowvector);
1432         i++;
1433         // std::cout << std::endl;
1434     }
1435 }
1436 fin.close();
1437 } else {
1438     HDNUM_ERROR("Could not open file!");
1439 }
1440 }
1441
1442 template <typename REAL>
1443 inline void readMatrixFromFileMatrixMarket(const std::string& filename,
1444                                             DenseMatrix<REAL>& A) {
1445     std::string buffer;
1446     std::ifstream fin(filename.c_str());
1447     std::size_t i = 0;
1448     std::size_t j = 0;
1449     if (fin.is_open()) {
1450         // ignore all comments from the file (starting with %)
1451         while (fin.peek() == '%') fin.ignore(2048, '\n');
1452
1453         std::getline(fin, buffer);
1454         std::istringstream first_line(buffer);
1455         first_line >> i >> j;
1456         DenseMatrix<REAL> A_temp(i, j);
1457
1458         while (std::getline(fin, buffer)) {
1459             std::istringstream iss(buffer);
1460
1461             REAL value {};
1462             iss >> i >> j >> value;
1463             // i-1, j-1, because matrix market does not use zero based indexing
1464             A_temp(i - 1, j - 1) = value;
1465         }
1466         A = A_temp;
1467         fin.close();
1468     } else {
1469         HDNUM_ERROR("Could not open file! \"" + filename + "\"");
1470     }
1471 }
1472
1473 } // namespace hdnum
1474
1475 #endif // DENSEMATRIX_HH

```

## 5.2 src/exceptions.hh File Reference

A few common exception classes.

```

#include <string>
#include <sstream>

```

### Classes

- class `hdnum::Exception`  
*Base class for Exceptions.*
- class `hdnum::IOError`  
*Default exception class for I/O errors.*
- class `hdnum::MathError`  
*Default exception class for mathematical errors.*

- class `hdnum::RangeError`  
*Default exception class for range errors.*
- class `hdnum::NotImplemented`  
*Default exception for dummy implementations.*
- class `hdnum::SystemError`  
*Default exception class for OS errors.*
- class `hdnum::OutOfMemoryError`  
*Default exception if memory allocation fails.*
- class `hdnum::InvalidStateException`  
*Default exception if a function was called while the object is not in a valid state for that function.*
- class `hdnum::ErrorException`  
*General Error.*

## Macros

- `#define THROWSPEC(E) #E << ": "`
- `#define HDNUM_THROW(E, m)`
- `#define HDNUM_ERROR(m)`

## Functions

- `std::ostream & hdnum::operator<< (std::ostream &stream, const Exception &e)`

### 5.2.1 Detailed Description

A few common exception classes.

This file defines a common framework for generating exception subclasses and to throw them in a simple manner.  
Taken from the DUNE project [www.dune-project.org](http://www.dune-project.org)

### 5.2.2 Macro Definition Documentation

#### 5.2.2.1 HDNUM\_ERROR

```
#define HDNUM_ERROR(  
    m )
```

**Value:**

```
do { hdnum::ErrorException th__ex; std::ostringstream th__out; \
    th__out << THROWSPEC(hdnum::ErrorException) << m; \
    th__ex.message(th__out.str()); \
    std::cout << th__ex.what() << std::endl; \
    throw th__ex; \
} while (0)
```

### 5.2.2.2 HDNUM\_THROW

```
#define HDNUM_THROW(  
    E,  
    m )
```

**Value:**

```
do { E th__ex; std::ostringstream th__out;  
    th__out << THROWSPEC(E) << m; th__ex.message(th__out.str()); throw th__ex; \  
} while (0)
```

Macro to throw an exception

## Parameters

<i>E</i>	exception class derived from <code>Dune::Exception</code>
<i>m</i>	reason for this exception in ostream-notation

## Example:

```

    if (filehandle == 0)
DUNE_THROW(FileError, "Could not open " « filename « " for reading!")

```

`DUNE_THROW` automatically adds information about the exception thrown to the text. If `DUNE_DEVEL_MODE` is defined more detail about the function where the exception happened is included. This mode can be activated via the `--enable-dunedevel` switch of `./configure`

## 5.3 exceptions.hh

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

```

1 #ifndef HDNUM_EXCEPTIONS_HH
2 #define HDNUM_EXCEPTIONS_HH
3
4 #include <string>
5 #include <sstream>
6
7 namespace hdnum {
8
9     class Exception {
10     public:
11         void message(const std::string &message);
12         const std::string& what() const;
13     private:
14         std::string _message;
15     };
16
17     inline void Exception::message(const std::string &message)
18     {
19         _message = message;
20     }
21
22     inline const std::string& Exception::what() const
23     {
24         return _message;
25     }
26
27     inline std::ostream& operator<<(std::ostream &stream, const Exception &e)
28     {
29         return stream << e.what();
30     }
31
32     // the "format" the exception-type gets printed. __FILE__ and
33     // __LINE__ are standard C-defines, the GNU cpp-ifile claims that
34     // C99 defines __func__ as well. __FUNCTION__ is a GNU-extension
35     #ifndef HDNUM_DEVEL_MODE
36     #define THROWSPEC(E) #E < " [" < __func__ < ":" < __FILE__ < ":" < __LINE__ < "]: "
37     #else
38     #define THROWSPEC(E) #E < ": "
39     #endif
40
41     // this is the magic: use the usual do { ... } while (0) trick, create
42     // the full message via a string stream and throw the created object
43     #define HDNUM_THROW(E, m) do { E th_ex; std::ostringstream th_out;
44         th_out << THROWSPEC(E) < m; th_ex.message(th_out.str()); throw th_ex; \
45     } while (0)
46
47     class IOError : public Exception {};
48
49     class MathError : public Exception {};
50
51     class RangeError : public Exception {};
52
53     class NotImplemented : public Exception {};
54
55     class SystemError : public Exception {};
56
57     class OutOfMemoryError : public SystemError {};
58
59

```

```

149  class InvalidStateException : public Exception {};
150
153  class ErrorException : public Exception {};
154
155  // throw ErrorException with message
156  #define HDNUM_ERROR(m) do { hdnum::ErrorException th__ex; std::ostringstream th__out; \
157      th__out << THROWSPEC(hdnum::ErrorException) << m; \
158      th__ex.message(th__out.str()); \
159      std::cout << th__ex.what() << std::endl; \
160      throw th__ex; \
161  } while (0)
162
163 } // end namespace
164
165 #endif

```

## 5.4 src/lr.hh File Reference

This file implements LU decomposition.

```

#include "vector.hh"
#include "densematrix.hh"

```

### Functions

- template<class T >  
void **hdnum::lr** (DenseMatrix< T > &A, Vector< std::size\_t > &p)  
*compute lr decomposition of A with first nonzero pivoting*
- template<class T >  
T **hdnum::abs** (const T &t)  
*our own abs class that works also for multiprecision types*
- template<class T >  
void **hdnum::lr\_partialpivot** (DenseMatrix< T > &A, Vector< std::size\_t > &p)  
*lr decomposition of A with column pivoting*
- template<class T >  
void **hdnum::lr\_fullpivot** (DenseMatrix< T > &A, Vector< std::size\_t > &p, Vector< std::size\_t > &q)  
*lr decomposition of A with full pivoting*
- template<class T >  
void **hdnum::permute\_forward** (const Vector< std::size\_t > &p, Vector< T > &b)  
*apply permutations to a right hand side vector*
- template<class T >  
void **hdnum::permute\_backward** (const Vector< std::size\_t > &q, Vector< T > &z)  
*apply permutations to a solution vector*
- template<class T >  
void **hdnum::row\_equilibrate** (DenseMatrix< T > &A, Vector< T > &s)  
*perform a row equilibration of a matrix; return scaling for later use*
- template<class T >  
void **hdnum::apply\_equilibrate** (Vector< T > &s, Vector< T > &b)  
*apply row equilibration to right hand side vector*
- template<class T >  
void **hdnum::solveL** (const DenseMatrix< T > &A, Vector< T > &x, const Vector< T > &b)  
*Assume L = lower triangle of A with l<sub>ii</sub>=1, solve L x = b.*
- template<class T >  
void **hdnum::solveR** (const DenseMatrix< T > &A, Vector< T > &x, const Vector< T > &b)  
*Assume R = upper triangle of A and solve R x = b.*
- template<class T >  
void **hdnum::linsolve** (DenseMatrix< T > &A, Vector< T > &x, Vector< T > &b)  
*a complete solver; Note A, x and b are modified!*

### 5.4.1 Detailed Description

This file implements LU decomposition.

## 5.5 lr.hh

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

```

1 // -*- tab-width: 4; indent-tabs-mode: nil -*-
2 #ifndef HDNUM_LR_HH
3 #define HDNUM_LR_HH
4
5 #include "vector.hh"
6 #include "densematrix.hh"
7
12 namespace hdnum {
13
14     template<class T>
15     void lr (DenseMatrix<T>& A, Vector<std::size_t>& p)
16     {
17         if (A.rowsize() != A.colsize() || A.rowsize() == 0)
18             HDNUM_ERROR("need square and nonempty matrix");
19         if (A.rowsize() != p.size())
20             HDNUM_ERROR("permutation vector incompatible with matrix");
21
22         // transformation to upper triangular
23         for (std::size_t k=0; k<A.rowsize()-1; ++k)
24         {
25             // find pivot element and exchange rows
26             for (std::size_t r=k; r<A.rowsize(); ++r)
27                 if (A[r][k] != 0)
28                 {
29                     p[k] = r; // store permutation in step k
30                     if (r>k) // exchange complete row if r!=k
31                         for (std::size_t j=0; j<A.colsize(); ++j)
32                         {
33                             T temp(A[k][j]);
34                             A[k][j] = A[r][j];
35                             A[r][j] = temp;
36                         }
37                     break;
38                 }
39             if (A[k][k] == 0) HDNUM_ERROR("matrix is singular");
40
41             // modification
42             for (std::size_t i=k+1; i<A.rowsize(); ++i)
43             {
44                 T qik(A[i][k]/A[k][k]);
45                 A[i][k] = qik;
46                 for (std::size_t j=k+1; j<A.colsize(); ++j)
47                     A[i][j] -= qik * A[k][j];
48             }
49         }
50     }
51
52     template<class T>
53     T abs (const T& t)
54     {
55         if (t<0.0)
56             return -t;
57         else
58             return t;
59     }
60
61     template<class T>
62     void lr_partialpivot (DenseMatrix<T>& A, Vector<std::size_t>& p)
63     {
64         if (A.rowsize() != A.colsize() || A.rowsize() == 0)
65             HDNUM_ERROR("need square and nonempty matrix");
66         if (A.rowsize() != p.size())
67             HDNUM_ERROR("permutation vector incompatible with matrix");
68
69         // initialize permutation
70         for (std::size_t k=0; k<A.rowsize(); ++k)
71             p[k] = k;
72
73         // transformation to upper triangular
74         for (std::size_t k=0; k<A.rowsize()-1; ++k)
75         {
76             // find pivot element

```



```

80     for (std::size_t r=k+1; r<A.rowsize(); ++r)
81         if (abs(A[r][k])>abs(A[k][k]))
82             p[k] = r; // store permutation in step k
83
84     if (p[k]>k) // exchange complete row if r!=k
85         for (std::size_t j=0; j<A.colsize(); ++j)
86             {
87                 T temp(A[k][j]);
88                 A[k][j] = A[p[k]][j];
89                 A[p[k]][j] = temp;
90             }
91
92     if (A[k][k]==0) HDNUM_ERROR("matrix is singular");
93
94     // modification
95     for (std::size_t i=k+1; i<A.rowsize(); ++i)
96         {
97             T qik(A[i][k]/A[k][k]);
98             A[i][k] = qik;
99             for (std::size_t j=k+1; j<A.colsize(); ++j)
100                 A[i][j] -= qik * A[k][j];
101         }
102     }
103 }
104
105 template<class T>
106 void lr_fullpivot (DenseMatrix<T>& A, Vector<std::size_t>& p, Vector<std::size_t>& q)
107 {
108     if (A.rowsize()!=A.colsize() || A.rowsize()==0)
109         HDNUM_ERROR("need square and nonempty matrix");
110     if (A.rowsize()!=p.size())
111         HDNUM_ERROR("permutation vector incompatible with matrix");
112
113     // initialize permutation
114     for (std::size_t k=0; k<A.rowsize(); ++k)
115         p[k] = q[k] = k;
116
117     // transformation to upper triangular
118     for (std::size_t k=0; k<A.rowsize()-1; ++k)
119     {
120         // find pivot element
121         for (std::size_t r=k; r<A.rowsize(); ++r)
122             for (std::size_t s=k; s<A.colsize(); ++s)
123                 if (abs(A[r][s])>abs(A[k][k]))
124                     {
125                         p[k] = r; // store permutation in step k
126                         q[k] = s;
127                     }
128
129         if (p[k]>k) // exchange complete row if r!=k
130             for (std::size_t j=0; j<A.colsize(); ++j)
131                 {
132                     T temp(A[k][j]);
133                     A[k][j] = A[p[k]][j];
134                     A[p[k]][j] = temp;
135                 }
136         if (q[k]>k) // exchange complete column if s!=k
137             for (std::size_t i=0; i<A.rowsize(); ++i)
138                 {
139                     T temp(A[i][k]);
140                     A[i][k] = A[i][q[k]];
141                     A[i][q[k]] = temp;
142                 }
143
144         if (std::abs(A[k][k])==0) HDNUM_ERROR("matrix is singular");
145
146         // modification
147         for (std::size_t i=k+1; i<A.rowsize(); ++i)
148             {
149                 T qik(A[i][k]/A[k][k]);
150                 A[i][k] = qik;
151                 for (std::size_t j=k+1; j<A.colsize(); ++j)
152                     A[i][j] -= qik * A[k][j];
153             }
154     }
155 }
156
157 template<class T>
158 void permute_forward (const Vector<std::size_t>& p, Vector<T>& b)
159 {
160     if (b.size()!=p.size())
161         HDNUM_ERROR("permutation vector incompatible with rhs");
162     for (std::size_t k=0; k<b.size()-1; ++k)
163         if (p[k]!=k) std::swap(b[k],b[p[k]]);
164 }
165
166
167
168

```

```

170 template<class T>
171 void permute_backward (const Vector<std::size_t>& q, Vector<T>& z)
172 {
173     if (z.size()!=q.size())
174         HDNUM_ERROR("permutation vector incompatible with z");
175
176     for (int k=z.size()-2; k>=0; --k)
177         if (q[k]!=std::size_t(k)) std::swap(z[k],z[q[k]]);
178 }
179
180 template<class T>
181 void row_equilibrate (DenseMatrix<T>& A, Vector<T>& s)
182 {
183     if (A.rowsize()*A.colsize()==0)
184         HDNUM_ERROR("need nonempty matrix");
185     if (A.rowsize()!=s.size())
186         HDNUM_ERROR("scaling vector incompatible with matrix");
187
188     // equilibrate row sums
189     for (std::size_t k=0; k<A.rowsize(); ++k)
190     {
191         s[k] = T(0.0);
192         for (std::size_t j=0; j<A.colsize(); ++j)
193             s[k] += abs(A[k][j]);
194         if (std::abs(s[k])==0) HDNUM_ERROR("row sum is zero");
195         for (std::size_t j=0; j<A.colsize(); ++j)
196             A[k][j] /= s[k];
197     }
198 }
199
200 template<class T>
201 void apply_equilibrate (Vector<T>& s, Vector<T>& b)
202 {
203     if (s.size()!=b.size())
204         HDNUM_ERROR("s and b incompatible");
205
206     // equilibrate row sums
207     for (std::size_t k=0; k<b.size(); ++k)
208         b[k] /= s[k];
209 }
210
211 template<class T>
212 void solveL (const DenseMatrix<T>& A, Vector<T>& x, const Vector<T>& b)
213 {
214     if (A.rowsize()!=A.colsize() || A.rowsize()==0)
215         HDNUM_ERROR("need square and nonempty matrix");
216     if (A.rowsize()!=b.size())
217         HDNUM_ERROR("right hand side incompatible with matrix");
218
219     for (std::size_t i=0; i<A.rowsize(); ++i)
220     {
221         T rhs(b[i]);
222         for (std::size_t j=0; j<i; j++)
223             rhs -= A[i][j] * x[j];
224         x[i] = rhs;
225     }
226 }
227
228 template<class T>
229 void solveR (const DenseMatrix<T>& A, Vector<T>& x, const Vector<T>& b)
230 {
231     if (A.rowsize()!=A.colsize() || A.rowsize()==0)
232         HDNUM_ERROR("need square and nonempty matrix");
233     if (A.rowsize()!=b.size())
234         HDNUM_ERROR("right hand side incompatible with matrix");
235
236     for (int i=A.rowsize()-1; i>=0; --i)
237     {
238         T rhs(b[i]);
239         for (std::size_t j=i+1; j<A.colsize(); j++)
240             rhs -= A[i][j] * x[j];
241         x[i] = rhs/A[i][i];
242     }
243 }
244
245 template<class T>
246 void linsolve (DenseMatrix<T>& A, Vector<T>& x, Vector<T>& b)
247 {
248     if (A.rowsize()!=A.colsize() || A.rowsize()==0)
249         HDNUM_ERROR("need square and nonempty matrix");
250     if (A.rowsize()!=b.size())
251         HDNUM_ERROR("right hand side incompatible with matrix");
252
253     Vector<T> s(x.size());
254     Vector<std::size_t> p(x.size());
255     Vector<std::size_t> q(x.size());
256     row_equilibrate(A,s);
257 }

```

```

262     lr_fullpivot(A, p, q);
263     apply_equilibrate(s, b);
264     permute_forward(p, b);
265     solveL(A, b, b);
266     solveR(A, x, b);
267     permute_backward(q, x);
268 }
269
270 }
271 #endif

```

## 5.6 src/newton.hh File Reference

Newton's method with line search.

```

#include "lr.hh"
#include <type_traits>

```

### Classes

- class `hdnum::SquareRootProblem< N >`  
*Example class for a nonlinear model  $F(x) = 0$ .*
- class `hdnum::GenericNonlinearProblem< Lambda, Vec >`  
*A generic problem class that can be set up with a lambda defining  $F(x)=0$ .*
- class `hdnum::Newton`  
*Solve nonlinear problem using a damped [Newton](#) method.*
- class `hdnum::Banach`  
*Solve nonlinear problem using a fixed point iteration.*

### Functions

- `template<typename F, typename X >`  
`GenericNonlinearProblem< F, X > hdnum::getNonlinearProblem (const F &f, const X &x, typename X::value_type eps=1e-7)`  
*A function returning a problem class.*

#### 5.6.1 Detailed Description

Newton's method with line search.

#### 5.6.2 Function Documentation

##### 5.6.2.1 getNonlinearProblem()

```

template<typename F, typename X >
GenericNonlinearProblem< F, X > hdnum::getNonlinearProblem (
    const F & f,
    const X & x,
    typename X::value_type eps = 1e-7 )

```

A function returning a problem class.

Automatic template parameter extraction makes fiddling with types unnecessary.

## Template Parameters

<i>F</i>	a lambda mapping a Vector to a Vector
<i>X</i>	the type for the Vector

## 5.7 newton.hh

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

```

1 // -*- tab-width: 4; indent-tabs-mode: nil -*-
2 #ifndef HDNUM_NEWTON_HH
3 #define HDNUM_NEWTON_HH
4
5 #include "lr.hh"
6 #include <type_traits>
7
12 namespace hdnum {
13
20     template<class N>
21     class SquareRootProblem
22     {
23     public:
25         typedef std::size_t size_type;
26
28         typedef N number_type;
29
31         SquareRootProblem (number_type a_)
32             : a(a_)
33         {}
34
36         std::size_t size () const
37         {
38             return 1;
39         }
40
42         void F (const Vector<N>& x, Vector<N>& result) const
43         {
44             result[0] = x[0]*x[0] - a;
45         }
46
48         void F_x (const Vector<N>& x, DenseMatrix<N>& result) const
49         {
50             result[0][0] = number_type(2.0)*x[0];
51         }
52
53     private:
54         number_type a;
55     };
56
57
63     template<typename Lambda, typename Vec>
64     class GenericNonlinearProblem
65     {
66     public:
67         Lambda lambda; // lambda defining the problem "lambda(x)=0"
68         size_t s;
69         typename Vec::value_type eps;
70
72         typedef std::size_t size_type;
73
75         typedef typename Vec::value_type number_type;
76
78         GenericNonlinearProblem (const Lambda& l_, const Vec& x_, number_type eps_ = 1e-7)
79             : lambda(l_), s(x_.size()), eps(eps_)
80         {}
81
83         std::size_t size () const
84         {
85             return s;
86         }
87
89         void F (const Vec& x, Vec& result) const
90         {
91             result = lambda(x);
92         }
93
95         void F_x (const Vec& x, DenseMatrix<number_type>& result) const
96         {

```

```

97     Vec Fx(x.size());
98     F(x,Fx);
99     Vec z(x);
100     Vec Fz(x.size());
101
102     // numerische Jacobimatrix
103     for (int j=0; j<result.colsize(); ++j)
104     {
105         auto zj = z[j];
106         auto dz = (1.0+abs(zj))*eps;
107         z[j] += dz;
108         F(z,Fz);
109         for (int i=0; i<result.rowsize(); i++)
110             result[i][j] = (Fz[i]-Fx[i])/dz;
111         z[j] = zj;
112     }
113 }
114 };
115
116 template<typename F, typename X>
117 GenericNonlinearProblem<F,X> getNonlinearProblem (const F& f, const X& x, typename X::value_type eps =
118     1e-7)
119 {
120     return GenericNonlinearProblem<F,X>(f,x,eps);
121 }
122
123 class Newton
124 {
125     typedef std::size_t size_type;
126
127 public:
128     Newton ()
129         : maxit(25), linesearchsteps(10), verbosity(0),
130           reduction(1e-14), abslimit(1e-30), converged(false)
131     {}
132
133     void set_maxit (size_type n)
134     {
135         maxit = n;
136     }
137
138     void set_sigma (double sigma_)
139     {
140     }
141
142     void set_linesearchsteps (size_type n)
143     {
144         linesearchsteps = n;
145     }
146
147     void set_verbosity (size_type n)
148     {
149         verbosity = n;
150     }
151
152     void set_abslimit (double l)
153     {
154         abslimit = l;
155     }
156
157     void set_reduction (double l)
158     {
159         reduction = l;
160     }
161
162     template<class M>
163     void solve (const M& model, Vector<typename M::number_type> & x) const
164     {
165         typedef typename M::number_type N;
166         // In complex case, we still need to use real valued numbers for residual norms etc.
167         using Real = typename std::conditional<std::is_same<std::complex<double>, N::value, double>,
168             N::type;
169         Vector<N> r(model.size()); // residual
170         DenseMatrix<N> A(model.size(),model.size()); // Jacobian matrix
171         Vector<N> y(model.size()); // temporary solution in line search
172         Vector<N> z(model.size()); // solution of linear system
173         Vector<N> s(model.size()); // scaling factors
174         Vector<size_type> p(model.size()); // row permutations
175         Vector<size_type> q(model.size()); // column permutations
176
177         model.F(x,r); // compute nonlinear residual
178         Real R0(std::abs(norm(r))); // norm of initial residual
179         Real R(R0); // current residual norm
180         if (verbosity>=1)
181         {
182             std::cout << "Newton "

```

```

202         « "   norm=" « std::scientific « std::showpoint
203         « std::setprecision(4) « R0
204         « std::endl;
205     }
206
207     converged = false;
208     for (size_type i=1; i<=maxit; i++)                // do Newton iterations
209     {
210         // check absolute size of residual
211         if (R<=abslimit)
212         {
213             converged = true;
214             return;
215         }
216
217         // solve Jacobian system for update
218         model.F_x(x,A);                               // compute Jacobian matrix
219         row_equilibrate(A,s);                          // equilibrate rows
220         lr_fullpivot(A,p,q);                          // LR decomposition of A
221         z = N(0.0);                                    // clear solution
222         apply_equilibrate(s,r);                        // equilibration of right hand side
223         permute_forward(p,r);                         // permutation of right hand side
224         solveL(A,r,r);                                // forward substitution
225         solveR(A,z,r);                                // backward substitution
226         permute_backward(q,z);                        // backward permutation
227
228         // line search
229         Real lambda(1.0);                             // start with lambda=1
230         for (size_type k=0; k<linesearchsteps; k++)
231         {
232             y = x;
233             y.update(-lambda,z);                      // y = x+lambda*z
234             model.F(y,r);                             // r = F(y)
235             Real newR(std::abs(norm(r)));              // compute norm
236             if (verbosity>=3)
237             {
238                 std::cout « "   line search " « std::setw(2) « k
239                 « " lambda=" « std::scientific « std::showpoint
240                 « std::setprecision(4) « lambda
241                 « " norm=" « std::scientific « std::showpoint
242                 « std::setprecision(4) « newR
243                 « " red=" « std::scientific « std::showpoint
244                 « std::setprecision(4) « newR/R
245                 « std::endl;
246             }
247             if (newR<(1.0-0.25*lambda)*R)              // check convergence
248             {
249                 if (verbosity>=2)
250                 {
251                     std::cout « "   step" « std::setw(3) « i
252                     « " norm=" « std::scientific « std::showpoint
253                     « std::setprecision(4) « newR
254                     « " red=" « std::scientific « std::showpoint
255                     « std::setprecision(4) « newR/R
256                     « std::endl;
257                 }
258                 x = y;
259                 R = newR;
260                 break;                                // continue with Newton loop
261             }
262             else lambda *= 0.5;                        // reduce damping factor
263             if (k==linesearchsteps-1)
264             {
265                 if (verbosity>=3)
266                 {
267                     std::cout « "   line search not converged within " « linesearchsteps « " steps" «
268                     std::endl;
269                 }
270                 return;
271             }
272             // check convergence
273             if (R<=reduction*R0)
274             {
275                 if (verbosity>=1)
276                 {
277                     std::cout « "Newton converged in " « i « " steps"
278                     « " reduction=" « std::scientific « std::showpoint
279                     « std::setprecision(4) « R/R0
280                     « std::endl;
281                 }
282                 iterations_taken = i;
283                 converged = true;
284                 return;
285             }
286             if (i==maxit)
287             {
288                 iterations_taken = i;

```

```

288         if (verbosity>=1)
289             std::cout << "Newton not converged within " << maxit << " iterations" << std::endl;
290     }
291 }
292 }
293
294 bool has_converged () const
295 {
296     return converged;
297 }
298 size_type iterations() const {
299     return iterations_taken;
300 }
301
302
303 private:
304     size_type maxit;
305     mutable size_type iterations_taken = -1;
306     size_type linesearchsteps;
307     size_type verbosity;
308     double reduction;
309     double abslimit;
310     mutable bool converged;
311 };
312
313
314
315
316
317
318
319
320
321
322
323 class Banach
324 {
325     typedef std::size_t size_type;
326
327 public:
328     Banach ()
329         : maxit(25), linesearchsteps(10), verbosity(0),
330           reduction(1e-14), abslimit(1e-30), sigma(1.0), converged(false)
331     {}
332
333     void set_maxit (size_type n)
334     {
335         maxit = n;
336     }
337
338     void set_sigma (double sigma_)
339     {
340         sigma = sigma_;
341     }
342
343     void set_linesearchsteps (size_type n)
344     {
345         linesearchsteps = n;
346     }
347
348     void set_verbosity (size_type n)
349     {
350         verbosity = n;
351     }
352
353     void set_abslimit (double l)
354     {
355         abslimit = l;
356     }
357
358     void set_reduction (double l)
359     {
360         reduction = l;
361     }
362
363     template<class M>
364     void solve (const M& model, Vector<typename M::number_type>& x) const
365     {
366         typedef typename M::number_type N;
367         Vector<N> r(model.size()); // residual
368         Vector<N> y(model.size()); // temporary solution in line search
369
370         model.F(x,r); // compute nonlinear residual
371         N R0(norm(r)); // norm of initial residual
372         N R(R0); // current residual norm
373         if (verbosity>=1)
374         {
375             std::cout << "Banach "
376                       << " norm=" << std::scientific << std::showpoint
377                       << std::setprecision(4) << R0
378                       << std::endl;
379         }
380         converged = false;

```

```

390     for (size_type i=1; i<=maxit; i++)                // do iterations
391     {
392         // check absolute size of residual
393         if (R<=abslimit)
394         {
395             converged = true;
396             return;
397         }
398
399         // next iterate
400         y = x;
401         y.update(-sigma,r);                          // y = x+lambda*z
402         model.F(y,r);                                // r = F(y)
403         N newR(norm(r));                             // compute norm
404         if (verbosity>=2)
405         {
406             std::cout << " " << std::setw(3) << i
407             << " norm=" << std::scientific << std::showpoint
408             << std::setprecision(4) << newR
409             << " red=" << std::scientific << std::showpoint
410             << std::setprecision(4) << newR/R
411             << std::endl;
412         }
413         x = y;                                        // accept new iterate
414         R = newR;                                    // remember new norm
415
416         // check convergence
417         if (R<=reduction*R0 || R<=abslimit)
418         {
419             if (verbosity>=1)
420             {
421                 std::cout << "Banach converged in " << i << " steps"
422                 << " reduction=" << std::scientific << std::showpoint
423                 << std::setprecision(4) << R/R0
424                 << std::endl;
425             }
426             converged = true;
427             return;
428         }
429     }
430 }
431
432 bool has_converged () const
433 {
434     return converged;
435 }
436
437 private:
438     size_type maxit;
439     size_type linesearchsteps;
440     size_type verbosity;
441     double reduction;
442     double abslimit;
443     double sigma;
444     mutable bool converged;
445 };
446
447 } // namespace hdnum
448
449 #endif

```

## 5.8 src/ode.hh File Reference

solvers for ordinary differential equations

```

#include <vector>
#include "newton.hh"

```

### Classes

- class [hdnum::EE< M >](#)  
*Explicit Euler method as an example for an ODE solver.*
- class [hdnum::ModifiedEuler< M >](#)



- Modified Euler method (order 2 with 2 stages)
  - class `hdnum::Heun2< M >`
- Heun method (order 2 with 2 stages)
  - class `hdnum::Heun3< M >`
- Heun method (order 3 with 3 stages)
  - class `hdnum::Kutta3< M >`
- Kutta method (order 3 with 3 stages)
  - class `hdnum::RungeKutta4< M >`
- classical Runge-Kutta method (order 4 with 4 stages)
  - class `hdnum::RKF45< M >`
- Adaptive Runge-Kutta-Fehlberg method.
  - class `hdnum::RE< M, S >`
- Adaptive one-step method using Richardson extrapolation.
  - class `hdnum::IE< M, S >`
- Implicit Euler using [Newton's](#) method to solve nonlinear system.
  - class `hdnum::DIRK< M, S >`
- Implementation of a general Diagonal Implicit Runge-Kutta method.

## Functions

- template<class T , class N >  
 void **hdnum::gnuplot** (const std::string &fname, const std::vector< T > t, const std::vector< Vector< N >  
 > u)  
 gnuplot output for time and state sequence
- template<class T , class N >  
 void **hdnum::gnuplot** (const std::string &fname, const std::vector< T > t, const std::vector< Vector< N >  
 > u, const std::vector< T > dt)  
 gnuplot output for time and state sequence

### 5.8.1 Detailed Description

solvers for ordinary differential equations

## 5.9 ode.hh

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

```

1 // -*- tab-width: 4; indent-tabs-mode: nil -*-
2 #ifndef HDNUM_ODE_HH
3 #define HDNUM_ODE_HH
4
5 #include<vector>
6 #include "newton.hh"
7
12 namespace hdnum {
13
22   template<class M>
23   class EE
24   {
25   public:
27     typedef typename M::size_type size_type;
28
30     typedef typename M::time_type time_type;
31
33     typedef typename M::number_type number_type;
34
36     EE (const M& model_)
```

```

37     : model(model_), u(model.size()), f(model.size())
38     {
39         model.initialize(t,u);
40         dt = 0.1;
41     }
42
43
44     void set_dt (time_type dt_)
45     {
46         dt = dt_;
47     }
48
49
50     void step ()
51     {
52         model.f(t,u,f); // evaluate model
53         u.update(dt,f); // advance state
54         t += dt; // advance time
55     }
56
57
58     void set_state (time_type t_, const Vector<number_type>& u_)
59     {
60         t = t_;
61         u = u_;
62     }
63
64
65     const Vector<number_type>& get_state () const
66     {
67         return u;
68     }
69
70
71     time_type get_time () const
72     {
73         return t;
74     }
75
76
77     time_type get_dt () const
78     {
79         return dt;
80     }
81
82
83     size_type get_order () const
84     {
85         return 1;
86     }
87
88 private:
89     const M& model;
90     time_type t, dt;
91     Vector<number_type> u;
92     Vector<number_type> f;
93 };
94
95
96 template<class M>
97 class ModifiedEuler
98 {
99 public:
100     typedef typename M::size_type size_type;
101
102     typedef typename M::time_type time_type;
103
104     typedef typename M::number_type number_type;
105
106     ModifiedEuler (const M& model_)
107         : model(model_), u(model.size()), w(model.size()), k1(model.size()), k2(model.size())
108     {
109         c2 = 0.5;
110         a21 = 0.5;
111         b2 = 1.0;
112         model.initialize(t,u);
113         dt = 0.1;
114     }
115
116
117     void set_dt (time_type dt_)
118     {
119         dt = dt_;
120     }
121
122
123     void step ()
124     {
125         // stage 1
126         model.f(t,u,k1);
127
128         // stage 2
129         w = u;
130         w.update(dt*a21,k1);
131         model.f(t+c2*dt,w,k2);
132
133         // final

```

```

145     u.update(dt*b2,k2);
146     t += dt;
147 }
148
149 void set_state (time_type t_, const Vector<number_type>& u_)
150 {
151     t = t_;
152     u = u_;
153 }
154
155 const Vector<number_type>& get_state () const
156 {
157     return u;
158 }
159
160 time_type get_time () const
161 {
162     return t;
163 }
164
165 time_type get_dt () const
166 {
167     return dt;
168 }
169
170 size_type get_order () const
171 {
172     return 2;
173 }
174
175 private:
176     const M& model;
177     time_type t, dt;
178     time_type c2,a21,b2;
179     Vector<number_type> u,w;
180     Vector<number_type> k1,k2;
181 };
182
183 template<class M>
184 class Heun2
185 {
186 public:
187     typedef typename M::size_type size_type;
188     typedef typename M::time_type time_type;
189     typedef typename M::number_type number_type;
190
191     Heun2 (const M& model_)
192         : model(model_), u(model.size()), w(model.size()), k1(model.size()), k2(model.size())
193     {
194         c2 = 1.0;
195         a21 = 1.0;
196         b1 = 0.5;
197         b2 = 0.5;
198         model.initialize(t,u);
199         dt = 0.1;
200     }
201
202     void set_dt (time_type dt_)
203     {
204         dt = dt_;
205     }
206
207     void step ()
208     {
209         // stage 1
210         model.f(t,u,k1);
211
212         // stage 2
213         w = u;
214         w.update(dt*a21,k1);
215         model.f(t+c2*dt,w,k2);
216
217         // final
218         u.update(dt*b1,k1);
219         u.update(dt*b2,k2);
220         t += dt;
221     }
222
223     void set_state (time_type t_, const Vector<number_type>& u_)
224     {
225         t = t_;
226         u = u_;
227     }
228 }

```

```

253     const Vector<number_type>& get_state () const
254     {
255         return u;
256     }
257
258     time_type get_time () const
259     {
260         return t;
261     }
262
263     time_type get_dt () const
264     {
265         return dt;
266     }
267
268     size_type get_order () const
269     {
270         return 2;
271     }
272
273 private:
274     const M& model;
275     time_type t, dt;
276     time_type c2, a21, b1, b2;
277     Vector<number_type> u, w;
278     Vector<number_type> k1, k2;
279 };
280
281 template<class M>
282 class Heun3
283 {
284 public:
285     typedef typename M::size_type size_type;
286
287     typedef typename M::time_type time_type;
288
289     typedef typename M::number_type number_type;
290
291     Heun3 (const M& model_)
292         : model(model_), u(model.size()), w(model.size()), k1(model.size()),
293           k2(model.size()), k3(model.size())
294     {
295         c2 = time_type(1.0)/time_type(3.0);
296         c3 = time_type(2.0)/time_type(3.0);
297         a21 = time_type(1.0)/time_type(3.0);
298         a32 = time_type(2.0)/time_type(3.0);
299         b1 = 0.25;
300         b2 = 0.0;
301         b3 = 0.75;
302         model.initialize(t,u);
303         dt = 0.1;
304     }
305
306     void set_dt (time_type dt_)
307     {
308         dt = dt_;
309     }
310
311     void step ()
312     {
313         // stage 1
314         model.f(t,u,k1);
315
316         // stage 2
317         w = u;
318         w.update(dt*a21,k1);
319         model.f(t+c2*dt,w,k2);
320
321         // stage 3
322         w = u;
323         w.update(dt*a32,k2);
324         model.f(t+c3*dt,w,k3);
325
326         // final
327         u.update(dt*b1,k1);
328         u.update(dt*b3,k3);
329         t += dt;
330     }
331
332     void set_state (time_type t_, const Vector<number_type>& u_)
333     {
334         t = t_;
335         u = u_;
336     }
337
338     const Vector<number_type>& get_state () const

```

```

359     {
360         return u;
361     }
362
363     time_type get_time () const
364     {
365         return t;
366     }
367
368     time_type get_dt () const
369     {
370         return dt;
371     }
372
373     size_type get_order () const
374     {
375         return 3;
376     }
377
378 private:
379     const M& model;
380     time_type t, dt;
381     time_type c2, c3, a21, a31, a32, b1, b2, b3;
382     Vector<number_type> u, w;
383     Vector<number_type> k1, k2, k3;
384 };
385
386 template<class M>
387 class Kutta3
388 {
389 public:
390     typedef typename M::size_type size_type;
391     typedef typename M::time_type time_type;
392     typedef typename M::number_type number_type;
393
394     Kutta3 (const M& model_)
395         : model(model_), u(model.size()), w(model.size()), k1(model.size()),
396           k2(model.size()), k3(model.size())
397     {
398         c2 = 0.5;
399         c3 = 1.0;
400         a21 = 0.5;
401         a31 = -1.0;
402         a32 = 2.0;
403         b1 = time_type(1.0)/time_type(6.0);
404         b2 = time_type(4.0)/time_type(6.0);
405         b3 = time_type(1.0)/time_type(6.0);
406         model.initialize(t, u);
407         dt = 0.1;
408     }
409
410     void set_dt (time_type dt_)
411     {
412         dt = dt_;
413     }
414
415     void step ()
416     {
417         // stage 1
418         model.f(t, u, k1);
419
420         // stage 2
421         w = u;
422         w.update(dt*a21, k1);
423         model.f(t+c2*dt, w, k2);
424
425         // stage 3
426         w = u;
427         w.update(dt*a31, k1);
428         w.update(dt*a32, k2);
429         model.f(t+c3*dt, w, k3);
430
431         // final
432         u.update(dt*b1, k1);
433         u.update(dt*b2, k2);
434         u.update(dt*b3, k3);
435         t += dt;
436     }
437
438     void set_state (time_type t_, const Vector<number_type>& u_)
439     {
440         t = t_;
441         u = u_;
442     }
443 }

```

```

465     const Vector<number_type>& get_state () const
466     {
467         return u;
468     }
469
471     time_type get_time () const
472     {
473         return t;
474     }
475
477     time_type get_dt () const
478     {
479         return dt;
480     }
481
483     size_type get_order () const
484     {
485         return 3;
486     }
487
488 private:
489     const M& model;
490     time_type t, dt;
491     time_type c2,c3,a21,a31,a32,b1,b2,b3;
492     Vector<number_type> u,w;
493     Vector<number_type> k1,k2,k3;
494 };
495
504 template<class M>
505 class RungeKutta4
506 {
507 public:
509     typedef typename M::size_type size_type;
510
512     typedef typename M::time_type time_type;
513
515     typedef typename M::number_type number_type;
516
518     RungeKutta4 (const M& model_)
519         : model(model_), u(model.size()), w(model.size()), k1(model.size()),
520           k2(model.size()), k3(model.size()), k4(model.size())
521     {
522         c2 = 0.5;
523         c3 = 0.5;
524         c4 = 1.0;
525         a21 = 0.5;
526         a32 = 0.5;
527         a43 = 1.0;
528         b1 = time_type(1.0)/time_type(6.0);
529         b2 = time_type(2.0)/time_type(6.0);
530         b3 = time_type(2.0)/time_type(6.0);
531         b4 = time_type(1.0)/time_type(6.0);
532         model.initialize(t,u);
533         dt = 0.1;
534     }
535
537     void set_dt (time_type dt_)
538     {
539         dt = dt_;
540     }
541
543     void step ()
544     {
545         // stage 1
546         model.f(t,u,k1);
547
548         // stage 2
549         w = u;
550         w.update(dt*a21,k1);
551         model.f(t+c2*dt,w,k2);
552
553         // stage 3
554         w = u;
555         w.update(dt*a32,k2);
556         model.f(t+c3*dt,w,k3);
557
558         // stage 4
559         w = u;
560         w.update(dt*a43,k3);
561         model.f(t+c4*dt,w,k4);
562
563         // final
564         u.update(dt*b1,k1);
565         u.update(dt*b2,k2);
566         u.update(dt*b3,k3);
567         u.update(dt*b4,k4);
568         t += dt;

```

```

569     }
570
571     void set_state (time_type t_, const Vector<number_type>& u_)
572     {
573         t = t_;
574         u = u_;
575     }
576
577     const Vector<number_type>& get_state () const
578     {
579         return u;
580     }
581
582     time_type get_time () const
583     {
584         return t;
585     }
586
587     time_type get_dt () const
588     {
589         return dt;
590     }
591
592     size_type get_order () const
593     {
594         return 4;
595     }
596
597 private:
598     const M& model;
599     time_type t, dt;
600     time_type c2, c3, c4, a21, a32, a43, b1, b2, b3, b4;
601     Vector<number_type> u, w;
602     Vector<number_type> k1, k2, k3, k4;
603 };
604
605 template<class M>
606 class RKF45
607 {
608 public:
609     typedef typename M::size_type size_type;
610
611     typedef typename M::time_type time_type;
612
613     typedef typename M::number_type number_type;
614
615     RKF45 (const M& model_)
616         : model(model_), u(model.size()), w(model.size()), ww(model.size()), k1(model.size()),
617           k2(model.size()), k3(model.size()), k4(model.size()), k5(model.size()), k6(model.size()),
618           steps(0), rejected(0)
619     {
620         TOL = time_type(0.0001);
621         rho = time_type(0.8);
622         alpha = time_type(0.25);
623         beta = time_type(4.0);
624         dt_min = 1E-12;
625
626         c2 = time_type(1.0)/time_type(4.0);
627         c3 = time_type(3.0)/time_type(8.0);
628         c4 = time_type(12.0)/time_type(13.0);
629         c5 = time_type(1.0);
630         c6 = time_type(1.0)/time_type(2.0);
631
632         a21 = time_type(1.0)/time_type(4.0);
633
634         a31 = time_type(3.0)/time_type(32.0);
635         a32 = time_type(9.0)/time_type(32.0);
636
637         a41 = time_type(1932.0)/time_type(2197.0);
638         a42 = time_type(-7200.0)/time_type(2197.0);
639         a43 = time_type(7296.0)/time_type(2197.0);
640
641         a51 = time_type(439.0)/time_type(216.0);
642         a52 = time_type(-8.0);
643         a53 = time_type(3680.0)/time_type(513.0);
644         a54 = time_type(-845.0)/time_type(4104.0);
645
646         a61 = time_type(-8.0)/time_type(27.0);
647         a62 = time_type(2.0);
648         a63 = time_type(-3544.0)/time_type(2565.0);
649         a64 = time_type(1859.0)/time_type(4104.0);
650         a65 = time_type(-11.0)/time_type(40.0);
651
652         b1 = time_type(25.0)/time_type(216.0);
653         b2 = time_type(0.0);
654         b3 = time_type(1408.0)/time_type(2565.0);
655         b4 = time_type(2197.0)/time_type(4104.0);

```

```

669     b5 = time_type(-1.0)/time_type(5.0);
670
671     bb1 = time_type(16.0)/time_type(135.0);
672     bb2 = time_type(0.0);
673     bb3 = time_type(6656.0)/time_type(12825.0);
674     bb4 = time_type(28561.0)/time_type(56430.0);
675     bb5 = time_type(-9.0)/time_type(50.0);
676     bb6 = time_type(2.0)/time_type(55.0);
677
678     model.initialize(t,u);
679     dt = 0.1;
680 }
681
682 void set_dt (time_type dt_)
683 {
684     dt = dt_;
685 }
686
687 void set_TOL (time_type TOL_)
688 {
689     TOL = TOL_;
690 }
691
692 void step ()
693 {
694     steps++;
695
696     // stage 1
697     model.f(t,u,k1);
698
699     // stage 2
700     w = u;
701     w.update(dt*a21,k1);
702     model.f(t+c2*dt,w,k2);
703
704     // stage 3
705     w = u;
706     w.update(dt*a31,k1);
707     w.update(dt*a32,k2);
708     model.f(t+c3*dt,w,k3);
709
710     // stage 4
711     w = u;
712     w.update(dt*a41,k1);
713     w.update(dt*a42,k2);
714     w.update(dt*a43,k3);
715     model.f(t+c4*dt,w,k4);
716
717     // stage 5
718     w = u;
719     w.update(dt*a51,k1);
720     w.update(dt*a52,k2);
721     w.update(dt*a53,k3);
722     w.update(dt*a54,k4);
723     model.f(t+c5*dt,w,k5);
724
725     // stage 6
726     w = u;
727     w.update(dt*a61,k1);
728     w.update(dt*a62,k2);
729     w.update(dt*a63,k3);
730     w.update(dt*a64,k4);
731     w.update(dt*a65,k5);
732     model.f(t+c6*dt,w,k6);
733
734     // compute order 4 approximation
735     w = u;
736     w.update(dt*b1,k1);
737     w.update(dt*b2,k2);
738     w.update(dt*b3,k3);
739     w.update(dt*b4,k4);
740     w.update(dt*b5,k5);
741
742     // compute order 5 approximation
743     ww = u;
744     ww.update(dt*bb1,k1);
745     ww.update(dt*bb2,k2);
746     ww.update(dt*bb3,k3);
747     ww.update(dt*bb4,k4);
748     ww.update(dt*bb5,k5);
749     ww.update(dt*bb6,k6);
750
751     // estimate local error
752     w -= ww;
753     time_type error(norm(w));
754     time_type dt_opt(dt*pow(rho*TOL/error,0.2));
755     dt_opt = std::min(beta*dt,std::max(alpha*dt,dt_opt));

```



```

759     //std::cout << "est. error=" << error << " dt_opt=" << dt_opt << std::endl;
760
761     if (error<=TOL)
762     {
763         t += dt;
764         u = ww;
765         dt = dt_opt;
766     }
767     else
768     {
769         rejected++;
770         dt = dt_opt;
771         if (dt>dt_min) step();
772     }
773 }
774
775 const Vector<number_type>& get_state () const
776 {
777     return u;
778 }
779
780 time_type get_time () const
781 {
782     return t;
783 }
784
785 time_type get_dt () const
786 {
787     return dt;
788 }
789
790 size_type get_order () const
791 {
792     return 5;
793 }
794
795 void get_info () const
796 {
797     std::cout << "RE: steps=" << steps << " rejected=" << rejected << std::endl;
798 }
799
800 private:
801 const M& model;
802 time_type t, dt;
803 time_type TOL, rho, alpha, beta, dt_min;
804 time_type c2, c3, c4, c5, c6;
805 time_type a21, a31, a32, a41, a42, a43, a51, a52, a53, a54, a61, a62, a63, a64, a65;
806 time_type b1, b2, b3, b4, b5; // 4th order
807 time_type bb1, bb2, bb3, bb4, bb5, bb6; // 5th order
808 Vector<number_type> u, w, ww;
809 Vector<number_type> k1, k2, k3, k4, k5, k6;
810 mutable size_type steps, rejected;
811 };
812
813 template<class M, class S>
814 class RE
815 {
816 public:
817     typedef typename M::size_type size_type;
818
819     typedef typename M::time_type time_type;
820
821     typedef typename M::number_type number_type;
822
823     RE (const M& model_, S& solver_)
824     : model(model_), solver(solver_), u(model.size()),
825       wlow(model.size()), whigh(model.size()), ww(model.size()),
826       steps(0), rejected(0)
827     {
828         model.initialize(t, u); // initialize state
829         dt = 0.1; // set initial time step
830         two_power_m = 1.0;
831         for (size_type i=0; i<solver.get_order(); i++)
832             two_power_m *= 2.0;
833         TOL = time_type(0.0001);
834         rho = time_type(0.8);
835         alpha = time_type(0.25);
836         beta = time_type(4.0);
837         dt_min = 1E-12;
838     }
839
840     void set_dt (time_type dt_)
841     {
842         dt = dt_;
843     }
844 }

```

```

862 void set_TOL (time_type TOL_)
863 {
864     TOL = TOL_;
865 }
866
867 void step ()
868 {
869     // count steps done
870     steps++;
871
872     // do 1 step with 2*dt
873     time_type H(2.0*dt);
874     solver.set_state(t,u);
875     solver.set_dt(H);
876     solver.step();
877     wlow = solver.get_state();
878
879     // do 2 steps with dt
880     solver.set_state(t,u);
881     solver.set_dt(dt);
882     solver.step();
883     solver.step();
884     whigh = solver.get_state();
885
886     // estimate local error
887     ww = wlow;
888     ww -= whigh;
889     time_type error(norm(ww)/(pow(H,1.0+solver.get_order())*(1.0-1.0/two_power_m)));
890     time_type dt_opt(pow(rho*TOL/error,1.0/((time_type)solver.get_order())));
891     dt_opt = std::min(beta*dt, std::max(alpha*dt, dt_opt));
892     //std::cout << "est. error=" << error << " dt_opt=" << dt_opt << std::endl;
893
894     if (dt<=dt_opt)
895     {
896         t += H;
897         u = whigh;
898         u *= two_power_m;
899         u -= wlow;
900         u /= two_power_m-1.0;
901         dt = dt_opt;
902     }
903     else
904     {
905         rejected++;
906         dt = dt_opt;
907         if (dt>dt_min) step();
908     }
909 }
910
911 const Vector<number_type>& get_state () const
912 {
913     return u;
914 }
915
916 time_type get_time () const
917 {
918     return t;
919 }
920
921 time_type get_dt () const
922 {
923     return dt;
924 }
925
926 size_type get_order () const
927 {
928     return solver.get_order()+1;
929 }
930
931 void get_info () const
932 {
933     std::cout << "RE: steps=" << steps << " rejected=" << rejected << std::endl;
934 }
935
936 private:
937     const M& model;
938     S& solver;
939     time_type t, dt;
940     time_type two_power_m;
941     Vector<number_type> u,wlow,whigh,ww;
942     time_type TOL,rho,alpha,beta,dt_min;
943     mutable size_type steps, rejected;
944 };
945
946 template<class M, class S>
947 class IE

```

```

964 {
965     // h_n f(t_n, y_n) - y_n + y_{n-1} = 0
966     class NonlinearProblem
967     {
968     public:
969         typedef typename M::size_type size_type;
970
971         typedef typename M::number_type number_type;
972
973         NonlinearProblem (const M& model_, const Vector<number_type>& yold_,
974                         typename M::time_type tnew_, typename M::time_type dt_)
975             : model(model_), yold(yold_), tnew(tnew_), dt(dt_)
976         {}
977
978         std::size_t size () const
979         {
980             return model.size();
981         }
982
983         void F (const Vector<number_type>& x, Vector<number_type>& result) const
984         {
985             model.f(tnew,x,result);
986             result *= dt;
987             result -= x;
988             result += yold;
989         }
990
991         void F_x (const Vector<number_type>& x, DenseMatrix<number_type>& result) const
992         {
993             model.f_x(tnew,x,result);
994             result *= dt;
995             for (size_type i=0; i<model.size(); i++) result[i][i] -= number_type(1.0);
996         }
997
998         void set_tnew_dt (typename M::time_type tnew_, typename M::time_type dt_)
999         {
1000             tnew = tnew_;
1001             dt = dt_;
1002         }
1003
1004     private:
1005         const M& model;
1006         const Vector<number_type>& yold;
1007         typename M::time_type tnew;
1008         typename M::time_type dt;
1009     };
1010
1011 public:
1012     typedef typename M::size_type size_type;
1013
1014     typedef typename M::time_type time_type;
1015
1016     typedef typename M::number_type number_type;
1017
1018     IE (const M& model_, const S& newton_)
1019         : verbosity(0), model(model_), newton(newton_), u(model.size()), unew(model.size())
1020     {
1021         model.initialize(t,u);
1022         dt = dtmax = 0.1;
1023     }
1024
1025     void set_dt (time_type dt_)
1026     {
1027         dt = dtmax = dt_;
1028     }
1029
1030     void set_verbosity (size_type verbosity_)
1031     {
1032         verbosity = verbosity_;
1033     }
1034
1035     void step ()
1036     {
1037         if (verbosity>=2)
1038             std::cout << "IE: step" << " t=" << t << " dt=" << dt << std::endl;
1039         NonlinearProblem nlp(model,u,t+dt,dt);
1040         bool reduced = false;
1041         error = false;
1042         while (1)
1043         {
1044             unew = u;
1045             newton.solve(nlp,unew);
1046             if (newton.has_converged())
1047             {
1048                 u = unew;
1049                 t += dt;
1050                 if (!reduced && dt<dtmax-1e-13)

```

```

1065         {
1066             dt = std::min(2.0*dt,dtmax);
1067             if (verbosity>0)
1068                 std::cout << "IE: increasing time step to " << dt << std::endl;
1069         }
1070         return;
1071     }
1072     else
1073     {
1074         if (dt<1e-12)
1075         {
1076             HDNUM_ERROR("time step too small in implicit Euler");
1077             error = true;
1078             break;
1079         }
1080         dt *= 0.5;
1081         reduced = true;
1082         nlp.set_tnew_dt(t+dt,dt);
1083         if (verbosity>0) std::cout << "IE: reducing time step to " << dt << std::endl;
1084     }
1085 }
1086 }
1087
1088 bool get_error () const
1089 {
1090     return error;
1091 }
1092
1093 void set_state (time_type t_, const Vector<number_type>& u_)
1094 {
1095     t = t_;
1096     u = u_;
1097 }
1098
1099 const Vector<number_type>& get_state () const
1100 {
1101     return u;
1102 }
1103
1104 time_type get_time () const
1105 {
1106     return t;
1107 }
1108
1109 time_type get_dt () const
1110 {
1111     return dt;
1112 }
1113
1114 size_type get_order () const
1115 {
1116     return 1;
1117 }
1118
1119 void get_info () const
1120 {
1121 }
1122
1123 private:
1124     size_type verbosity;
1125     const M& model;
1126     const S& newton;
1127     time_type t, dt, dtmax;
1128     number_type reduction;
1129     size_type linesearchsteps;
1130     Vector<number_type> u;
1131     Vector<number_type> unew;
1132     mutable bool error;
1133 };
1134
1135 template<class M, class S>
1136 class DIRK
1137 {
1138 public:
1139     typedef typename M::size_type size_type;
1140
1141     typedef typename M::time_type time_type;
1142
1143     typedef typename M::number_type number_type;
1144
1145     typedef DenseMatrix<number_type> ButcherTableau;
1146
1147 private:
1148     static ButcherTableau initTableau(const std::string method)
1149     {

```

```

1175     if(method.find("Implicit Euler") != std::string::npos){
1176         ButcherTableau butcher(2,2,0.0);
1177         butcher[1][1] = 1;
1178         butcher[0][1] = 1;
1179         butcher[0][0] = 1;
1180
1181         return butcher;
1182     }
1183     else if(method.find("Alexander") != std::string::npos){
1184         ButcherTableau butcher(3,3,0.0);
1185         const number_type alpha = 1. - sqrt(2.)/2.;
1186         butcher[0][0] = alpha;
1187         butcher[0][1] = alpha;
1188
1189         butcher[1][0] = 1.;
1190         butcher[1][1] = 1. - alpha;
1191         butcher[1][2] = alpha;
1192
1193         butcher[2][1] = 1. - alpha;
1194         butcher[2][2] = alpha;
1195
1196         return butcher;
1197     }
1198     else if(method.find("Crouzieux") != std::string::npos){
1199         ButcherTableau butcher(3,3,0.0);
1200         const number_type beta = 1./2./sqrt(3);
1201         butcher[0][0] = 0.5 + beta;
1202         butcher[0][1] = 0.5 + beta;
1203
1204         butcher[1][0] = 0.5 - beta;
1205         butcher[1][1] = -1. / sqrt(3);
1206         butcher[1][2] = 0.5 + beta;
1207
1208         butcher[2][1] = 0.5;
1209         butcher[2][2] = 0.5;
1210
1211         return butcher;
1212     }
1213     else if(method.find("Midpoint Rule") != std::string::npos){
1214         ButcherTableau butcher(2,2,0.0);
1215         butcher[0][0] = 0.5;
1216         butcher[0][1] = 0.5;
1217         butcher[1][1] = 1;
1218
1219         return butcher;
1220     }
1221     else if(method.find("Fractional Step Theta") != std::string::npos){
1222         ButcherTableau butcher(5,5,0.0);
1223         const number_type theta = 1 - sqrt(2.)/2.;
1224         const number_type alpha = 2. - sqrt(2.);
1225         const number_type beta = 1. - alpha;
1226         butcher[1][0] = theta;
1227         butcher[1][1] = beta * theta;
1228         butcher[1][2] = alpha * theta;
1229
1230         butcher[2][0] = 1.-theta;
1231         butcher[2][1] = beta * theta;
1232         butcher[2][2] = alpha * (1.-theta);
1233         butcher[2][3] = alpha * theta;
1234
1235         butcher[3][0] = 1.;
1236         butcher[3][1] = beta * theta;
1237         butcher[3][2] = alpha * (1.-theta);
1238         butcher[3][3] = (alpha + beta) * theta;
1239         butcher[3][4] = alpha * theta;
1240
1241         butcher[4][1] = beta * theta;
1242         butcher[4][2] = alpha * (1.-theta);
1243         butcher[4][3] = (alpha + beta) * theta;
1244         butcher[4][4] = alpha * theta;
1245
1246         return butcher;
1247     }
1248     else{
1249         HDNUM_ERROR("Order not available for Runge Kutta solver.");
1250     }
1251 }
1252
1253 static int initOrder(const std::string method)
1254 {
1255     if(method.find("Implicit Euler") != std::string::npos){
1256         return 1;
1257     }
1258     else if(method.find("Alexander") != std::string::npos){
1259         return 2;
1260     }
1261     else if(method.find("Crouzieux") != std::string::npos){

```

```

1262         return 3;
1263     }
1264     else if(method.find("Midpoint Rule") != std::string::npos){
1265         return 2;
1266     }
1267     else if(method.find("Fractional Step Theta") != std::string::npos){
1268         return 2;
1269     }
1270     else{
1271         HDNUM_ERROR("Order not available for Runge Kutta solver.");
1272     }
1273 }
1274
1275
1276 // h_n f(t_n, y_n) - y_n + y_{n-1} = 0
1277 class NonlinearProblem
1278 {
1279 public:
1280     typedef typename M::size_type size_type;
1281
1282     typedef typename M::number_type number_type;
1283
1284     NonlinearProblem (const M& model_, const Vector<number_type>& yold_,
1285                     typename M::time_type told_, typename M::time_type dt_,
1286                     const ButcherTableau & butcher_, const int rk_step_,
1287                     const std::vector< Vector<number_type> > & k_)
1288     : model(model_), yold(yold_), told(told_),
1289       dt(dt_), butcher(butcher_), rk_step(rk_step_), k_old(model.size(),0)
1290     {
1291         for(int i=0; i<rk_step; ++i)
1292             k_old.update(butcher[rk_step][1+i] * dt, k_[i]);
1293     }
1294
1295     std::size_t size () const
1296     {
1297         return model.size();
1298     }
1299
1300     void F (const Vector<number_type>& x, Vector<number_type>& result) const
1301     {
1302         result = k_old;
1303
1304         Vector<number_type> current_z(x);
1305         current_z.update(1.,yold);
1306
1307         const number_type tnew = told + butcher[rk_step][0] * dt;
1308
1309         Vector<number_type> current_k(model.size(),0.);
1310         model.f(tnew,current_z,current_k);
1311         result.update(butcher[rk_step][rk_step+1] * dt, current_k);
1312
1313         result.update(-1.,x);
1314     }
1315
1316     void F_x (const Vector<number_type>& x, DenseMatrix<number_type>& result) const
1317     {
1318         const number_type tnew = told + butcher[rk_step][0] * dt;
1319
1320         Vector<number_type> current_z(x);
1321         current_z.update(1.,yold);
1322
1323         model.f_x(tnew,current_z,result);
1324
1325         result *= dt * butcher[rk_step][rk_step+1];
1326
1327         for (size_type i=0; i<model.size(); i++) result[i][i] -= number_type(1.0);
1328     }
1329
1330     void set_told_dt (typename M::time_type told_, typename M::time_type dt_)
1331     {
1332         told = told_;
1333         dt = dt_;
1334     }
1335
1336 private:
1337     const M& model;
1338     const Vector<number_type>& yold;
1339     typename M::time_type told;
1340     typename M::time_type dt;
1341     const ButcherTableau & butcher;
1342     const int rk_step;
1343     Vector<number_type> k_old;
1344 };
1345
1346 public:
1347     DIRK (const M& model_, const S& newton_, const ButcherTableau & butcher_, const int order_)

```

```

1358         : verbosity(0), butcher(butcher_), model(model_), newton(newton_),
1359         u(model.size()), order(order_)
1360     {
1361         model.initialize(t,u);
1362         dt = dtmax = 0.1;
1363     }
1364
1365     DIRK (const M& model_, const S& newton_, const std::string method)
1366         : verbosity(0), butcher(initTableau(method)), model(model_), newton(newton_), u(model.size()),
1367         order(initOrder(method))
1368     {
1369         model.initialize(t,u);
1370         dt = dtmax = 0.1;
1371     }
1372
1373 void set_dt (time_type dt_)
1374 {
1375     dt = dtmax = dt_;
1376 }
1377
1378 void set_verbosity (size_type verbosity_)
1379 {
1380     verbosity = verbosity_;
1381 }
1382
1383 void step ()
1384 {
1385     const size_type R = butcher.colsize()-1;
1386
1387     bool reduced = false;
1388     error = false;
1389     if(verbosity>=2)
1390         std::cout << "DIRK: step to" << " t+dt=" << t+dt << " dt=" << dt << std::endl;
1391
1392     while (1)
1393     {
1394         bool converged = true;
1395
1396         // Perform R Runge-Kutta steps
1397         std::vector< Vector<number_type> > k;
1398         for(size_type i=0; i<R; ++i) {
1399             if (verbosity>=2)
1400                 std::cout << "DIRK: step nr " << i << std::endl;
1401
1402             Vector<number_type> current_z(model.size(),0.0);
1403
1404             // Set starting value of k_i
1405             // model.f(t,u,current_k);
1406
1407             // Solve nonlinear problem
1408             NonlinearProblem nlp(model,u,t,dt,butcher,i,k);
1409
1410             newton.solve(nlp,current_z);
1411
1412             converged = converged && newton.has_converged();
1413             if(!converged)
1414                 break;
1415
1416             current_z.update(1., u);
1417             const number_type t_i = t + butcher[i][0] * dt;
1418             Vector<number_type> current_k(model.size(),0.);
1419             model.f(t_i,current_z,current_k);
1420
1421             k.push_back( current_k );
1422         }
1423
1424         if (converged)
1425         {
1426             if(verbosity >= 2)
1427                 std::cout << "DIRK finished" << std::endl;
1428
1429             // Update to new solution
1430             for(size_type i=0; i<R; ++i)
1431                 u.update(dt*butcher[R][1+i],k[i]);
1432
1433             t += dt;
1434             if (!reduced && dt<dtmax-1e-13)
1435             {
1436                 dt = std::min(2.0*dt,dtmax);
1437                 if (verbosity>0)
1438                     std::cout << "DIRK: increasing time step to " << dt << std::endl;
1439             }
1440             return;
1441         }
1442         else

```

```

1450         {
1451             if (dt<1e-12)
1452             {
1453                 HDNUM_ERROR("time step too small in implicit Euler");
1454                 error = true;
1455                 break;
1456             }
1457             dt *= 0.5;
1458             reduced = true;
1459             if (verbosity>0) std::cout << "DIRK: reducing time step to " << dt << std::endl;
1460         }
1461     }
1462 }
1463
1464 bool get_error () const
1465 {
1466     return error;
1467 }
1468
1469 void set_state (time_type t_, const Vector<number_type>& u_)
1470 {
1471     t = t_;
1472     u = u_;
1473 }
1474
1475 const Vector<number_type>& get_state () const
1476 {
1477     return u;
1478 }
1479
1480 time_type get_time () const
1481 {
1482     return t;
1483 }
1484
1485 time_type get_dt () const
1486 {
1487     return dt;
1488 }
1489
1490 size_type get_order () const
1491 {
1492     return order;
1493 }
1494
1495 void get_info () const
1496 {
1497 }
1498
1499 private:
1500     size_type verbosity;
1501     const DenseMatrix<number_type> butcher;
1502     const M& model;
1503     const S& newton;
1504     time_type t, dt, dtmax;
1505     number_type reduction;
1506     size_type linesearchsteps;
1507     Vector<number_type> u;
1508     int order;
1509     mutable bool error;
1510 };
1511
1512 template<class T, class N>
1513 inline void gnuplot (const std::string& fname, const std::vector<T> t, const std::vector<Vector<N> >
1514 u)
1515 {
1516     std::fstream f(fname.c_str(),std::ios::out);
1517     for (typename std::vector<T>::size_type n=0; n<t.size(); n++)
1518     {
1519         f << std::scientific << std::showpoint
1520           << std::setprecision(16) << t[n];
1521         for (typename Vector<N>::size_type i=0; i<u[n].size(); i++)
1522             f << " " << std::scientific << std::showpoint
1523               << std::setprecision(u[n].precision()) << u[n][i];
1524         f << std::endl;
1525     }
1526     f.close();
1527 }
1528
1529 template<class T, class N>
1530 inline void gnuplot (const std::string& fname, const std::vector<T> t, const std::vector<Vector<N> >
1531 u, const std::vector<T> dt)
1532 {
1533     std::fstream f(fname.c_str(),std::ios::out);
1534     for (typename std::vector<T>::size_type n=0; n<t.size(); n++)
1535     {
1536         f << std::scientific << std::showpoint

```



```

1544         « std::setprecision(16) « t[n];
1545     for (typename Vector<N>::size_type i=0; i<u[n].size(); i++)
1546         f « " " « std::scientific « std::showpoint
1547         « std::setprecision(u[n].precision()) « u[n][i];
1548     f « " " « std::scientific « std::showpoint
1549     « std::setprecision(16) « dt[n];
1550     f « std::endl;
1551 }
1552     f.close();
1553 }
1554
1555 } // namespace hdnum
1556
1557 #endif

```

## 5.10 src/opcounter.hh File Reference

This file implements an operator counting class.

```

#include <type_traits>
#include <iostream>
#include <cmath>
#include <cstdlib>

```

### Classes

- class [hdnum::oc::OpCounter< F >](#)
- struct [hdnum::oc::OpCounter< F >::Counters](#)  
*Struct storing the number of operations.*

### Functions

- template<typename F >  
**OpCounter< F > hdnum::oc::operator-** (const OpCounter< F > &a)
- template<typename F >  
**OpCounter< F > hdnum::oc::operator+** (const OpCounter< F > &a, const OpCounter< F > &b)
- template<typename F >  
**OpCounter< F > hdnum::oc::operator+** (const OpCounter< F > &a, const F &b)
- template<typename F >  
**OpCounter< F > hdnum::oc::operator+** (const F &a, const OpCounter< F > &b)
- template<typename F, typename T >  
**std::enable\_if< std::is\_arithmetic< T >::value, OpCounter< F > >::type hdnum::oc::operator+** (const OpCounter< F > &a, const T &b)
- template<typename F, typename T >  
**std::enable\_if< std::is\_arithmetic< T >::value, OpCounter< F > >::type hdnum::oc::operator+** (const T &a, const OpCounter< F > &b)
- template<typename F >  
**OpCounter< F > & hdnum::oc::operator+=** (OpCounter< F > &a, const OpCounter< F > &b)
- template<typename F >  
**OpCounter< F > & hdnum::oc::operator+=** (OpCounter< F > &a, const F &b)
- template<typename F, typename T >  
**std::enable\_if< std::is\_arithmetic< T >::value, OpCounter< F > & >::type hdnum::oc::operator+=** (OpCounter< F > &a, const T &b)
- template<typename F >  
**OpCounter< F > hdnum::oc::operator-** (const OpCounter< F > &a, const OpCounter< F > &b)



- [illegible]

- `template<typename F, typename T>`  
`bool hdnum::oc::operator== (const T &a, const OpCounter< F > &b)`
- `template<typename F>`  
`OpCounter< F > hdnum::oc::exp (const OpCounter< F > &a)`
- `template<typename F>`  
`OpCounter< F > hdnum::oc::pow (const OpCounter< F > &a, const OpCounter< F > &b)`
- `template<typename F>`  
`OpCounter< F > hdnum::oc::pow (const OpCounter< F > &a, const F &b)`
- `template<typename F, typename T>`  
`OpCounter< F > hdnum::oc::pow (const OpCounter< F > &a, const T &b)`
- `template<typename F>`  
`OpCounter< F > hdnum::oc::pow (const F &a, const OpCounter< F > &b)`
- `template<typename F, typename T>`  
`OpCounter< F > hdnum::oc::pow (const T &a, const OpCounter< F > &b)`
- `template<typename F>`  
`OpCounter< F > hdnum::oc::sin (const OpCounter< F > &a)`
- `template<typename F>`  
`OpCounter< F > hdnum::oc::cos (const OpCounter< F > &a)`
- `template<typename F>`  
`OpCounter< F > hdnum::oc::sqrt (const OpCounter< F > &a)`
- `template<typename F>`  
`OpCounter< F > hdnum::oc::abs (const OpCounter< F > &a)`

### 5.10.1 Detailed Description

This file implements an operator counting class.

## 5.11 opcounter.hh

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

```
1 // -*- tab-width: 4; indent-tabs-mode: nil -*-
2 #ifndef __OPCOUNTER__
3 #define __OPCOUNTER__
4
5 #include <type_traits>
6 #include <iostream>
7 #include <cmath>
8 #include <cstdlib>
9
10 namespace hdnum {
11     namespace oc {
12         template<typename F>
13         class OpCounter;
14     }
15 }
16
17 namespace hdnum {
18     namespace oc {
19         template<typename F>
20         class OpCounter
21         {
22         public:
23             using size_type = std::size_t;
24             using value_type = F;
25
26             OpCounter()
27             : _v()
28             {}
29
30             template<typename T>
31             OpCounter(const T& t, typename std::enable_if<std::is_same<T,int>::value and
32 !std::is_same<F,int>::value>::type* = nullptr)
```

```

44         : _v(t)
45     {}
46
47     OpCounter(const F& f)
48         : _v(f)
49     {}
50
51     OpCounter(F&& f)
52         : _v(f)
53     {}
54
55     explicit OpCounter(const char* s)
56         : _v(strtod(s,nullptr))
57     {}
58
59     OpCounter& operator=(const char* s)
60     {
61         _v = strtod(s,nullptr);
62         return *this;
63     }
64
65     explicit operator F() const
66     {
67         return _v;
68     }
69
70     OpCounter& operator=(const F& f)
71     {
72         _v = f;
73         return *this;
74     }
75
76     OpCounter& operator=(F&& f)
77     {
78         _v = f;
79         return *this;
80     }
81
82     friend std::ostream& operator<<(std::ostream& os, const OpCounter& f)
83     {
84         os << "OC(" << f._v << ")";
85         return os;
86     }
87
88     friend std::istream& operator>>(std::istream& iss, OpCounter& f)
89     {
90         iss >> f._v;
91         return iss;
92     }
93
94     F* data()
95     {
96         return &_v;
97     }
98
99     const F* data() const
100    {
101        return &_v;
102    }
103
104    F _v;
105
106    struct Counters {
107
108        size_type addition_count;
109        size_type multiplication_count;
110        size_type division_count;
111        size_type exp_count;
112        size_type pow_count;
113        size_type sin_count;
114        size_type sqrt_count;
115        size_type comparison_count;
116
117        Counters()
118            : addition_count(0)
119            , multiplication_count(0)
120            , division_count(0)
121            , exp_count(0)
122            , pow_count(0)
123            , sin_count(0)
124            , sqrt_count(0)
125            , comparison_count(0)
126        {}
127
128        void reset()
129        {
130            addition_count = 0;

```

```

132     multiplication_count = 0;
133     division_count = 0;
134     exp_count = 0;
135     pow_count = 0;
136     sin_count = 0;
137     sqrt_count = 0;
138     comparison_count = 0;
139 }
140
141
142 template<typename Stream>
143 void reportOperations(Stream& os, bool doReset = false)
144 {
145     os << "additions: " << addition_count << std::endl
146     << "multiplications: " << multiplication_count << std::endl
147     << "divisions: " << division_count << std::endl
148     << "exp: " << exp_count << std::endl
149     << "pow: " << pow_count << std::endl
150     << "sin: " << sin_count << std::endl
151     << "sqrt: " << sqrt_count << std::endl
152     << "comparisons: " << comparison_count << std::endl
153     << std::endl
154     << "total: " << addition_count + multiplication_count + division_count + exp_count +
pow_count + sin_count + sqrt_count + comparison_count << std::endl;
155
156     if (doReset)
157         reset();
158 }
159
160 size_type totalOperationCount(bool doReset=false)
161 {
162     if (doReset)
163         reset();
164
165     return addition_count + multiplication_count + division_count + exp_count + pow_count +
sin_count + sqrt_count + comparison_count;
166 }
167
168
169 Counters& operator+=(const Counters& rhs)
170 {
171     addition_count += rhs.addition_count;
172     multiplication_count += rhs.multiplication_count;
173     division_count += rhs.division_count;
174     exp_count += rhs.exp_count;
175     pow_count += rhs.pow_count;
176     sin_count += rhs.sin_count;
177     sqrt_count += rhs.sqrt_count;
178     comparison_count += rhs.comparison_count;
179     return *this;
180 }
181
182 Counters operator-(const Counters& rhs)
183 {
184     Counters r;
185     r.addition_count = addition_count - rhs.addition_count;
186     r.multiplication_count = multiplication_count - rhs.multiplication_count;
187     r.division_count = division_count - rhs.division_count;
188     r.exp_count = exp_count - rhs.exp_count;
189     r.pow_count = pow_count - rhs.pow_count;
190     r.sin_count = sin_count - rhs.sin_count;
191     r.sqrt_count = sqrt_count - rhs.sqrt_count;
192     r.comparison_count = comparison_count - rhs.comparison_count;
193     return r;
194 }
195
196 };
197
198 static void additions(std::size_t n)
199 {
200     counters.addition_count += n;
201 }
202
203 static void multiplications(std::size_t n)
204 {
205     counters.multiplication_count += n;
206 }
207
208 static void divisions(std::size_t n)
209 {
210     counters.division_count += n;
211 }
212
213 static void reset()
214 {
215     counters.reset();
216 }
217
218 template<typename Stream>

```

```

220     static void reportOperations(Stream& os, bool doReset = false)
221     {
222         counters.reportOperations(os,doReset);
223     }
224
225     static size_type totalOperationCount (bool doReset=false)
226     {
227         return counters.totalOperationCount (doReset);
228     }
229
230     static Counters counters;
231
232 };
233
234 template<typename F>
235 typename OpCounter<F>::Counters OpCounter<F>::counters;
236
237 // *****
238 // negation
239 // *****
240
241 template<typename F>
242 OpCounter<F> operator-(const OpCounter<F>& a)
243 {
244     ++OpCounter<F>::counters.addition_count;
245     return {-a._v};
246 }
247
248 // *****
249 // addition
250 // *****
251
252 template<typename F>
253 OpCounter<F> operator+(const OpCounter<F>& a, const OpCounter<F>& b)
254 {
255     ++OpCounter<F>::counters.addition_count;
256     return {a._v + b._v};
257 }
258
259 template<typename F>
260 OpCounter<F> operator+(const OpCounter<F>& a, const F& b)
261 {
262     ++OpCounter<F>::counters.addition_count;
263     return {a._v + b};
264 }
265
266 template<typename F>
267 OpCounter<F> operator+(const F& a, const OpCounter<F>& b)
268 {
269     ++OpCounter<F>::counters.addition_count;
270     return {a + b._v};
271 }
272
273 template<typename F, typename T>
274 typename std::enable_if<
275     std::is_arithmetic<T>::value,
276     OpCounter<F>
277 >::type
278 operator+(const OpCounter<F>& a, const T& b)
279 {
280     ++OpCounter<F>::counters.addition_count;
281     return {a._v + b};
282 }
283
284 template<typename F, typename T>
285 typename std::enable_if<
286     std::is_arithmetic<T>::value,
287     OpCounter<F>
288 >::type
289 operator+(const T& a, const OpCounter<F>& b)
290 {
291     ++OpCounter<F>::counters.addition_count;
292     return {a + b._v};
293 }
294
295 template<typename F>
296 OpCounter<F>& operator+=(OpCounter<F>& a, const OpCounter<F>& b)
297 {
298     ++OpCounter<F>::counters.addition_count;
299     a._v += b._v;
300     return a;
301 }
302
303 template<typename F>
304 OpCounter<F>& operator+=(OpCounter<F>& a, const F& b)
305 {
306     ++OpCounter<F>::counters.addition_count;
307     a._v += b;
308     return a;
309 }

```

```

308     ++OpCounter<F>::counters.addition_count;
309     a._v += b;
310     return a;
311 }
312
313 template<typename F, typename T>
314 typename std::enable_if<
315     std::is_arithmetic<T>::value,
316     OpCounter<F>&
317 >::type
318 operator+=(OpCounter<F>& a, const T& b)
319 {
320     ++OpCounter<F>::counters.addition_count;
321     a._v += b;
322     return a;
323 }
324
325 // *****
326 // subtraction
327 // *****
328
329 template<typename F>
330 OpCounter<F> operator-(const OpCounter<F>& a, const OpCounter<F>& b)
331 {
332     ++OpCounter<F>::counters.addition_count;
333     return {a._v - b._v};
334 }
335
336 template<typename F>
337 OpCounter<F> operator-(const OpCounter<F>& a, const F& b)
338 {
339     ++OpCounter<F>::counters.addition_count;
340     return {a._v - b};
341 }
342
343 template<typename F>
344 OpCounter<F> operator-(const F& a, const OpCounter<F>& b)
345 {
346     ++OpCounter<F>::counters.addition_count;
347     return {a - b._v};
348 }
349
350 template<typename F, typename T>
351 typename std::enable_if<
352     std::is_arithmetic<T>::value,
353     OpCounter<F>
354 >::type
355 operator-(const OpCounter<F>& a, const T& b)
356 {
357     ++OpCounter<F>::counters.addition_count;
358     return {a._v - b};
359 }
360
361 template<typename F, typename T>
362 typename std::enable_if<
363     std::is_arithmetic<T>::value,
364     OpCounter<F>
365 >::type
366 operator-(const T& a, const OpCounter<F>& b)
367 {
368     ++OpCounter<F>::counters.addition_count;
369     return {a - b._v};
370 }
371
372 template<typename F>
373 OpCounter<F>& operator-=(OpCounter<F>& a, const OpCounter<F>& b)
374 {
375     ++OpCounter<F>::counters.addition_count;
376     a._v -= b._v;
377     return a;
378 }
379
380 template<typename F>
381 OpCounter<F>& operator-=(OpCounter<F>& a, const F& b)
382 {
383     ++OpCounter<F>::counters.addition_count;
384     a._v -= b;
385     return a;
386 }
387
388 template<typename F, typename T>
389 typename std::enable_if<
390     std::is_arithmetic<T>::value,
391     OpCounter<F>&
392 >::type
393 operator-=(OpCounter<F>& a, const T& b)
394 {

```



```

395     ++OpCounter<F>::counters.addition_count;
396     a._v -= b;
397     return a;
398 }
399
400
401 // *****
402 // multiplication
403 // *****
404
405 template<typename F>
406 OpCounter<F> operator*(const OpCounter<F>& a, const OpCounter<F>& b)
407 {
408     ++OpCounter<F>::counters.multiplication_count;
409     return {a._v * b._v};
410 }
411
412 template<typename F>
413 OpCounter<F> operator*(const OpCounter<F>& a, const F& b)
414 {
415     ++OpCounter<F>::counters.multiplication_count;
416     return {a._v * b};
417 }
418
419 template<typename F>
420 OpCounter<F> operator*(const F& a, const OpCounter<F>& b)
421 {
422     ++OpCounter<F>::counters.multiplication_count;
423     return {a * b._v};
424 }
425
426 template<typename F, typename T>
427 typename std::enable_if<
428     std::is_arithmetic<T>::value,
429     OpCounter<F>
430 >::type
431 operator*(const OpCounter<F>& a, const T& b)
432 {
433     ++OpCounter<F>::counters.multiplication_count;
434     return {a._v * b};
435 }
436
437 template<typename F, typename T>
438 typename std::enable_if<
439     std::is_arithmetic<T>::value,
440     OpCounter<F>
441 >::type
442 operator*(const T& a, const OpCounter<F>& b)
443 {
444     ++OpCounter<F>::counters.multiplication_count;
445     return {a * b._v};
446 }
447
448 template<typename F>
449 OpCounter<F>& operator*=(OpCounter<F>& a, const OpCounter<F>& b)
450 {
451     ++OpCounter<F>::counters.multiplication_count;
452     a._v *= b._v;
453     return a;
454 }
455
456 template<typename F>
457 OpCounter<F>& operator*=(OpCounter<F>& a, const F& b)
458 {
459     ++OpCounter<F>::counters.multiplication_count;
460     a._v *= b;
461     return a;
462 }
463
464 template<typename F, typename T>
465 typename std::enable_if<
466     std::is_arithmetic<T>::value,
467     OpCounter<F>&
468 >::type
469 operator*=(OpCounter<F>& a, const T& b)
470 {
471     ++OpCounter<F>::counters.multiplication_count;
472     a._v *= b;
473     return a;
474 }
475
476
477 // *****
478 // division
479 // *****
480
481 template<typename F>

```

```

482 OpCounter<F> operator/(const OpCounter<F>& a, const OpCounter<F>& b)
483 {
484     ++OpCounter<F>::counters.division_count;
485     return {a._v / b._v};
486 }
487
488 template<typename F>
489 OpCounter<F> operator/(const OpCounter<F>& a, const F& b)
490 {
491     ++OpCounter<F>::counters.division_count;
492     return {a._v / b};
493 }
494
495 template<typename F>
496 OpCounter<F> operator/(const F& a, const OpCounter<F>& b)
497 {
498     ++OpCounter<F>::counters.division_count;
499     return {a / b._v};
500 }
501
502 template<typename F, typename T>
503 typename std::enable_if<
504     std::is_arithmetic<T>::value,
505     OpCounter<F>
506 >::type
507 operator/(const OpCounter<F>& a, const T& b)
508 {
509     ++OpCounter<F>::counters.division_count;
510     return {a._v / b};
511 }
512
513 template<typename F, typename T>
514 typename std::enable_if<
515     std::is_arithmetic<T>::value,
516     OpCounter<F>
517 >::type
518 operator/(const T& a, const OpCounter<F>& b)
519 {
520     ++OpCounter<F>::counters.division_count;
521     return {a / b._v};
522 }
523
524 template<typename F>
525 OpCounter<F>& operator/=(OpCounter<F>& a, const OpCounter<F>& b)
526 {
527     ++OpCounter<F>::counters.division_count;
528     a._v /= b._v;
529     return a;
530 }
531
532 template<typename F>
533 OpCounter<F>& operator/=(OpCounter<F>& a, const F& b)
534 {
535     ++OpCounter<F>::counters.division_count;
536     a._v /= b;
537     return a;
538 }
539
540 template<typename F, typename T>
541 typename std::enable_if<
542     std::is_arithmetic<T>::value,
543     OpCounter<F>&
544 >::type
545 operator/=(OpCounter<F>& a, const T& b)
546 {
547     ++OpCounter<F>::counters.division_count;
548     a._v /= b;
549     return a;
550 }
551
552
553
554 // *****
555 // comparisons
556 // *****
557
558
559 // *****
560 // less
561 // *****
562
563 template<typename F>
564 bool operator<(const OpCounter<F>& a, const OpCounter<F>& b)
565 {
566     ++OpCounter<F>::counters.comparison_count;
567     return {a._v < b._v};
568 }

```

```

569
570     template<typename F>
571     bool operator<(const OpCounter<F>& a, const F& b)
572     {
573         ++OpCounter<F>::counters.comparison_count;
574         return {a._v < b};
575     }
576
577     template<typename F>
578     bool operator<(const F& a, const OpCounter<F>& b)
579     {
580         ++OpCounter<F>::counters.comparison_count;
581         return {a < b._v};
582     }
583
584     template<typename F, typename T>
585     bool operator<(const OpCounter<F>& a, const T& b)
586     {
587         ++OpCounter<F>::counters.comparison_count;
588         return {a._v < b};
589     }
590
591     template<typename F, typename T>
592     bool operator<(const T& a, const OpCounter<F>& b)
593     {
594         ++OpCounter<F>::counters.comparison_count;
595         return {a < b._v};
596     }
597
598
599     // *****
600     // less_or_equals
601     // *****
602
603     template<typename F>
604     bool operator<=(const OpCounter<F>& a, const OpCounter<F>& b)
605     {
606         ++OpCounter<F>::counters.comparison_count;
607         return {a._v <= b._v};
608     }
609
610     template<typename F>
611     bool operator<=(const OpCounter<F>& a, const F& b)
612     {
613         ++OpCounter<F>::counters.comparison_count;
614         return {a._v <= b};
615     }
616
617     template<typename F>
618     bool operator<=(const F& a, const OpCounter<F>& b)
619     {
620         ++OpCounter<F>::counters.comparison_count;
621         return {a <= b._v};
622     }
623
624     template<typename F, typename T>
625     bool operator<=(const OpCounter<F>& a, const T& b)
626     {
627         ++OpCounter<F>::counters.comparison_count;
628         return {a._v <= b};
629     }
630
631     template<typename F, typename T>
632     bool operator<=(const T& a, const OpCounter<F>& b)
633     {
634         ++OpCounter<F>::counters.comparison_count;
635         return {a <= b._v};
636     }
637
638
639     // *****
640     // greater
641     // *****
642
643     template<typename F>
644     bool operator>(const OpCounter<F>& a, const OpCounter<F>& b)
645     {
646         ++OpCounter<F>::counters.comparison_count;
647         return {a._v > b._v};
648     }
649
650     template<typename F>
651     bool operator>(const OpCounter<F>& a, const F& b)
652     {
653         ++OpCounter<F>::counters.comparison_count;
654         return {a._v > b};
655     }

```

```

656
657     template<typename F>
658     bool operator>(const F& a, const OpCounter<F>& b)
659     {
660         ++OpCounter<F>::counters.comparison_count;
661         return {a > b._v};
662     }
663
664     template<typename F, typename T>
665     bool operator>(const OpCounter<F>& a, const T& b)
666     {
667         ++OpCounter<F>::counters.comparison_count;
668         return {a._v > b};
669     }
670
671     template<typename F, typename T>
672     bool operator>(const T& a, const OpCounter<F>& b)
673     {
674         ++OpCounter<F>::counters.comparison_count;
675         return {a > b._v};
676     }
677
678
679     // *****
680     // greater_or_equals
681     // *****
682
683     template<typename F>
684     bool operator>=(const OpCounter<F>& a, const OpCounter<F>& b)
685     {
686         ++OpCounter<F>::counters.comparison_count;
687         return {a._v >= b._v};
688     }
689
690     template<typename F>
691     bool operator>=(const OpCounter<F>& a, const F& b)
692     {
693         ++OpCounter<F>::counters.comparison_count;
694         return {a._v >= b};
695     }
696
697     template<typename F>
698     bool operator>=(const F& a, const OpCounter<F>& b)
699     {
700         ++OpCounter<F>::counters.comparison_count;
701         return {a >= b._v};
702     }
703
704     template<typename F, typename T>
705     bool operator>=(const OpCounter<F>& a, const T& b)
706     {
707         ++OpCounter<F>::counters.comparison_count;
708         return {a._v >= b};
709     }
710
711     template<typename F, typename T>
712     bool operator>=(const T& a, const OpCounter<F>& b)
713     {
714         ++OpCounter<F>::counters.comparison_count;
715         return {a >= b._v};
716     }
717
718
719     // *****
720     // inequality
721     // *****
722
723     template<typename F>
724     bool operator!=(const OpCounter<F>& a, const OpCounter<F>& b)
725     {
726         ++OpCounter<F>::counters.comparison_count;
727         return {a._v != b._v};
728     }
729
730     template<typename F>
731     bool operator!=(const OpCounter<F>& a, const F& b)
732     {
733         ++OpCounter<F>::counters.comparison_count;
734         return {a._v != b};
735     }
736
737     template<typename F>
738     bool operator!=(const F& a, const OpCounter<F>& b)
739     {
740         ++OpCounter<F>::counters.comparison_count;
741         return {a != b._v};
742     }

```

```

743
744     template<typename F, typename T>
745     bool operator!=(const OpCounter<F>& a, const T& b)
746     {
747         ++OpCounter<F>::counters.comparison_count;
748         return {a._v != b};
749     }
750
751     template<typename F, typename T>
752     bool operator!=(const T& a, const OpCounter<F>& b)
753     {
754         ++OpCounter<F>::counters.comparison_count;
755         return {a != b._v};
756     }
757
758
759     // *****
760     // equals
761     // *****
762
763     template<typename F>
764     bool operator==(const OpCounter<F>& a, const OpCounter<F>& b)
765     {
766         ++OpCounter<F>::counters.comparison_count;
767         return {a._v == b._v};
768     }
769
770     template<typename F>
771     bool operator==(const OpCounter<F>& a, const F& b)
772     {
773         ++OpCounter<F>::counters.comparison_count;
774         return {a._v == b};
775     }
776
777     template<typename F>
778     bool operator==(const F& a, const OpCounter<F>& b)
779     {
780         ++OpCounter<F>::counters.comparison_count;
781         return {a == b._v};
782     }
783
784     template<typename F, typename T>
785     bool operator==(const OpCounter<F>& a, const T& b)
786     {
787         ++OpCounter<F>::counters.comparison_count;
788         return {a._v == b};
789     }
790
791     template<typename F, typename T>
792     bool operator==(const T& a, const OpCounter<F>& b)
793     {
794         ++OpCounter<F>::counters.comparison_count;
795         return {a == b._v};
796     }
797
798
799
800     // *****
801     // functions
802     // *****
803
804     template<typename F>
805     OpCounter<F> exp(const OpCounter<F>& a)
806     {
807         ++OpCounter<F>::counters.exp_count;
808         return {std::exp(a._v)};
809     }
810
811     template<typename F>
812     OpCounter<F> pow(const OpCounter<F>& a, const OpCounter<F>& b)
813     {
814         ++OpCounter<F>::counters.pow_count;
815         return {std::pow(a._v, b._v)};
816     }
817
818     template<typename F>
819     OpCounter<F> pow(const OpCounter<F>& a, const F& b)
820     {
821         ++OpCounter<F>::counters.pow_count;
822         return {std::pow(a._v, b)};
823     }
824
825     template<typename F, typename T>
826     OpCounter<F> pow(const OpCounter<F>& a, const T& b)
827     {
828         ++OpCounter<F>::counters.pow_count;
829         return {std::pow(a._v, b)};

```

```

830     }
831
832     template<typename F>
833     OpCounter<F> pow(const F& a, const OpCounter<F>& b)
834     {
835         ++OpCounter<F>::counters.pow_count;
836         return {std::pow(a,b._v)};
837     }
838
839     template<typename F, typename T>
840     OpCounter<F> pow(const T& a, const OpCounter<F>& b)
841     {
842         ++OpCounter<F>::counters.pow_count;
843         return {std::pow(a,b._v)};
844     }
845
846     template<typename F>
847     OpCounter<F> sin(const OpCounter<F>& a)
848     {
849         ++OpCounter<F>::counters.sin_count;
850         return {std::sin(a._v)};
851     }
852
853     template<typename F>
854     OpCounter<F> cos(const OpCounter<F>& a)
855     {
856         ++OpCounter<F>::counters.sin_count;
857         return {std::cos(a._v)};
858     }
859
860     template<typename F>
861     OpCounter<F> sqrt(const OpCounter<F>& a)
862     {
863         ++OpCounter<F>::counters.sqrt_count;
864         return {std::sqrt(a._v)};
865     }
866
867     template<typename F>
868     OpCounter<F> abs(const OpCounter<F>& a)
869     {
870         ++OpCounter<F>::counters.comparison_count;
871         return {std::abs(a._v)};
872     }
873 }
874 }
875
876 #endif // __OPCOUNTER__

```

## 5.12 src/pde.hh File Reference

solvers for partial differential equations

```

#include <vector>
#include "newton.hh"

```

### Classes

- class `hdnum::StationarySolver< M >`  
*Stationary problem solver. E.g. for elliptic problems.*

### Functions

- template<class N , class G >  
void `hdnum::pde_gnuplot2d` (const std::string &fname, const Vector< N > solution, const G &grid)  
*gnuplot output for stationary state*

### 5.12.1 Detailed Description

solvers for partial differential equations

## 5.13 pde.hh

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

```

1 // -*- tab-width: 4; indent-tabs-mode: nil -*-
2 #ifndef HDNUM_PDE_HH
3 #define HDNUM_PDE_HH
4
5 #include<vector>
6 #include "newton.hh"
7
12 namespace hdnum {
13
22     template<class M>
23     class StationarySolver
24     {
25     public:
27         typedef typename M::size_type size_type;
28
30         typedef typename M::time_type time_type;
31
33         typedef typename M::number_type number_type;
34
36         StationarySolver (const M& model_)
37             : model(model_), x(model.size())
38         {
39         }
40
42         void solve ()
43         {
44             const size_t n_dofs = model.size();
45
46             DenseMatrix<number_type> A(n_dofs,n_dofs,0.);
47             Vector<number_type> b(n_dofs,0.);
48
49             Vector<number_type> s(n_dofs);           // scaling factors
50             Vector<size_t> p(n_dofs);                // row permutations
51             Vector<size_t> q(n_dofs);                // column permutations
52
53             number_type t = 0.;
54
55             x = 0.;
56
57             model.f_x(t, x, A);
58             model.f(t, x, b);
59
60             b*=-1.;
61
62             row_equilibrate(A,s);                     // equilibrate rows
63             lr_fullpivot(A,p,q);                     // LR decomposition of A
64             apply_equilibrate(s,b);                  // equilibration of right hand side
65             permute_forward(p,b);                    // permutation of right hand side
66             solveL(A,b,b);                           // forward substitution
67             solveR(A,x,b);                           // backward substitution
68             permute_backward(q,x);                   // backward permutation
69         }
70
72         const Vector<number_type>& get_state () const
73         {
74             return x;
75         }
76
78         size_type get_order () const
79         {
80             return 2;
81         }
82
83     private:
84         const M& model;
85         Vector<number_type> x;
86     };
87
88
90     template<class N, class G>
91     inline void pde_gnuplot2d (const std::string& fname, const Vector<N> solution,
92                               const G & grid)

```

```

93  {
94
95      const std::vector<Vector<N> > coords = grid.getNodeCoordinates();
96      Vector<typename G::size_type> gsize = grid.getGridSize();
97
98      std::fstream f(fname.c_str(),std::ios::out);
99      // f << "set dgrid3d ";
100
101      // f << gsize[0] << "," << gsize[1] << std::endl;
102
103      // f << "set hidden3d" << std::endl;
104      f << "set ticslevel 0" << std::endl;
105      f << "splot \"-\" using 1:2:3 with points" << std::endl;
106      f << "#" << std::endl;
107      for (typename Vector<N>::size_type n=0; n<solution.size(); n++)
108      {
109          for (typename Vector<N>::size_type d=0; d<coords[n].size(); d++){
110              f << std::scientific << std::showpoint
111                << std::setprecision(16) << coords[n][d] << " ";
112          }
113
114          f << std::scientific << std::showpoint
115            << std::setprecision(solution.precision()) << solution[n];
116
117          f << std::endl;
118      }
119      f << "end" << std::endl;
120      f << "pause -1" << std::endl;
121      f.close();
122  }
123
124
125 }
126 #endif

```

## 5.14 src/precision.hh File Reference

find machine precision for given float type

### Functions

- `template<typename X >`  
`int hdnum::precision (X &eps)`

#### 5.14.1 Detailed Description

find machine precision for given float type

## 5.15 precision.hh

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

```

1 // -*- tab-width: 4; indent-tabs-mode: nil -*-
2 #ifndef HDNUM_PRECISION_HH
3 #define HDNUM_PRECISION_HH
4
5 namespace hdnum {
6
7     // find largest eps such that 0.5 + eps > 0.5
8     template<typename X>
9     int precision (X& eps)
10     {
11         X x,large,largex,two;
12         large = 0.5;
13         two = 2.0;
14         x = 0.5;
15         largex = large+x;
16     }
17 }

```



```

20         int i(0);
21         while (largex>large)
22         {
23             eps = x;
24             i = i+1;
25             //          std::cout << i << " " << std::scientific << std::showpoint
26             //          << std::setprecision(15) << large+x << " " << x << std::endl;
27             x = x/two;
28             largex = large+x;
29         }
30         return i;
31     }
32 } // namespace hdnum
33
34
35 #endif

```

## 5.16 src/qr.hh File Reference

This file implements QR decomposition using Gram-Schmidt method.

```

#include <cmath>
#include <utility>
#include "densematrix.hh"
#include "vector.hh"

```

### Functions

- `template<class T >`  
`DenseMatrix< T > hdnum::gram\_schmidt (const DenseMatrix< T > &A)`  
*computes orthonormal basis of  $Im(A)$  using classical Gram-Schmidt*
- `template<class T >`  
`DenseMatrix< T > hdnum::modified\_gram\_schmidt (const DenseMatrix< T > &A)`  
*computes orthonormal basis of  $Im(A)$  using modified Gram-Schmidt*
- `template<class T >`  
`DenseMatrix< T > hdnum::qr\_gram\_schmidt\_simple (DenseMatrix< T > &Q)`  
*computes qr decomposition using modified Gram-Schmidt - works only with small ( $m>n$ ) and square matrices*
- `template<class T >`  
`DenseMatrix< T > hdnum::qr\_gram\_schmidt (DenseMatrix< T > &Q)`  
*computes qr decomposition using modified Gram-Schmidt - works only with small ( $m>n$ ) and square matrices*
- `template<class T >`  
`DenseMatrix< T > hdnum::qr\_gram\_schmidt\_pivoting (DenseMatrix< T > &Q, Vector< int > &p, int &rank, T threshold=0.00000000001)`  
*computes qr decomposition using modified Gram-Schmidt and pivoting - works with all types of matrices*
- `template<typename T >`  
`void hdnum::permute\_forward (DenseMatrix< T > &A, Vector< int > &p)`  
*applies a permutation vector to a matrix*

### 5.16.1 Detailed Description

This file implements QR decomposition using Gram-Schmidt method.

## 5.16.2 Function Documentation

### 5.16.2.1 gram\_schmidt()

```
template<class T >
DenseMatrix< T > hdnum::gram_schmidt (
    const DenseMatrix< T > & A )
```

computes orthonormal basis of  $\text{Im}(A)$  using classical Gram-Schmidt

#### Template Parameters

<i>hdnum::DenseMatrix&lt;T&gt;</i>	A
------------------------------------	---

#### Example:

```
hdnum::DenseMatrix<double> A({{2, 9},
                             {1, -5}});
hdnum::DenseMatrix<double> Q(hdnum::gram_schmidt(A));
std::cout << "A = " << A << std::endl;
std::cout << "Q = " << Q << std::endl;
```

#### Output:

```
A =
      0      1
0  2.000e+00  9.000e+00
1  1.000e+00 -5.000e+00

Q =
      0      1
0  8.944e-01  4.472e-01
1  4.472e-01 -8.944e-01
```

### 5.16.2.2 modified\_gram\_schmidt()

```
template<class T >
DenseMatrix< T > hdnum::modified_gram_schmidt (
    const DenseMatrix< T > & A )
```

computes orthonormal basis of  $\text{Im}(A)$  using modified Gram-Schmidt

#### Template Parameters

<i>hdnum::DenseMatrix&lt;T&gt;</i>	A
------------------------------------	---

#### Example:

```
hdnum::DenseMatrix<double> A({{2, 9},
                             {1, -5}});
hdnum::DenseMatrix<double> Q(hdnum::modified_gram_schmidt(A));
std::cout << "A = " << A << std::endl;
std::cout << "Q = " << Q << std::endl;
```

**Output:**

```

A =
      0      1
0  2.000e+00  9.000e+00
1  1.000e+00 -5.000e+00

Q =
      0      1
0  8.944e-01  4.472e-01
1  4.472e-01 -8.944e-01

```

**5.16.2.3 permute\_forward()**

```

template<typename T >
void hdnum::permute_forward (
    DenseMatrix< T > & A,
    Vector< int > & p )

```

applies a permutation vector to a matrix

**Template Parameters**

<code>hdnum::DenseMatrix&lt;T&gt;</code>	<code>A</code>
--	----------------

**Parameters**

<code>hdnum::Vector&lt;int&gt;</code>	<code>p</code>
---------------------------------------	----------------

**Example:**

```

hdnum::DenseMatrix<double> A({{2, 9},
                             {1, -5}});
hdnum::Vector<int> p({1, 0});
hdnum::permute_forward(A, p);
std::cout << "A = " << A << std::endl;
std::cout << "p = " << p << std::endl;

```

**Output:**

```

A =
      0      1
0  9.000e+00  2.000e+00
1 -5.000e+00  1.000e+00

p =
 [ 0]      0
 [ 1]      1

```

**5.16.2.4 qr\_gram\_schmidt()**

```

template<class T >
DenseMatrix< T > hdnum::qr_gram_schmidt (
    DenseMatrix< T > & Q )

```

computes qr decomposition using modified Gram-Schmidt - works only with small ( $m > n$ ) and square matrices

## Template Parameters

<code>hdnum::DenseMatrix&lt;T&gt;</code>	Q
--	---

## Example:

```
hdnum::DenseMatrix<double> A({{2, 9},
                             {1, -5}});
hdnum::DenseMatrix<double> Q(A);
hdnum::DenseMatrix<double> R(hdnum::qr_gram_schmidt(Q));
std::cout << "A = " << A << std::endl;
std::cout << "Q = " << Q << std::endl;
std::cout << "R = " << R << std::endl;
std::cout << "QR = " << Q*R << std::endl;
```

## Output:

```
A =
      0      1
0  2.000e+00  9.000e+00
1  1.000e+00 -5.000e+00

Q =
      0      1
0  8.944e-01  4.472e-01
1  4.472e-01 -8.944e-01

R =
      0      1
0  2.236e+00  5.814e+00
1  0.000e+00  8.497e+00

QR =
      0      1
0  2.000e+00  9.000e+00
1  1.000e+00 -5.000e+00
```

## 5.16.2.5 qr\_gram\_schmidt\_pivoting()

```
template<class T >
DenseMatrix< T > hdnum::qr_gram_schmidt_pivoting (
    DenseMatrix< T > & Q,
    Vector< int > & p,
    int & rank,
    T threshold = 0.00000000001 )
```

computes qr decomposition using modified Gram-Schmidt and pivoting - works with all types of matrices

## Template Parameters

<code>hdnum::DenseMatrix&lt;T&gt;</code>	Q
<code>T</code>	threshold (optional)

## Parameters

<code>hdnum::Vector&lt;int&gt;</code>	p
<code>int</code>	rank

**Example:**

```

hdnum::DenseMatrix<double> A({{5, 2, 3},
                             {11, 9, 2}});
hdnum::DenseMatrix<double> Q(A);
hdnum::Vector<int> p(A.colsize());
int rank;
hdnum::DenseMatrix<double> R(hdnum::qr_gram_schmidt_pivoting(Q, p, rank));
hdnum::DenseMatrix<double> Q_right_dimension(A.rowsize(), rank);
hdnum::DenseMatrix<double> R_right_dimension(rank, A.colsize());
for (int i = 0; i < Q_right_dimension.rowsize(); i++) {
    for (int j = 0; j < Q_right_dimension.colsize(); j++) {
        Q_right_dimension(i, j) = Q(i, j);
    }
}
for (int i = 0; i < R_right_dimension.rowsize(); i++) {
    for (int j = 0; j < R_right_dimension.colsize(); j++) {
        R_right_dimension(i, j) = R(i, j);
    }
}
hdnum::DenseMatrix<double> QR(Q_right_dimension*R_right_dimension);
hdnum::permute_forward(QR, p);
std::cout << "A = " << A << std::endl;
std::cout << "Q = " << Q_right_dimension << std::endl;
std::cout << "R = " << R_right_dimension << std::endl;
std::cout << "QR = " << QR << std::endl;

```

**Output:**

```

A =
      0      1      2
0  5.000e+00  2.000e+00  3.000e+00
1  1.100e+01  9.000e+00  2.000e+00

Q =
      0      1
0  4.138e-01 -9.104e-01
1  9.104e-01  4.138e-01

R =
      0      1      2
0  1.208e+01  9.021e+00  3.062e+00
1  0.000e+00  1.903e+00 -1.903e+00

QR =
      0      1      2
0  5.000e+00  2.000e+00  3.000e+00
1  1.100e+01  9.000e+00  2.000e+00

```

**5.16.2.6 qr\_gram\_schmidt\_simple()**

```

template<class T >
DenseMatrix< T > hdnum::qr_gram_schmidt_simple (
    DenseMatrix< T > & Q )

```

computes qr decomposition using modified Gram-Schmidt - works only with small ( $m > n$ ) and square matrices

**Template Parameters**

<code>hdnum::DenseMatrix&lt;T&gt;</code>	<code>Q</code>
--	----------------

**Example:**

```

hdnum::DenseMatrix<double> A({{2, 9},
                             {1, -5}});
hdnum::DenseMatrix<double> Q(A);
hdnum::DenseMatrix<double> R(hdnum::qr_gram_schmidt_simple(Q));

```

```
std::cout << "A = " << A << std::endl;
std::cout << "Q = " << Q << std::endl;
std::cout << "R = " << R << std::endl;
std::cout << "QR = " << Q*R << std::endl;
```

### Output:

```
A =
      0      1
0  2.000e+00  9.000e+00
1  1.000e+00 -5.000e+00

Q =
      0      1
0  8.944e-01  4.472e-01
1  4.472e-01 -8.944e-01

R =
      0      1
0  2.236e+00  5.814e+00
1  0.000e+00  8.497e+00

QR =
      0      1
0  2.000e+00  9.000e+00
1  1.000e+00 -5.000e+00
```

## 5.17 qr.hh

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

```
1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 /*
3  * File:    qr.hh
4  * Author:  Raphael Vogt <cx238@stud.uni-heidelberg.de>
5  *
6  * Created on August 30, 2020
7  */
8
9 #ifndef HDNUM_QR_HH
10 #define HDNUM_QR_HH
11
12 #include <cmath>
13 #include <utility>
14
15 #include "densematrix.hh"
16 #include "vector.hh"
17
18 namespace hdnum {
19
20 template <class T>
21 DenseMatrix<T> gram_schmidt(const DenseMatrix<T>& A) {
22     DenseMatrix<T> Q(A);
23
24     // for all columns except the first
25     for (int k = 1; k < Q.colsize(); k++) {
26         // orthogonalize column k against all previous
27         for (int j = 0; j < k; j++) {
28             // compute factor
29             T sum_nom(0.0);
30             T sum_denom(0.0);
31             for (int i = 0; i < Q.rowsize(); i++) {
32                 sum_nom += A[i][k] * Q[i][j];
33                 sum_denom += Q[i][j] * Q[i][j];
34             }
35             // modify
36             T alpha = sum_nom / sum_denom;
37             for (int i = 0; i < Q.rowsize(); i++) Q[i][k] -= alpha * Q[i][j];
38         }
39     }
40
41     // compute norm of column j
42     for (int j = 0; j < Q.colsize(); j++) {
43         T sum(0.0);
44         for (int i = 0; i < Q.rowsize(); i++) sum += Q[i][j] * Q[i][j];
45         sum = sqrt(sum);
46         // scale
47         for (int i = 0; i < Q.rowsize(); i++) Q[i][j] = Q[i][j] / sum;
48     }
49 }
```

```

79     }
80     return Q;
81 }
82
83 template <class T>
84 DenseMatrix<T> modified_gram_schmidt(const DenseMatrix<T>& A) {
85     DenseMatrix<T> Q(A);
86
87     for (int k = 0; k < Q.colsize(); k++) {
88         // modify all later columns with column k
89         for (int j = k + 1; j < Q.colsize(); j++) {
90             // compute factor
91             T sum_nom(0.0);
92             T sum_denom(0.0);
93             for (int i = 0; i < Q.rowsize(); i++) {
94                 sum_nom += Q[i][j] * Q[i][k];
95                 sum_denom += Q[i][k] * Q[i][k];
96             }
97             // modify
98             T alpha = sum_nom / sum_denom;
99             for (int i = 0; i < Q.rowsize(); i++) Q[i][j] -= alpha * Q[i][k];
100         }
101     }
102
103     for (int j = 0; j < Q.colsize(); j++) {
104         // compute norm of column j
105         T sum(0.0);
106         for (int i = 0; i < Q.rowsize(); i++) sum += Q[i][j] * Q[i][j];
107         sum = sqrt(sum);
108         // scale
109         for (int i = 0; i < Q.rowsize(); i++) Q[i][j] = Q[i][j] / sum;
110     }
111     return Q;
112 }
113
114 template <class T>
115 DenseMatrix<T> qr_gram_schmidt_simple(DenseMatrix<T>& Q) {
116     // save matrix A, before it's replaced with Q
117     DenseMatrix<T> A(Q);
118
119     // create matrix R
120     DenseMatrix<T> R(Q.colsize(), Q.colsize());
121
122     // start orthogonalizing
123     for (int k = 0; k < Q.colsize(); k++) {
124         // modify all later columns with column k
125         for (int j = k + 1; j < Q.colsize(); j++) {
126             // compute factor
127             T sum_nom(0.0);
128             T sum_denom(0.0);
129             for (int i = 0; i < Q.rowsize(); i++) {
130                 sum_nom += Q(i, j) * Q(i, k);
131                 sum_denom += Q(i, k) * Q(i, k);
132             }
133
134             T alpha = sum_nom / sum_denom;
135             for (int i = 0; i < Q.rowsize(); i++) Q(i, j) -= alpha * Q(i, k);
136         }
137     }
138
139     // add values to R, except main diagonal
140     for (int i = 1; i < R.colsize(); i++) {
141         for (int j = 0; j < i; j++) {
142             T sum_nom(0.0);
143             T sum_l2nom(0.0);
144             for (int k = 0; k < Q.rowsize(); k++) {
145                 sum_nom += A(k, i) * Q(k, j);
146                 sum_l2nom += Q(k, j) * Q(k, j);
147             }
148             sum_l2nom = sqrt(sum_l2nom);
149             // add element
150             R(j, i) = sum_nom / sum_l2nom;
151         }
152     }
153
154     // add missing values and scale
155     for (int j = 0; j < Q.colsize(); j++) {
156         // compute norm of column j
157         T sum(0.0);
158         for (int i = 0; i < Q.rowsize(); i++) sum += Q(i, j) * Q(i, j);
159         sum = sqrt(sum);
160         // add main diagonal to R
161         R(j, j) = sum;
162         // scale Q
163         for (int i = 0; i < Q.rowsize(); i++) Q(i, j) = Q(i, j) / sum;
164     }
165     return R;
166 }

```

```

235
277 template <class T>
278 DenseMatrix<T> qr_gram_schmidt(DenseMatrix<T>& Q) {
279     // create matrix R
280     DenseMatrix<T> R(Q.colsize(), Q.colsize());
281
282     // start orthogonalizing
283     for (int k = 0; k < Q.colsize(); k++) {
284         // compute norm of column k
285         T sum_denom(0.0);
286         for (int i = 0; i < Q.rowsize(); i++) {
287             sum_denom += Q(i, k) * Q(i, k);
288         }
289
290         // fill the main diagonal of R with elements
291         sum_denom = sqrt(sum_denom);
292         R(k, k) = sum_denom;
293
294         // scale column k to the main diagonal
295         for (int i = 0; i < Q.rowsize(); i++) {
296             Q(i, k) /= R(k, k);
297         }
298
299         // modify all later columns with column k
300         for (int j = k + 1; j < Q.colsize(); j++) {
301             // compute norm of column j
302             T sum_nom(0.0);
303             for (int i = 0; i < Q.rowsize(); i++) {
304                 sum_nom += Q(i, k) * Q(i, j);
305             }
306             // insert missing elements to R
307             R(k, j) = sum_nom;
308
309             // orthogonalize column j
310             for (int i = 0; i < Q.rowsize(); i++) {
311                 Q(i, j) -= Q(i, k) * R(k, j);
312             }
313         }
314     }
315     return R;
316 }
317
381 template <class T>
382 DenseMatrix<T> qr_gram_schmidt_pivoting(DenseMatrix<T>& Q, Vector<int>& p, int& rank, T
    threshold=0.00000000001) {
383     // check if permutation vector has the right size
384     if (p.size() != Q.colsize()) {
385         HDNUM_ERROR("Permutation Vector incompatible with Matrix!");
386     }
387
388     // initialize permutation vector
389     for (int i = 0; i < p.size(); i++) {
390         p[i] = i;
391     }
392
393     // initialize rank
394     rank = 0;
395
396     // save matrix A, before it's replaced with Q
397     DenseMatrix<T> A(Q);
398
399     // create Matrix R
400     hdn::DenseMatrix<T> R(A.colsize(), A.colsize());
401
402     // start orthogonalizing
403     for (int k = 0; k < Q.colsize(); k++) {
404         // find column with highest norm
405         // compute norm of column k
406         T norm_k(0.0);
407         for (int r = 0; r < Q.rowsize(); r++) {
408             norm_k += Q(r, k) * Q(r, k);
409         }
410         norm_k = sqrt(norm_k);
411
412         // compare norm of column k to the following column norms
413         for (int c = k+1; c < Q.colsize(); c++) {
414             T norm(0.0);
415             for (int r = 0; r < Q.rowsize(); r++) {
416                 norm += Q(r, c) * Q(r, c);
417             }
418             norm = sqrt(norm);
419             // store permutation
420             if (norm > norm_k) {
421                 p[k] = c;
422             }
423         }
424     }

```



```

425         // swap columns if necessary
426         if (p[k] > k) {
427             for (int r = 0; r < Q.rowsize(); r++) {
428                 T temp_Q = Q(r, k);
429                 Q(r, k) = Q(r, p[k]);
430                 Q(r, p[k]) = temp_Q;
431             }
432             p[p[k]] = k;
433
434             // compute norm of the new column k
435             norm_k = 0;
436             for (int i = 0; i < Q.rowsize(); i++) {
437                 norm_k += Q(i, k) * Q(i, k);
438             }
439             norm_k = sqrt(norm_k);
440         }
441
442         // if norm of column k > threshold -> column k is linear independent
443         if (norm_k > threshold) {
444             rank++;
445         } else {
446             break;
447         }
448
449         // modify all later columns with column k
450         for (int j = k + 1; j < Q.colsize(); j++) {
451             // compute factor
452             T sum_nom(0.0);
453             T sum_denom(0.0);
454             for (int i = 0; i < Q.rowsize(); i++) {
455                 sum_nom += Q(i, j) * Q(i, k);
456                 sum_denom += Q(i, k) * Q(i, k);
457             }
458
459             T alpha = sum_nom / sum_denom;
460             for (int i = 0; i < Q.rowsize(); i++) Q(i, j) -= alpha * Q(i, k);
461         }
462     }
463
464     // add values to R, except main diagonal
465     for (int i = 1; i < R.colsize(); i++) {
466         for (int j = 0; j < i; j++) {
467             T sum_nom(0.0);
468             T sum_l2nom(0.0);
469             for (int k = 0; k < Q.rowsize(); k++) {
470                 sum_nom += A(k, p[i]) * Q(k, j);
471                 sum_l2nom += Q(k, j) * Q(k, j);
472             }
473             sum_l2nom = sqrt(sum_l2nom);
474             // add element
475             R(j, i) = sum_nom / sum_l2nom;
476         }
477     }
478
479     // add missing values and scale
480     for (int j = 0; j < Q.colsize(); j++) {
481         // compute norm of column j
482         T sum(0.0);
483         for (int i = 0; i < Q.rowsize(); i++) sum += Q(i, j) * Q(i, j);
484         sum = sqrt(sum);
485         // add main diagonal to R
486         R(j, j) = sum;
487         // scale Q
488         for (int i = 0; i < Q.rowsize(); i++) Q(i, j) = Q(i, j) / sum;
489     }
490
491     return R;
492 }
493
494 template<typename T>
495 void permute_forward(DenseMatrix<T>& A, Vector<int>& p) {
496     // check if permutation vector has the right size
497     if (p.size() != A.colsize()) {
498         HDNUM_ERROR("Permutation Vector incompatible with Matrix!");
499     }
500
501     // permute the columns
502     for (int k = 0; k < p.size(); k++) {
503         if (p[k] != k) {
504             // swap column
505             for (int j=0; j < A.rowsize(); j++) {
506                 T temp_A = A(j, k);
507                 A(j, k) = A(j, p[k]);
508                 A(j, p[k]) = temp_A;
509             }
510             // swap inside permutation vector
511             int temp_p = p[k];

```

```

542         p[k] = p[temp_p];
543         p[temp_p] = temp_p;
544     }
545 }
546 }
547
548 } // namespace hdnum
549
550 #endif

```

## 5.18 src/qrhousholder.hh File Reference

This file implements QR decomposition using housholder transformation.

```

#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>
#include "densematrix.hh"
#include "vector.hh"

```

### Functions

- `template<class REAL >`  
`DenseMatrix< REAL > hdnum::creat_I_matrix (size_t n)`
- `template<typename REAL >`  
`size_t hdnum::sgn (REAL val)`  
*Function that return the sign of a number.*
- `template<class REAL >`  
`void hdnum::qrhousholder (DenseMatrix< REAL > &A, hdnum::Vector< REAL > &v)`  
*Function that calculate the QR decoposition in place the elements of A will be replaced with the elements of  $v_{\leftarrow i}$  vectors and the upper diagonals elements of R and the diagonal elements of R will be saved in vectro v.*
- `template<class REAL >`  
`DenseMatrix< REAL > hdnum::qrhousholderexplicitQ (DenseMatrix< REAL > &A, hdnum::Vector< REAL > &v, bool show_Hi=false)`  
*Funktion that calculate the QR decoposition in place and return Q the elements of A will be replaced with the elements of  $v_{\leftarrow i}$  vectors and the upper diagonals elements of R and the diagonal elements of R will be saved in vectro v.*

### 5.18.1 Detailed Description

This file implements QR decomposition using housholder transformation.

### 5.18.2 Function Documentation

### 5.18.2.1 qrhousholder()

```
template<class REAL >
void hdnum::qrhousholder (
    DenseMatrix< REAL > & A,
    hdnum::Vector< REAL > & v )
```

Funktion that calculate the QR decoposition in place the elements of A will be replaced with the elements of  $v_{\leftarrow}$  {i}vectors and the upper diagonals elements of R and the diagonal elements of R will be saved in vectro v.

## Template Parameters

<i>A</i>	the Matrix
<i>v</i>	oa vector of <a href="#">hdnum::Vector</a>

## 5.18.2.2 qrhousholderexplicitQ()

```
template<class REAL >
DenseMatrix< REAL > hdnum::qrhousholderexplicitQ (
    DenseMatrix< REAL > & A,
    hdnum::Vector< REAL > & v,
    bool show_Hi = false )
```

Funktion that calculate the QR decoposition in place and return Q the elements of A will be replaced with the elements of  $v_{\{i\}}$  vectors and the upper diagonals elements of R and the diagonal elements of R will be saved in vectro v.

## Template Parameters

<i>A</i>	the Matrix
<i>v</i>	oa vector of <a href="#">hdnum::Vector</a>

## Returns

Q matrix

## 5.19 qrhousholder.hh

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

```
1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 /*
3  * File:    qrhousholder
4  * Author:  Ahmad Fadl <abohmaid@windowslive.com>
5  *
6  * Created on August 25, 2020
7  */
8
9 #ifndef HDNUM_QRHOUSHOLDER_HH
10 #define HDNUM_QRHOUSHOLDER_HH
11 #include <cmath>
12 #include <cstdlib>
13 #include <fstream>
14 #include <iomanip>
15 #include <iostream>
16 #include <sstream>
17 #include <string>
18
19 #include "densematrix.hh"
20 #include "vector.hh"
21
22 namespace hdnum {
23 template <class REAL>
24 DenseMatrix<REAL> creat_I_matrix(size_t n) {
25     DenseMatrix<REAL> res(n, n, 0);
26     for (size_t i = 0; i < n; i++) {
27         res(i, i) = 1;
28     }
29     return res;
30 }
31 }
32
33
```

```

36 template <typename REAL>
37 size_t sgn(REAL val) {
38     return (REAL(0) < val) - (val < REAL(0));
39 }
40
41 template <class REAL>
42 void qrhousholder(DenseMatrix<REAL>& A, hdnnum::Vector<REAL>& v) {
43     auto m = A.rowsize();
44     auto n = A.colsize();
45     for (size_t j = 0; j < n; j++) {
46         REAL s = 0;
47         for (size_t i = j; i < m; i++) {
48             s = s + pow(A(i, j), 2);
49         }
50         s = sqrt(s);
51         v[j] = (-1.0) * sgn(A(j, j)) * s;
52         REAL fak = sqrt(s * (s + std::abs(A(j, j))));
53         A(j, j) = A(j, j) - v[j];
54         for (size_t k = j; k < m; k++) {
55             A(k, j) = A(k, j) / fak;
56         }
57         for (size_t i = j + 1; i < n; i++) {
58             s = 0;
59             for (size_t k = j; k < m; k++) {
60                 s = s + (A(k, j) * A(k, i));
61             }
62             for (size_t k = j; k < m; k++) {
63                 A(k, i) = A(k, i) - (A(k, j) * s);
64             }
65         }
66         // normalize the vi vectors again
67         for (size_t i = m; i >= 0; i--) {
68             A(i, j) = A(i, j) * fak;
69             if (i == j) {
70                 break;
71             }
72         }
73     }
74 }
75
76 template <class REAL>
77 DenseMatrix<REAL> qrhousholderexplicitQ(DenseMatrix<REAL>& A,
78                                         hdnnum::Vector<REAL>& v,
79                                         bool show_Hi = false) {
80     auto m = A.rowsize();
81     auto n = A.colsize();
82     auto I = creat_I_matrix<REAL>(std::max(m, n));
83
84     DenseMatrix<REAL> Q(m, m, 0);
85     for (size_t j = 0; j < n; j++) {
86         REAL s = 0;
87         for (size_t i = j; i < m; i++) {
88             s = s + pow(A(i, j), 2);
89         }
90         s = sqrt(s);
91         v[j] = (-1.0) * sgn(A(j, j)) * s;
92         REAL fak = sqrt(s * (s + std::abs(A(j, j))));
93         A(j, j) = A(j, j) - v[j];
94         for (size_t k = j; k < m; k++) {
95             A(k, j) = A(k, j) / fak;
96         }
97         for (size_t i = j + 1; i < n; i++) {
98             s = 0;
99             for (size_t k = j; k < m; k++) {
100                 s = s + (A(k, j) * A(k, i));
101             }
102             for (size_t k = j; k < m; k++) {
103                 A(k, i) = A(k, i) - (A(k, j) * s);
104             }
105         }
106         // normalize the vi vectors again
107         for (size_t i = m; i >= 0; i--) {
108             A(i, j) = A(i, j) * fak;
109             if (i == j) {
110                 break;
111             }
112         }
113     }
114     // create qi and multiply them
115     if (m >= n) {
116         for (size_t j = 0; j < n; j++) {
117             DenseMatrix<REAL> TempQ(m, m, 0.0);
118             DenseMatrix<REAL> v1(m, 1, 0.0);
119             DenseMatrix<REAL> v1t(1, m, 0.0);
120             hdnnum::Vector<double> v__i(m, 0);
121             for (size_t i = 0; i < m; i++) {
122                 if (i < j) {
123                     v1(i, 0) = 0;
124                 }
125             }
126         }
127     }
128 }

```

```

142         v__i[i] = 0;
143         continue;
144     }
145     v1(i, 0) = A(i, j);
146
147     v__i[i] = A(i, j);
148 }
149 v1t = v1.transpose();
150
151 TempQ = (v1 * v1t);
152
153 TempQ *= (-2.0);
154
155 TempQ /= v__i.two_norm_2();
156
157 TempQ += I;
158 if (show_Hi) {
159     std::cout << "H[" << j + 1 << "]" << TempQ;
160 }
161 if (j == 0) {
162     Q = TempQ;
163 }
164 if (j > 0) {
165     Q = Q * TempQ;
166 }
167 }
168 }
169 if (n > m) {
170     for (size_t j = 0; j < m; j++) {
171         DenseMatrix<REAL> TempQ(m, m, 0.0);
172         DenseMatrix<REAL> v1(m, 1, 0.0);
173         DenseMatrix<REAL> v1t(1, m, 0.0);
174         hdnum::Vector<double> v__i(m, 0);
175         for (size_t i = 0; i < m; i++) {
176             if (i < j) {
177                 v1(i, 0) = 0;
178
179                 v__i[i] = 0;
180                 continue;
181             }
182             v1(i, 0) = A(i, j);
183
184             v__i[i] = A(i, j);
185         }
186         v1t = v1.transpose();
187
188         TempQ = (v1 * v1t);
189
190         TempQ *= (-2.0);
191
192         TempQ /= v__i.two_norm_2();
193
194         TempQ += I;
195         if (show_Hi) {
196             std::cout << "H[" << j + 1 << "]" << TempQ;
197         }
198         if (j == 0) {
199             Q = TempQ;
200         }
201         if (j > 0) {
202             Q = Q * TempQ;
203         }
204     }
205 }
206 return Q;
207 }
208 } // namespace hdnum
209 #endif

```

## 5.20 src/rungekutta.hh File Reference

```

#include "vector.hh"
#include "newton.hh"

```

### Classes

- class [hdnum::ImplicitRungeKuttaStepProblem< M >](#)

*Nonlinear problem we need to solve to do one step of an implicit Runge Kutta method.*

- class `hdnum::RungeKutta< M, S >`  
*classical Runge-Kutta method (order n with n stages)*

## Functions

- template<class M , class S >  
void `hdnum::ordertest` (const M &model, S solver, typename M::number\_type T, typename M::number\_type h\_0, int l)  
*Test convergence order of an ODE solver applied to a model problem.*

### 5.20.1 Detailed Description

@general Runge-Kutta solver

### 5.20.2 Function Documentation

#### 5.20.2.1 ordertest()

```
template<class M , class S >
void hdnum::ordertest (
    const M & model,
    S solver,
    typename M::number_type T,
    typename M::number_type h_0,
    int l )
```

Test convergence order of an ODE solver applied to a model problem.

#### Template Parameters

<i>M</i>	Type of model
<i>S</i>	Type of ODE solver

#### Parameters

<i>model</i>	Model problem
<i>solver</i>	ODE solver
<i>T</i>	Solve to time T
<i>dt</i>	Roughest time step size
<i>l</i>	Number of different time step sizes dt, dt/2, dt/4, ...

## 5.21 rungekutta.hh

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

```

1 // -*- tab-width: 4; indent-tabs-mode: nil -*-
2 #ifndef HDNUM_RUNGEKUTTA_HH
3 #define HDNUM_RUNGEKUTTA_HH
4
5 #include "vector.hh"
6 #include "newton.hh"
7
12 namespace hdnum {
15     template<class M>
16     class ImplicitRungeKuttaStepProblem
17     {
18     public:
19         typedef typename M::size_type size_type;
20
21         typedef typename M::time_type time_type;
22
23         typedef typename M::number_type number_type;
24
25         ImplicitRungeKuttaStepProblem (const M& model_,
26                                         DenseMatrix<number_type> A_,
27                                         Vector<number_type> b_,
28                                         Vector<number_type> c_,
29                                         time_type t_,
30                                         Vector<number_type> u_,
31                                         time_type dt_)
32             : model(model_) , u(model.size())
33         {
34             A = A_;
35             b = b_;
36             c = c_;
37             s = A_.rowsize ();
38             dt = dt_;
39             n = model.size();
40             t = t_;
41             u = u_;
42         }
43
44         std::size_t size () const
45         {
46             return n*s;
47         }
48
49         void F (const Vector<number_type>& x, Vector<number_type>& result) const
50         {
51             Vector<Vector<number_type> > xx (s);
52             for (int i = 0; i < s; i++)
53             {
54                 xx[i].resize(n,number_type(0));
55                 for(int k = 0; k < n; k++)
56                 {
57                     xx[i][k] = x[i*n + k];
58                 }
59             }
60             Vector<Vector<number_type> > f (s);
61             for (int i = 0; i < s; i++)
62             {
63                 f[i].resize(n, number_type(0));
64                 model.f(t + c[i] * dt, u + xx[i], f[i]);
65             }
66             Vector<Vector<number_type> > hr (s);
67             for (int i = 0; i < s; i++)
68             {
69                 hr[i].resize(n, number_type(0));
70             }
71             for (int i = 0; i < s; i++)
72             {
73                 Vector<number_type> sum (n, number_type(0));
74                 for (int j = 0; j < s; j++)
75                 {
76                     sum.update(dt*A[i][j], f[j]);
77                 }
78                 hr[i] = xx[i] - sum;
79             }
80             //translating hr into result
81             for (int i = 0; i < s; i++)
82             {
83                 for (int j = 0; j < n; j++)
84                 {
85                     result[i*n + j] = hr[i][j];
86                 }
87             }
88         }
89     }
90 }

```



```

95
96
97 void F_x (const Vector<number_type>& x, DenseMatrix<number_type>& result) const
98 {
99     Vector<Vector<number_type> > xx (s);
100     for (int i = 0; i < s; i++)
101     {
102         xx[i].resize(n);
103         for(int k = 0; k < n; k++)
104         {
105             xx[i][k] = x[i*n + k];
106         }
107     }
108     DenseMatrix<number_type> I (n, n, 0.0);
109     for (int i = 0; i < n; i++)
110     {
111         I[i][i] = 1.0;
112     }
113     for (int i = 0; i < s; i++)
114     {
115         for (int j = 0; j < s; j++)
116         {
117             DenseMatrix<number_type> J (n, n, number_type(0));
118             DenseMatrix<number_type> H (n, n, number_type(0));
119             model.f_x(t+c[j]*dt, u + xx[j],H);
120             J.update(-dt*A[i][j],H);
121             if(i==j) //add I on diagonal
122             {
123                 J+=I;
124             }
125             for (int k = 0; k < n; k++)
126             {
127                 for (int l = 0; l < n; l++)
128                 {
129                     result[n * i + k][n * j + l] = J[k][l];
130                 }
131             }
132         }
133     }
134 }
135
136 private:
137     const M& model;
138     time_type t, dt;
139     Vector<number_type> u;
140     int n, s; // dimension of matrix A and model.size
141     DenseMatrix<number_type> A; // A, b, c as in the butcher tableau
142     Vector<number_type> b;
143     Vector<number_type> c;
144 };
145
146
147
148
149
150
151
152
153
154
155
156 template<class M, class S = Newton>
157 class RungeKutta
158 {
159 public:
160     typedef typename M::size_type size_type;
161     typedef typename M::time_type time_type;
162     typedef typename M::number_type number_type;
163
164     RungeKutta (const M& model_,
165                 DenseMatrix<number_type> A_,
166                 Vector<number_type> b_,
167                 Vector<number_type> c_)
168     : model(model_), u(model.size()), w(model.size()), K(A_.rowsize ())
169     {
170         A = A_;
171         b = b_;
172         c = c_;
173         s = A_.rowsize ();
174         n = model.size();
175         model.initialize(t,u);
176         dt = 0.1;
177         for (int i = 0; i < s; i++)
178         {
179             K[i].resize(n, number_type(0));
180         }
181         sigma = 0.01;
182         verbosity = 0;
183
184         if (A_.rowsize() != A_.colsize())
185             HDNUM_ERROR("need square and nonempty matrix");
186         if (A_.rowsize() != b_.size())
187             HDNUM_ERROR("vector incompatible with matrix");
188         if (A_.colsize() != c_.size())
189             HDNUM_ERROR("vector incompatible with matrix");
190

```

```

196     }
197
198     RungeKutta (const M& model_,
199                 DenseMatrix<number_type> A_,
200                 Vector<number_type> b_,
201                 Vector<number_type> c_,
202                 number_type sigma_)
203     : model(model_), u(model.size()), w(model.size()), K(A_.rowsize ())
204     {
205         A = A_;
206         b = b_;
207         c = c_;
208         s = A_.rowsize ();
209         n = model.size();
210         model.initialize(t,u);
211         dt = 0.1;
212         for (int i = 0; i < s; i++)
213         {
214             K[i].resize(n, number_type(0));
215         }
216         sigma = sigma_;
217         verbosity = 0;
218         if (A_.rowsize ()!=A_.colsize())
219             HDNUM_ERROR("need square and nonempty matrix");
220         if (A_.rowsize ()!=b_.size())
221             HDNUM_ERROR("vector incompatible with matrix");
222         if (A_.colsize ()!=c_.size())
223             HDNUM_ERROR("vector incompatible with matrix");
224     }
225
226     void set_dt (time_type dt_)
227     {
228         dt = dt_;
229     }
230
231     bool check_explicit ()
232     {
233         bool is_explicit = true;
234         for (int i = 0; i < s; i++)
235         {
236             for (int j = i; j < s; j++)
237             {
238                 if (A[i][j] != 0.0)
239                 {
240                     is_explicit = false;
241                 }
242             }
243         }
244         return is_explicit;
245     }
246
247     void step ()
248     {
249         if (check_explicit())
250         {
251             // compute k_1
252             w = u;
253             model.f(t, w, K[0]);
254             for (int i = 0; i < s; i++)
255             {
256                 Vector<number_type> sum (K[0].size(), 0.0);
257                 sum.update(b[0], K[0]);
258                 //compute k_i
259                 for (int j = 0; j < i+1; j++)
260                 {
261                     sum.update(A[i][j],K[j]);
262                 }
263                 Vector<number_type> wert = w.update(dt,sum);
264                 model.f(t + c[i]*dt, wert, K[i]);
265                 u.update(dt *b[i], K[i]);
266             }
267         }
268         if (not check_explicit())
269         {
270             // In the implicit case we need to solve a nonlinear problem
271             // to do a time step.
272             ImplicitRungeKuttaStepProblem<M> problem(model, A, b, c, t, u, dt);
273             bool last_row_eq_b = true;
274             for (int i = 0; i<s; i++)
275             {
276                 if (A[s-1][i] != b[i])
277                 {
278                     last_row_eq_b = false;
279                 }
280             }
281             // Solve nonlinear problem and determine coefficients

```

```

287     S solver;
288     solver.set_maxit(2000);
289     solver.set_verbosity(verbosity);
290     solver.set_reduction(1e-10);
291     solver.set_abslimit(1e-10);
292     solver.set_linesearchsteps(10);
293     solver.set_sigma(0.01);
294     Vector<number_type> zij (s*n,0.0);
295     solver.solve(problem,zij);
296
297
298     DenseMatrix<number_type> Ainv (s,s,number_type(0));
299     if (not last_row_eq_b)
300     {
301         // Compute LR decomposition of A
302         Vector<number_type> w (s, number_type(0));
303         Vector<number_type> x (s, number_type(0));
304         Vector<number_type> z (s, number_type(0));
305         Vector<std::size_t> p(s);
306         Vector<std::size_t> q(s);
307         DenseMatrix<number_type> Temp (s,s,0.0);
308         Temp = A;
309         row_equilibrate(Temp,w);
310         lr_fullpivot(Temp,p,q);
311
312         // Use LR decomposition to calculate inverse of A
313         for (int i=0; i<s; i++)
314         {
315             Vector<number_type> e (s, number_type(0));
316             e[i]=number_type(1);
317             apply_equilibrate(w,e);
318             permute_forward(p,e);
319             solveL(Temp,e,e);
320             solveR(Temp,z,e);
321             permute_backward(q,z);
322             for (int j = 0; j < s; j++)
323             {
324                 Ainv[j][i] = z[j];
325             }
326         }
327     }
328
329     Vector<Vector<number_type> > Z (s, 0.0);
330     for(int i=0; i<s; i++)
331     {
332         Vector<number_type> zero(n,number_type(0));
333         Z[i] = zero;
334         for (int j = 0; j < n; j++)
335         {
336             Z[i][j] = zij[i*n+j];
337         }
338     }
339     if (last_row_eq_b)
340     {
341         u += Z[s-1];
342     }
343     else
344     {
345         // compute ki
346         Vector<number_type> zero(n,number_type(0));
347         for (int i = 0; i < s; i++)
348         {
349             K[i] = zero;
350             for (int j=0; j < s; j++)
351             {
352                 K[i].update(Ainv[i][j],Z[j]);
353             }
354             K[i]*= (1.0/dt);
355
356             // compute u
357             u.update(dt*b[i], K[i]);
358         }
359     }
360 }
361 t = t+dt;
362 }
363
364 void set_state (time_type t_, const Vector<number_type>& u_)
365 {
366     t = t_;
367     u = u_;
368 }
369
370
371 const Vector<number_type>& get_state () const
372 {
373     return u;
374 }
375

```

```

376
377 time_type get_time () const
378 {
379     return t;
380 }
381
382 time_type get_dt () const
383 {
384     return dt;
385 }
386
387 void set_verbosity(int verbosity_)
388 {
389     verbosity = verbosity_;
390 }
391
392 private:
393     const M& model;
394     time_type t, dt;
395     Vector<number_type> u;
396     Vector<number_type> w;
397     Vector<Vector<number_type> > K; // save ki
398     int n; //
399     dimension of matrix A
400     int s;
401     DenseMatrix<number_type> A; // A, b, c as in the butcher tableau
402     Vector<number_type> b;
403     Vector<number_type> c;
404     number_type sigma;
405     int verbosity;
406 };
407
408 template<class M, class S>
409 void ordertest(const M& model,
410               S solver,
411               typename M::number_type T,
412               typename M::number_type h_0,
413               int l)
414 {
415     // Get types
416     typedef typename M::time_type time_type;
417     typedef typename M::number_type number_type;
418
419     // error_array[i] = ||u(T)-u_i(T)||
420     number_type error_array[l];
421
422     Vector<number_type> exact_solution;
423     model.exact_solution(T, exact_solution);
424
425     for (int i=0; i<l; i++)
426     {
427         // Set initial time and value
428         time_type t_start;
429         Vector<number_type> initial_solution(l);
430         model.initialize(t_start, initial_solution);
431         solver.set_state(t_start, initial_solution);
432
433         // Initial time step
434         time_type dt = h_0/pow(2,i) ;
435         solver.set_dt(dt);
436
437         // Time loop
438         while (solver.get_time()<T-2*solver.get_dt())
439         {
440             solver.step();
441         }
442
443         // Last steps
444         if (solver.get_time()<T-solver.get_dt())
445         {
446             solver.set_dt((T-solver.get_time())/2.0);
447             for(int i=0; i<2; i++)
448             {
449                 solver.step();
450             }
451         }
452         else
453         {
454             solver.set_dt(T-solver.get_time());
455             solver.step();
456         }
457
458         // Error
459         Vector<number_type> state = solver.get_state();
460         error_array[i] = norm(exact_solution-state);
461     }
462 }

```

```

476         if(i==0)
477         {
478             std::cout << "dt: "
479                       << std::scientific << std::showpoint << std::setprecision(8)
480                       << dt
481                       << " "
482                       << "Error: "
483                       << error_array[0] << std::endl;
484         }
485         if(i>0)
486         {
487             number_type rate = log(error_array[i-1]/error_array[i])/log(2);
488             std::cout << "dt: "
489                       << std::scientific << std::showpoint << std::setprecision(8)
490                       << dt
491                       << " "
492                       << "Error: "
493                       << error_array[i]
494                       << " "
495                       << "Rate: "
496                       << rate << std::endl;
497         }
498     }
499 }
500
501 } // namespace hdnun
502
503 #endif

```

## 5.22 sgrid.hh

```

1  #ifndef HDNUM_SGRID_HH
2  #define HDNUM_SGRID_HH
3  #include <limits>
4  #include <assert.h>
5
6  namespace hdnun {
13     template<class N, class DF, int dimension>
14     class SGrid
15     {
16     public:
17
18         typedef std::size_t size_type;
19
20         typedef N number_type;
21
22         typedef DF DomainFunction;
23
24         enum { dim = dimension };
25
26         static const int positive = 1;
27         static const int negative = -1;
28
29     private:
30
31         const Vector<number_type> extent;
32         const Vector<size_type> size;
33         const DomainFunction & df;
34         Vector<number_type> h;
35         Vector<size_type> offsets;
36         std::vector<size_type> node_map;
37         std::vector<size_type> grid_map;
38         std::vector<bool> inside_map;
39         std::vector<bool> boundary_map;
40
41         size_t n_nodes;
42
43         inline Vector<size_type> index2grid(size_type index) const
44         {
45             Vector<size_type> c(dim);
46             for(int d=dim-1; d>=0; --d){
47                 c[d] = index / offsets[d];
48                 index -= c[d] * offsets[d];
49             }
50             return c;
51         }
52
53         inline Vector<number_type> grid2world(const Vector<size_type> & c) const
54         {
55             Vector<number_type> w(dim);
56             for(int d=dim-1; d>=0; --d)
57                 w[d] = c[d] * h[d];
58         }
59     };
60
61 }
62

```

```

63     return w;
64 }
65
66 inline Vector<number_type> index2world(size_type index) const
67 {
68     Vector<number_type> w(dim);
69     Vector<size_type> c = index2grid(index);
70     return grid2world(c);
71 }
72
73
74 public:
75
76     const size_type invalid_node;
77
78     SGrid(const Vector<number_type> extent_,
79           const Vector<size_type> size_,
80           const DomainFunction & df_)
81     : extent(extent_), size(size_), df(df_),
82       h(dim), offsets(dim),
83       invalid_node(std::numeric_limits<size_type>::max())
84     {
85         // Determine total number of nodes, increment offsets, and cell
86         // widths.
87         n_nodes = 1;
88         offsets.resize(dim);
89         h.resize(dim);
90         for(int d=0; d<dim; ++d){
91             n_nodes *= size[d];
92             offsets[d] = d==0 ? 1 : size[d-1] * offsets[d-1];
93             h[d] = extent[d] / number_type(size[d]-1);
94         }
95
96         // Initialize maps.
97         node_map.resize(0);
98         inside_map.resize(n_nodes);
99         grid_map.resize(n_nodes);
100        boundary_map.resize(0);
101        boundary_map.resize(n_nodes, false);
102
103        for(size_type n=0; n<n_nodes; ++n){
104            Vector<size_type> c = index2grid(n);
105            Vector<number_type> x = grid2world(c);
106
107            inside_map[n] = df.evaluate(x);
108            if(inside_map[n]){
109                node_map.push_back(n);
110                grid_map[n] = node_map.size()-1;
111            }
112            else
113                grid_map[n] = invalid_node;
114        }
115
116        // Find boundary nodes
117        for(size_type n=0; n<node_map.size(); ++n){
118            for(int d=0; d<dim; ++d){
119                for(int s=0; s<2; ++s){
120                    const int side = s*2-1;
121                    const size_type neighbor = getNeighborIndex(n,d,side,1);
122                    if(neighbor == invalid_node)
123                        boundary_map[node_map[n]] = true;
124                }
125            }
126        }
127    }
128
129    size_type getNeighborIndex(const size_type ln, const size_type n_dim, const int n_side, const int k
130    = 1) const
131    {
132        const size_type n = node_map[ln];
133        const Vector<size_type> c = index2grid(n);
134        size_type neighbors[2];
135        neighbors[0] = c[n_dim];
136        neighbors[1] = size[n_dim]-c[n_dim]-1;
137
138        assert(n_side == 1 || n_side == -1);
139        if(size_type(k) > neighbors[(n_side+1)/2])
140            return invalid_node;
141
142        const size_type neighbor = n + offsets[n_dim] * n_side * k;
143
144        if(!inside_map[neighbor])
145            return invalid_node;
146
147        return grid_map[neighbor];
148    }

```

```

183
187     bool isBoundaryNode(const size_type ln) const
188     {
189         return boundary_map[node_map[ln]];
190     }
191
195     size_type getNumberOfNodes() const
196     {
197         return node_map.size();
198     }
199
200     Vector<size_type> getGridSize() const
201     {
202         return size;
203     }
204
207     Vector<number_type> getCellWidth() const
208     {
209         return h;
210     }
211
215     Vector<number_type> getCoordinates(const size_type ln) const
216     {
217         return index2world(node_map[ln]);
218     }
219
220     std::vector<Vector<number_type> > getNodeCoordinates() const
221     {
222         std::vector<Vector<number_type> > coords;
223         for(size_type n=0; n<node_map.size(); ++n){
224             coords.push_back(Vector<number_type>(dim));
225             coords.back() = index2world(node_map[n]);
226         }
227         return coords;
228     }
229
230 };
231
232 }
233
234 #endif // HDNUM_SGRID_HH

```

## 5.23 sparsematrix.hh

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 /*
3  * File:    sparsematrix.hh
4  * Author:  Christian Heusel <christian@heusel.eu>
5  *
6  * Created on August 25, 2020
7  */
8
9 #ifndef SPARSEMATRIX_HH
10 #define SPARSEMATRIX_HH
11
12 #include <algorithm>
13 #include <complex>
14 #include <functional>
15 #include <iomanip>
16 #include <iostream>
17 #include <map>
18 #include <numeric>
19 #include <string>
20 #include <type_traits>
21 #include <vector>
22
23 #include "densematrix.hh"
24 #include "vector.hh"
25
26 namespace hdnum {
27
28 template <typename REAL>
29 class SparseMatrix {
30 public:
31     using size_type = std::size_t;
32
33     using column_iterator = typename std::vector<REAL>::iterator;
34     using const_column_iterator = typename std::vector<REAL>::const_iterator;
35
36 private:
37     // Matrix data is stored in an STL vector!
38     std::vector<REAL> _data;
39
40 };
41
42 #endif

```

```

45 // The non-null indices are stored in STL vectors with the size_type!
46 // Explanation on how the mapping works can be found here:
47 // https://de.wikipedia.org/wiki/Compressed_Row_Storage
48 std::vector<size_type> _colIndices;
49 std::vector<size_type> _rowPtr;
50
51 size_type m_rows = 0; // Number of Matrix rows
52 size_type m_cols = 0; // Number of Matrix columns
53
54 static bool bScientific;
55 static size_type nIndexWidth;
56 static size_type nValueWidth;
57 static size_type nValuePrecision;
58 static const REAL _zero;
59
60 // !function that converts container contents into
61 // { 1, 2, 3, 4 }
62 template <typename T>
63 [[nodiscard]] std::string comma_fold(T container) const {
64     return "{ " +
65         std::accumulate(
66             std::next(container.cbegin()), container.cend(),
67             std::to_string(container[0]), // start with first element
68             [](const std::string &a, REAL b) {
69                 return a + ", " + std::to_string(b);
70             }) +
71         " }";
72 }
73
74 // This code was copied from StackOverflow to gerneralize a check whether a
75 // template is a specialization i.e. for std::complex
76 // https://stackoverflow.com/questions/31762958/check-if-class-is-a-template-specialization
77 template <class T, template <class...> class Template>
78 struct is_specialization : std::false_type {};
79
80 template <template <class...> class Template, class... Args>
81 struct is_specialization<Template<Args...>, Template> : std::true_type {};
82
83 bool checkIfAccessIsInBounds(const size_type row_index,
84                             const size_type col_index) const {
85     if (not (row_index < m_rows)) {
86         HDNUM_ERROR("Out of bounds access: row too big! -> " +
87             std::to_string(row_index) + " is not < " +
88             std::to_string(m_rows));
89         return false;
90     }
91     if (not (col_index < m_cols)) {
92         HDNUM_ERROR("Out of bounds access: column too big! -> " +
93             std::to_string(col_index) + " is not < " +
94             std::to_string(m_cols));
95         return false;
96     }
97     return true;
98 }
99
100 public:
101     SparseMatrix() = default;
102
103     SparseMatrix(const size_type _rows, const size_type _cols)
104         : _rowPtr(_rows + 1), m_rows(_rows), m_cols(_cols) {}
105
106     [[nodiscard]] size_type rowsize() const { return m_rows; }
107
108     [[nodiscard]] size_type colsize() const { return m_cols; }
109
110     [[nodiscard]] bool scientific() const { return bScientific; }
111
112     class column_index_iterator {
113     public:
114         using self_type = column_index_iterator;
115
116         // conform to the iterator traits
117         // https://en.cppreference.com/w/cpp/iterator/iterator_traits
118         using difference_type = std::ptrdiff_t;
119         using value_type = std::pair<REAL &, size_type const &>;
120         using pointer = value_type *;
121         using reference = value_type &;
122         using iterator_category = std::bidirectional_iterator_tag;
123
124         column_index_iterator(typename std::vector<REAL>::iterator valIter,
125                             std::vector<size_type>::iterator colIndicesIter)
126             : _valIter(valIter), _colIndicesIter(colIndicesIter) {}
127
128         // prefix
129         self_type &operator++() {
130             _valIter++;
131             _colIndicesIter++;
132         }

```



```

180         return *this;
181     }
182
183     // postfix
184     self_type &operator++(int junk) {
185         self_type cached = *this;
186         _valIter++;
187         _colIndicesIter++;
188         return cached;
189     }
190
191     [[nodiscard]] value_type operator*() {
192         return std::make_pair(std::ref(*_valIter),
193                               std::cref(*_colIndicesIter));
194     }
195     // [[nodiscard]] value_type operator->() {
196     //     return std::make_pair(std::ref(*_valIter),
197     //                           std::cref(*_colIndicesIter));
198     // }
199
200     [[nodiscard]] typename value_type::first_type value() {
201         return std::ref(*_valIter);
202     }
203
204     [[nodiscard]] typename value_type::second_type index() {
205         return std::cref(*_colIndicesIter);
206     }
207
208     [[nodiscard]] bool operator==(const self_type &other) {
209         return (_valIter == other._valIter) and
210                (_colIndicesIter == other._colIndicesIter);
211     }
212     [[nodiscard]] bool operator!=(const self_type &other) {
213         return not (*this == other);
214     }
215
216 private:
217     typename std::vector<REAL>::iterator _valIter;
218     std::vector<size_type>::iterator _colIndicesIter;
219 };
220
221 class const_column_index_iterator {
222 public:
223     using self_type = const_column_index_iterator;
224
225     // conform to the iterator traits
226     // https://en.cppreference.com/w/cpp/iterator/iterator_traits
227     using difference_type = std::ptrdiff_t;
228     using value_type = std::pair<REAL const &, size_type const &>;
229     using pointer = value_type *;
230     using reference = value_type &;
231     using iterator_category = std::bidirectional_iterator_tag;
232
233     const_column_index_iterator(
234         typename std::vector<REAL>::const_iterator valIter,
235         std::vector<size_type>::const_iterator colIndicesIter)
236         : _valIter(valIter), _colIndicesIter(colIndicesIter) {}
237
238     // prefix
239     self_type &operator++() {
240         _valIter++;
241         _colIndicesIter++;
242         return *this;
243     }
244
245     // postfix
246     self_type operator++(int junk) {
247         self_type cached = *this;
248         _valIter++;
249         _colIndicesIter++;
250         return cached;
251     }
252
253     [[nodiscard]] value_type operator*() {
254         return std::make_pair(std::ref(*_valIter),
255                               std::cref(*_colIndicesIter));
256     }
257     // TODO: This is wrong
258     // [[nodiscard]] value_type operator->() {
259     //     return std::make_pair(*_valIter, *_colIndicesIter);
260     // }
261
262     [[nodiscard]] typename value_type::first_type value() {
263         return std::ref(*_valIter);
264     }
265
266     [[nodiscard]] typename value_type::second_type index() {

```

```

267         return std::cref(*_colIndicesIter);
268     }
269
270     [[nodiscard]] bool operator==(const self_type &other) {
271         return (_valIter == other._valIter) and
272             (_colIndicesIter == other._colIndicesIter);
273     }
274     [[nodiscard]] bool operator!=(const self_type &other) {
275         return not (*this == other);
276     }
277
278 private:
279     typename std::vector<REAL>::const_iterator _valIter;
280     std::vector<size_type>::const_iterator _colIndicesIter;
281 };
282
283 class row_iterator {
284 public:
285     using self_type = row_iterator;
286
287     // conform to the iterator traits
288     // https://en.cppreference.com/w/cpp/iterator/iterator_traits
289     using difference_type = std::ptrdiff_t;
290     using value_type = self_type;
291     using pointer = self_type*;
292     using reference = self_type&;
293     using iterator_category = std::random_access_iterator_tag;
294
295     row_iterator(std::vector<size_type>::iterator rowPtrIter,
296                 std::vector<size_type>::iterator colIndicesIter,
297                 typename std::vector<REAL>::iterator valIter)
298         : _rowPtrIter(rowPtrIter), _colIndicesIter(colIndicesIter),
299           _valIter(valIter) {}
300
301     [[nodiscard]] column_iterator begin() {
302         return column_iterator(_valIter + *_rowPtrIter);
303     }
304     [[nodiscard]] column_iterator end() {
305         return column_iterator(_valIter + *(_rowPtrIter + 1));
306     }
307
308     [[nodiscard]] column_index_iterator ibegin() {
309         return column_index_iterator(_valIter + *_rowPtrIter,
310                                     (_colIndicesIter + *_rowPtrIter));
311     }
312     [[nodiscard]] column_index_iterator iend() {
313         return column_index_iterator(
314             _valIter + *(_rowPtrIter + 1),
315             (_colIndicesIter + *(_rowPtrIter + 1)));
316     }
317
318     // prefix
319     self_type &operator++() {
320         _rowPtrIter++;
321         return *this;
322     }
323
324     // postfix
325     self_type operator++(int junk) {
326         self_type cached = *this;
327         _rowPtrIter++;
328         return cached;
329     }
330
331     self_type &operator+=(difference_type offset) {
332         _rowPtrIter += offset;
333         return *this;
334     }
335
336     self_type &operator-=(difference_type offset) {
337         _rowPtrIter -= offset;
338         return *this;
339     }
340
341     // iter - n
342     self_type operator-(difference_type offset) {
343         self_type cache(*this);
344         cache -= offset;
345         return cache;
346     }
347
348     // iter + n
349     self_type operator+(difference_type offset) {
350         self_type cache(*this);
351         cache += offset;
352         return cache;
353     }

```

```

354         // n + iter
355     friend self_type operator+(const difference_type &offset,
356                               const self_type &sec) {
357         self_type cache(sec);
358         cache += offset;
359         return cache;
360     }
361
362     reference operator[](difference_type offset) {
363         return *(*this + offset);
364     }
365
366     bool operator<(const self_type &other) {
367         return other - (*this) > 0; //
368     }
369
370     bool operator>(const self_type &other) {
371         return other < (*this); //
372     }
373
374     [[nodiscard]] self_type &operator*() { return *this; }
375     // [[nodiscard]] self_type &operator->() { return *this; }
376
377     [[nodiscard]] bool operator==(const self_type &rhs) {
378         return _rowPtrIter == rhs._rowPtrIter;
379     }
380     [[nodiscard]] bool operator!=(const self_type &rhs) {
381         return _rowPtrIter != rhs._rowPtrIter;
382     }
383
384 private:
385     std::vector<size_type>::iterator _rowPtrIter;
386     std::vector<size_type>::iterator _colIndicesIter;
387     typename std::vector<REAL>::iterator _valIter;
388 };
389
390 class const_row_iterator {
391 public:
392     using self_type = const_row_iterator;
393
394     // conform to the iterator traits
395     // https://en.cppreference.com/w/cpp/iterator/iterator_traits
396     using difference_type = std::ptrdiff_t;
397     using value_type = self_type;
398     using pointer = self_type *;
399     using reference = self_type &;
400     using iterator_category = std::bidirectional_iterator_tag;
401
402     const_row_iterator(
403         std::vector<size_type>::const_iterator rowPtrIter,
404         std::vector<size_type>::const_iterator colIndicesIter,
405         typename std::vector<REAL>::const_iterator valIter)
406         : _rowPtrIter(rowPtrIter), _colIndicesIter(colIndicesIter),
407           _valIter(valIter) {}
408
409     [[nodiscard]] const_column_iterator begin() const {
410         return const_column_iterator((_valIter + *_rowPtrIter));
411     }
412     [[nodiscard]] const_column_iterator end() const {
413         return const_column_iterator((_valIter + *(_rowPtrIter + 1)));
414     }
415
416     [[nodiscard]] const_column_index_iterator ibegin() const {
417         return const_column_index_iterator(
418             (_valIter + *_rowPtrIter), (_colIndicesIter + *_rowPtrIter));
419     }
420     [[nodiscard]] const_column_index_iterator iend() const {
421         return const_column_index_iterator(
422             (_valIter + *(_rowPtrIter + 1)),
423             (_colIndicesIter + *(_rowPtrIter + 1)));
424     }
425
426     [[nodiscard]] const_column_iterator cbegin() const {
427         return this->begin();
428     }
429     [[nodiscard]] const_column_iterator cend() const {
430         return this->end(); //
431     }
432
433     // prefix
434     self_type &operator++() {
435         _rowPtrIter++;
436         return *this;
437     }
438
439     // postfix
440     self_type &operator++(int junk) {

```

```

441         self_type cached = *this;
442         _rowPtrIter++;
443         return cached;
444     }
445
446     self_type &operator+=(difference_type offset) {
447         _rowPtrIter += offset;
448         return *this;
449     }
450
451     self_type &operator-=(difference_type offset) {
452         _rowPtrIter -= offset;
453         return *this;
454     }
455
456     // iter - n
457     self_type operator-(difference_type offset) {
458         self_type cache(*this);
459         cache -= offset;
460         return cache;
461     }
462
463     // iter + n
464     self_type operator+(difference_type offset) {
465         self_type cache(*this);
466         cache += offset;
467         return cache;
468     }
469
470     // n + iter
471     friend self_type operator+(const difference_type &offset,
472                                const self_type &sec) {
473         self_type cache(sec);
474         cache += offset;
475         return cache;
476     }
477
478     reference operator[](difference_type offset) {
479         return *(*this + offset);
480     }
481
482     bool operator<(const self_type &other) {
483         return other - (*this) > 0; //
484     }
485
486     bool operator>(const self_type &other) {
487         return other < (*this); //
488     }
489
490     [[nodiscard]] self_type &operator*() { return *this; }
491     // [[nodiscard]] self_type &operator->() { return this; }
492
493     [[nodiscard]] bool operator==(const self_type &rhs) {
494         return _rowPtrIter == rhs._rowPtrIter;
495     }
496     [[nodiscard]] bool operator!=(const self_type &rhs) {
497         return _rowPtrIter != rhs._rowPtrIter;
498     }
499 private:
500     std::vector<size_type>::const_iterator _rowPtrIter;
501     std::vector<size_type>::const_iterator _colIndicesIter;
502     typename std::vector<REAL>::const_iterator _valIter;
503 };
504
505 [[nodiscard]] row_iterator begin() {
506     return row_iterator(_rowPtr.begin(), _colIndices.begin(),
507                        _data.begin());
508 }
509
510 [[nodiscard]] row_iterator end() {
511     return row_iterator(_rowPtr.end() - 1, _colIndices.begin(),
512                        _data.begin());
513 }
514
515 [[nodiscard]] const_row_iterator cbegin() const {
516     return const_row_iterator(_rowPtr.cbegin(), _colIndices.cbegin(),
517                              _data.cbegin());
518 }
519
520 [[nodiscard]] const_row_iterator cend() const {
521     return const_row_iterator(_rowPtr.cend() - 1, _colIndices.cbegin(),
522                              _data.cbegin());
523 }
524
525 [[nodiscard]] const_row_iterator begin() const { return this->cbegin(); }
526 [[nodiscard]] const_row_iterator end() const { return this->cend(); }
527

```

```

600 void scientific(bool b) const { bScientific = b; }
601
602 size_type iwidth() const { return nIndexWidth; }
603
604 size_type width() const { return nValueWidth; }
605
606 size_type precision() const { return nValuePrecision; }
607
608 void iwidth(size_type i) const { nIndexWidth = i; }
609
610 void width(size_type i) const { nValueWidth = i; }
611
612 void precision(size_type i) const { nValuePrecision = i; }
613
614 column_iterator find(const size_type row_index,
615                     const size_type col_index) const {
616     checkIfAccessIsInBounds(row_index, col_index);
617
618     using value_pair = typename const_column_index_iterator::value_type;
619     auto row = const_row_iterator(_rowPtr.begin() + row_index,
620                                   _colIndices.begin(), _data.begin());
621     return std::find_if(row.ibegin(), row.iend(),
622                         [col_index](value_pair el) {
623                             // only care for the index here since the value
624                             // is unknown
625                             return el.second == col_index;
626                         });
627 }
628
629 bool exists(const size_type row_index, const size_type col_index) const {
630     auto row = const_row_iterator(_rowPtr.begin() + row_index,
631                                   _colIndices.begin(), _data.begin());
632     return find(row_index, col_index) != row.iend();
633 }
634
635 REAL &get(const size_type row_index, const size_type col_index) {
636     checkIfAccessIsInBounds(row_index, col_index);
637     // look for the entry
638     using value_pair = typename const_column_index_iterator::value_type;
639     auto row = row_iterator(_rowPtr.begin() + row_index,
640                              _colIndices.begin(), _data.begin());
641     auto result =
642         std::find_if(row.ibegin(), row.iend(), [col_index](value_pair el) {
643             // only care for the index here
644             // since the value is unknown
645             // anyways
646             return el.second == col_index;
647         });
648     // we found something within the right row
649     if (result != row.iend()) {
650         return result.value();
651     }
652     throw std::out_of_range(
653         "There is no non-zero element for these given indicies!");
654 }
655
656 const REAL &operator()(const size_type row_index,
657                        const size_type col_index) const {
658     checkIfAccessIsInBounds(row_index, col_index);
659
660     using value_pair = typename const_column_index_iterator::value_type;
661     auto row = const_row_iterator(_rowPtr.begin() + row_index,
662                                   _colIndices.begin(), _data.begin());
663     auto result =
664         std::find_if(row.ibegin(), row.iend(), [col_index](value_pair el) {
665             // only care for the index here since the value is unknown
666             return el.second == col_index;
667         });
668     // we found something within the right row
669     if (result != row.iend()) {
670         return result.value();
671     }
672     return _zero;
673 }
674
675 [[nodiscard]] bool operator==(const SparseMatrix &other) const {
676     return (_data == other._data) and //
677            (_rowPtr == other._rowPtr) and //
678            (_colIndices == other._colIndices) and //
679            (m_cols == other.m_cols) and //
680            (m_rows == other.m_rows);
681 }
682
683 [[nodiscard]] bool operator!=(const SparseMatrix &other) const {
684     return not (*this == other);
685 }
686

```

```

697 // delete all the invalid comparisons
698 bool operator<(const SparseMatrix &other) = delete;
699 bool operator>(const SparseMatrix &other) = delete;
700 bool operator<=(const SparseMatrix &other) = delete;
701 bool operator>=(const SparseMatrix &other) = delete;
702
703 SparseMatrix transpose() const {
704     // TODO: remove / find bug here!
705     SparseMatrix::builder builder(m_cols, m_rows);
706     SparseMatrix::size_type curr_row = 0;
707     for (auto &row : (*this)) {
708         for (auto it = row.ibegin(); it != row.iend(); it++) {
709             builder.addEntry(it.index(), curr_row, it.value());
710         }
711         curr_row++;
712     }
713
714     return builder.build();
715 }
716
717 [[nodiscard]] SparseMatrix operator*=(const REAL scalar) {
718     // This could also be done out of order
719     std::transform(_data.cbegin(), _data.cend(), _data.begin(),
720         [&](REAL value) { return value * scalar; });
721 }
722
723 [[nodiscard]] SparseMatrix operator/=(const REAL scalar) {
724     // This could also be done out of order
725     std::transform(_data.cbegin(), _data.cend(), _data.begin(),
726         [&](REAL value) { return value / scalar; });
727 }
728
729 template <class V>
730 void mv(Vector<V> &result, const Vector<V> &x) const {
731     static_assert(std::is_convertible<V, REAL>::value,
732         "The types in the Matrix vector multiplication cant be "
733         "converted properly!");
734
735     if (result.size() != this->colsize()) {
736         HDNUM_ERROR(
737             (std::string("The result vector has the wrong dimension! ") +
738             "Vector dimension " + std::to_string(result.size()) +
739             " != " + std::to_string(this->colsize()) + " colsize"));
740     }
741
742     if (x.size() != this->colsize()) {
743         HDNUM_ERROR(
744             (std::string("The input vector has the wrong dimension! ") +
745             "Vector dimension " + std::to_string(x.size()) +
746             " != " + std::to_string(this->colsize()) + " colsize"));
747     }
748
749     size_type curr_row = 0;
750     for (auto row : (*this)) {
751         result[curr_row] = std::accumulate(
752             row.ibegin(), row.iend(), V {}, [&](V result, auto el) -> V {
753                 return result + (x[el.second] * el.first);
754             });
755         curr_row++;
756     }
757 }
758
759 [[nodiscard]] Vector<REAL> operator*(const Vector<REAL> &x) const {
760     hdnum::Vector<REAL> result(this->colsize(), 0);
761     this->mv(result, x);
762     return result;
763 }
764
765 template <class V>
766 void umv(Vector<V> &result, const Vector<V> &x) const {
767     static_assert(std::is_convertible<V, REAL>::value,
768         "The types in the Matrix vector multiplication cant be "
769         "converted properly!");
770
771     if (result.size() != this->colsize()) {
772         HDNUM_ERROR(
773             (std::string("The result vector has the wrong dimension! ") +
774             "Vector dimension " + std::to_string(result.size()) +
775             " != " + std::to_string(this->colsize()) + " colsize"));
776     }
777
778     if (x.size() != this->colsize()) {
779         HDNUM_ERROR(
780             (std::string("The input vector has the wrong dimension! ") +
781             "Vector dimension " + std::to_string(result.size()) +
782             " != " + std::to_string(this->colsize()) + " colsize"));
783     }
784 }

```

```

811
812     size_type curr_row {};
813     for (auto row : (*this)) {
814         result[curr_row] += std::accumulate(
815             row.begin(), row.iend(), V {}, [&](V result, auto el) -> V {
816                 return result + (x[el.second] * el.first);
817             });
818         curr_row++;
819     }
820 }
821
822 private:
823 template <typename norm_type>
824 norm_type norm_infty_impl() const {
825     norm_type norm {};
826     for (auto row : *this) {
827         norm_type rowsum =
828             std::accumulate(row.begin(), row.end(), norm_type {},
829                             [](norm_type res, REAL value) -> norm_type {
830                                 return res + std::abs(value);
831                             });
832         if (norm < rowsum) {
833             norm = rowsum;
834         }
835     }
836     return norm;
837 }
838
839 public:
840 auto norm_infty() const {
841     if constexpr (is_specialization<REAL, std::complex> {}) {
842         return norm_infty_impl<double>();
843     } else {
844         return norm_infty_impl<REAL>();
845     }
846 }
847
848 [[nodiscard]] std::string to_string() const noexcept {
849     return "values=" + comma_fold(_data) + "\n" + //
850         "colInd=" + comma_fold(_colIndices) + "\n" + //
851         "rowPtr=" + comma_fold(_rowPtr) + "\n"; //
852 }
853
854 void print() const noexcept { std::cout << *this; }
855
856 static SparseMatrix identity(const size_type dimN) {
857     auto builder = typename SparseMatrix<REAL>::builder(dimN, dimN);
858     for (typename SparseMatrix<REAL>::size_type i = 0; i < dimN; ++i) {
859         builder.addEntry(i, i, REAL {1});
860     }
861     return builder.build();
862 }
863
864 SparseMatrix<REAL> matchingIdentity() const { return identity(m_cols); }
865
866 class builder {
867     size_type m_rows {}; // Number of Matrix rows, 0 by default
868     size_type m_cols {}; // Number of Matrix columns, 0 by default
869     std::vector<std::map<size_type, REAL> _rows;
870
871 public:
872     builder(size_type new_m_rows, size_type new_m_cols)
873         : m_rows {new_m_rows}, m_cols {new_m_cols}, _rows {m_rows} {}
874
875     builder(const std::initializer_list<std::initializer_list<REAL> &v)
876         : m_rows {v.size()}, m_cols {v.begin()->size()}, _rows(m_rows) {
877         size_type i = 0;
878         for (auto &row : v) {
879             size_type j = 0;
880             for (const REAL &element : row) {
881                 addEntry(i, j, element);
882                 j++;
883             }
884             i++;
885         }
886     }
887
888     builder() = default;
889
890     std::pair<typename std::map<size_type, REAL>::iterator, bool> addEntry(
891         size_type i, size_type j, REAL value) {
892         return _rows.at(i).emplace(j, value);
893     }
894
895     std::pair<typename std::map<size_type, REAL>::iterator, bool> addEntry(
896         size_type i, size_type j) {
897         return addEntry(i, j, REAL {});
898     }
899 }

```

```

952     };
953
954     [[nodiscard]] bool operator==(
955         const SparseMatrix::builder &other) const {
956         return (m_rows == other.m_rows) and //
957             (m_cols == other.m_cols) and //
958             (_rows == other._rows);
959     }
960
961     [[nodiscard]] bool operator!=(
962         const SparseMatrix::builder &other) const {
963         return not (*this == other);
964     }
965
966     [[nodiscard]] size_type colsize() const noexcept { return m_cols; }
967     [[nodiscard]] size_type rowsize() const noexcept { return m_rows; }
968
969     size_type setNumCols(size_type new_m_cols) noexcept {
970         m_cols = new_m_cols;
971         return m_cols;
972     }
973     size_type setNumRows(size_type new_m_rows) {
974         m_rows = new_m_rows;
975         _rows.resize(m_cols);
976         return m_rows;
977     }
978
979     void clear() noexcept {
980         for (auto &row : _rows) {
981             row.clear();
982         }
983     }
984
985     [[nodiscard]] std::string to_string() const {
986         std::string output;
987         for (std::size_t i = 0; i < _rows.size(); i++) {
988             for (const auto &indexpair : _rows[i]) {
989                 output += std::to_string(i) + ", " +
990                     std::to_string(indexpair.first) + " => " +
991                     std::to_string(indexpair.second) + "\n";
992             }
993         }
994         return output;
995     }
996
997     [[nodiscard]] SparseMatrix build() {
998         auto result = SparseMatrix<REAL>(m_rows, m_cols);
999
1000         for (std::size_t i = 0; i < _rows.size(); i++) {
1001             result._rowPtr[i + 1] = result._rowPtr[i];
1002             for (const auto &indexpair : _rows[i]) {
1003                 result._colIndices.push_back(indexpair.first);
1004                 result._data.push_back(indexpair.second);
1005                 result._rowPtr[i + 1]++;
1006             }
1007         }
1008         return result;
1009     }
1010 };
1011 };
1012
1013 template <typename REAL>
1014 bool SparseMatrix<REAL>::bScientific = true;
1015 template <typename REAL>
1016 std::size_t SparseMatrix<REAL>::nIndexWidth = 10;
1017 template <typename REAL>
1018 std::size_t SparseMatrix<REAL>::nValueWidth = 10;
1019 template <typename REAL>
1020 std::size_t SparseMatrix<REAL>::nValuePrecision = 3;
1021 template <typename REAL>
1022 const REAL SparseMatrix<REAL>::_zero {};
1023
1024 template <typename REAL>
1025 std::ostream &operator<<(std::ostream &s, const SparseMatrix<REAL> &A) {
1026     using size_type = typename SparseMatrix<REAL>::size_type;
1027
1028     s << std::endl;
1029     s << " " << std::setw(A.iwidth()) << " "
1030     << " ";
1031     for (size_type j = 0; j < A.colsize(); ++j) {
1032         s << std::setw(A.width()) << j << " ";
1033     }
1034     s << std::endl;
1035
1036     for (size_type i = 0; i < A.rowsize(); ++i) {
1037         s << " " << std::setw(A.iwidth()) << i << " ";
1038         for (size_type j = 0; j < A.colsize(); ++j) {

```



```

1039         if (A.scientific()) {
1040             s « std::setw(A.width()) « std::scientific « std::showpoint
1041             « std::setprecision(A.precision()) « A(i, j) « " ";
1042         } else {
1043             s « std::setw(A.width()) « std::fixed « std::showpoint
1044             « std::setprecision(A.precision()) « A(i, j) « " ";
1045         }
1046     }
1047     s « std::endl;
1048 }
1049 return s;
1050 }
1051
1052 template <typename REAL>
1053 inline void zero(SparseMatrix<REAL> &A) {
1054     A = SparseMatrix<REAL>(A.rowsize(), A.colsize());
1055 }
1056
1057 template <class REAL>
1058 inline void identity(SparseMatrix<REAL> &A) {
1059     if (A.rowsize() != A.colsize()) {
1060         HDNUM_ERROR("Will not overwrite A since Dimensions are not equal!");
1061     }
1062     A = SparseMatrix<REAL>::identity(A.colsize());
1063 }
1064
1065 template <typename REAL>
1066 inline void readMatrixFromFile(const std::string &filename,
1067                               SparseMatrix<REAL> &A) {
1068     // Format taken from here:
1069     // https://math.nist.gov/MatrixMarket/formats.html#coord
1070
1071     using size_type = typename SparseMatrix<REAL>::size_type;
1072     std::string buffer;
1073     std::ifstream fin(filename);
1074     size_type i = 0;
1075     size_type j = 0;
1076     size_type non_zeros = 0;
1077
1078     if (fin.is_open()) {
1079         // ignore all comments from the file (starting with %)
1080         while (fin.peek() == '%') fin.ignore(2048, '\n');
1081
1082         std::getline(fin, buffer);
1083         std::istringstream first_line(buffer);
1084         first_line « i « j « non_zeros;
1085
1086         auto builder = typename SparseMatrix<REAL>::builder(i, j);
1087
1088         while (std::getline(fin, buffer)) {
1089             std::istringstream iss(buffer);
1090
1091             REAL value {};
1092             iss « i « j « value;
1093             // i-1, j-1, because matrix market does not use zero based indexing
1094             builder.addEntry(i - 1, j - 1, value);
1095         }
1096         A = builder.build();
1097         fin.close();
1098     } else {
1099         HDNUM_ERROR(("Could not osspen file! \"" + filename + "\""));
1100     }
1101 }
1102
1103 // namespace hdnum
1104
1105 #endif // SPARSEMATRIX_HH

```

## 5.24 src/timer.hh File Reference

A simple timing class.

```

#include <sys/resource.h>
#include <ctime>
#include <cstring>
#include <cerrno>
#include "exceptions.hh"

```

## Classes

- class `hdnum::TimerError`  
*Exception thrown by the `Timer` class*
- class `hdnum::Timer`  
*A simple stop watch.*

### 5.24.1 Detailed Description

A simple timing class.

## 5.25 timer.hh

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

```

1 #ifndef DUNE_TIMER_HH
2 #define DUNE_TIMER_HH
3
4 #ifndef TIMER_USE_STD_CLOCK
5 // headers for getrusage(2)
6 #include <sys/resource.h>
7 #endif
8
9 #include <ctime>
10
11 // headers for stderr(3)
12 #include <cstring>
13
14 // access to errno in C++
15 #include <cerrno>
16
17 #include "exceptions.hh"
18
19 namespace hdnum {
20
21     class TimerError : public SystemError {} ;
22
23     class Timer
24     {
25     public:
26         Timer ()
27         {
28             reset();
29         }
30
31         void reset()
32         {
33 #ifdef TIMER_USE_STD_CLOCK
34             cstart = std::clock();
35 #else
36             rusage ru;
37             if (getrusage(RUSAGE_SELF, &ru))
38                 HDNUM_THROW(TimerError, strerror(errno));
39             cstart = ru.ru_utime;
40 #endif
41         }
42
43         double elapsed () const
44         {
45 #ifdef TIMER_USE_STD_CLOCK
46             return (std::clock()-cstart) / static_cast<double>(CLOCKS_PER_SEC);
47 #else
48             rusage ru;
49             if (getrusage(RUSAGE_SELF, &ru))
50                 HDNUM_THROW(TimerError, strerror(errno));
51             return 1.0 * (ru.ru_utime.tv_sec - cstart.tv_sec) + (ru.ru_utime.tv_usec - cstart.tv_usec) /
52                 (1000.0 * 1000.0);
53 #endif
54         }
55
56     private:
57 #ifdef TIMER_USE_STD_CLOCK
58         std::clock_t cstart;
59 #endif
60     };
61
62 }
```

```

79 #else
80     struct timeval cstart;
81 #endif
82 }; // end class Timer
83
84 } // end namespace
85
86 #endif

```

## 5.26 vector.hh

```

1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 /*
3  * File:    vector.hh
4  * Author:  ngo
5  *
6  * Created on April 14th, 2011
7  */
8
9 #ifndef _VECTOR_HH
10 #define _VECTOR_HH
11
12 #include <assert.h>
13
14 #include <cmath>
15 #include <cstdlib>
16 #include <fstream>
17 #include <iomanip>
18 #include <iostream>
19 #include <sstream>
20 #include <vector>
21
22 #include "exceptions.hh"
23
24 namespace hdnum {
25
26     template<typename REAL>
27     class Vector : public std::vector<REAL> // inherit from the STL vector
28     {
29     public:
30         typedef std::size_t size_type;
31
32     private:
33         static bool bScientific;
34         static std::size_t nIndexWidth;
35         static std::size_t nValueWidth;
36         static std::size_t nValuePrecision;
37
38     public:
39
40         Vector() : std::vector<REAL>()
41         {
42         }
43
44         Vector( const size_t size, // user must specify the size
45                const REAL defaultValue_ = 0 // if not specified, the value 0 will take effect
46                )
47             : std::vector<REAL>( size, defaultValue_ )
48             {
49             }
50
51         Vector( const std::initializer_list<REAL> &v )
52         {
53             for (auto elem : v) this->push_back(elem);
54         }
55
56         // Methods:
57
58         Vector& operator=( const REAL value )
59         {
60             const size_t s = this->size();
61             Vector &self = *this;
62             for( size_t i=0; i<s; ++i )
63                 self[i] = value;
64             return *this;
65         }
66
67         Vector sub (size_type i, size_type m)
68         {
69             Vector v(m);
70             Vector &self = *this;
71             size_type k=0;
72             for (size_type j=i; j<i+m; j++){

```

```

110         v[k]=self[j];
111         k++;
112     }
113     return v;
114 }
115
116
117
118 #ifdef DOXYGEN
119 Vector& operator=( const Vector& y )
120 {
121     // It is already implemented in the STL vector class itself!
122 }
123 #endif
124
125
126
127
128 Vector& operator*=( const REAL value )
129 {
130     Vector &self = *this;
131     for (size_t i = 0; i < this->size(); ++i)
132         self[i] *= value;
133     return *this;
134 }
135
136
137
138 Vector& operator/=( const REAL value )
139 {
140     Vector &self = *this;
141     for (size_t i = 0; i < this->size(); ++i)
142         self[i] /= value;
143     return *this;
144 }
145
146
147
148 Vector& operator+=( const Vector & y )
149 {
150     assert( this->size() == y.size() );
151     Vector &self = *this;
152     for (size_t i = 0; i < this->size(); ++i)
153         self[i] += y[i];
154     return *this;
155 }
156
157
158
159 Vector& operator-=( const Vector & y )
160 {
161     assert( this->size() == y.size() );
162     Vector &self = *this;
163     for (size_t i = 0; i < this->size(); ++i)
164         self[i] -= y[i];
165     return *this;
166 }
167
168
169
170 Vector & update(const REAL alpha, const Vector & y)
171 {
172     assert( this->size() == y.size() );
173     Vector &self = *this;
174     for (size_t i = 0; i < this->size(); ++i)
175         self[i] += alpha * y[i];
176     return *this;
177 }
178
179
180
181 REAL operator*(Vector & x) const
182 {
183     assert( x.size() == this->size() ); // checks if the dimensions of the two vectors are equal
184     REAL sum( 0 );
185     const Vector & self = *this;
186     for( size_t i = 0; i < this->size(); ++i )
187         sum += self[i] * x[i];
188     return sum;
189 }
190
191
192
193 Vector operator+(Vector & x) const
194 {
195     assert( x.size() == this->size() ); // checks if the dimensions of the two vectors are equal
196     Vector sum( *this );
197     sum += x;
198     return sum;
199 }
200
201
202

```

```

296
329 Vector operator-(Vector & x) const
330 {
331     assert( x.size() == this->size() ); // checks if the dimensions of the two vectors are equal
332     Vector sum( *this );
333     sum -= x;
334     return sum;
335 }
336
337
338
340 REAL two_norm_2() const
341 {
342     REAL sum( 0 );
343     const Vector & self = *this;
344     for (size_t i = 0; i < (size_t) this->size(); ++i)
345         sum += self[i] * self[i];
346     return sum;
347 }
348
373 REAL two_norm() const
374 {
375     return sqrt(two_norm_2());
376 }
377
379 bool scientific() const
380 {
381     return bScientific;
382 }
383
411 void scientific(bool b) const
412 {
413     bScientific=b;
414 }
415
417 std::size_t iwidth () const
418 {
419     return nIndexWidth;
420 }
421
423 std::size_t width () const
424 {
425     return nValueWidth;
426 }
427
429 std::size_t precision () const
430 {
431     return nValuePrecision;
432 }
433
435 void iwidth (std::size_t i) const
436 {
437     nIndexWidth=i;
438 }
439
441 void width (std::size_t i) const
442 {
443     nValueWidth=i;
444 }
445
447 void precision (std::size_t i) const
448 {
449     nValuePrecision=i;
450 }
451
452 };
453
454
455
456 template<typename REAL>
457 bool Vector<REAL>::bScientific = true;
458
459 template<typename REAL>
460 std::size_t Vector<REAL>::nIndexWidth = 2;
461
462 template<typename REAL>
463 std::size_t Vector<REAL>::nValueWidth = 15;
464
465 template<typename REAL>
466 std::size_t Vector<REAL>::nValuePrecision = 7;
467
468
490 template <typename REAL>
491 inline std::ostream & operator <<(std::ostream & os, const Vector<REAL> & x)
492 {
493     os << std::endl;
494

```

```

495     for (size_t r = 0; r < x.size(); ++r)
496     {
497         if( x.scientific() )
498         {
499             os << "["
500                << std::setw(x.iwidth())
501                << r
502                << "]"
503                << std::scientific
504                << std::showpoint
505                << std::setw( x.width() )
506                << std::setprecision( x.precision() )
507                << x[r]
508                << std::endl;
509         }
510         else
511         {
512             os << "["
513                << std::setw(x.iwidth())
514                << r
515                << "]"
516                << std::fixed
517                << std::showpoint
518                << std::setw( x.width() )
519                << std::setprecision( x.precision() )
520                << x[r]
521                << std::endl;
522         }
523     }
524     return os;
525 }
526
527
528
529
530
531 template<typename REAL>
532 inline void gnuplot(
533     const std::string& fname,
534     const Vector<REAL> x
535 )
536 {
537     std::fstream f(fname.c_str(),std::ios::out);
538     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
539     {
540         if( x.scientific() )
541         {
542             f << std::setw(x.width())
543                << i
544                << std::scientific
545                << std::showpoint
546                << std::setw( x.width() )
547                << std::setprecision( x.precision() )
548                << x[i]
549                << std::endl;
550         }
551         else
552         {
553             f << std::setw(x.width())
554                << i
555                << std::fixed
556                << std::showpoint
557                << std::setw( x.width() )
558                << std::setprecision( x.precision() )
559                << x[i]
560                << std::endl;
561         }
562     }
563     f.close();
564 }
565
566
567
568
569
570 template<typename REAL>
571 inline void gnuplot(
572     const std::string& fname,
573     const std::vector<std::string>& t,
574     const Vector<REAL> x
575 )
576 {
577     std::fstream f(fname.c_str(),std::ios::out);
578     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
579     {
580         if( x.scientific() )
581         {
582             f << t[i] << " "
583                << std::scientific
584                << std::showpoint
585                << std::setw( x.width() )
586                << std::setprecision( x.precision() )
587                << x[i]

```

```

604         « std::endl;
605     }
606     else
607     {
608         f « t[i] « " "
609         « std::fixed
610         « std::showpoint
611         « std::setw( x.width() )
612         « std::setprecision( x.precision() )
613         « x[i]
614         « std::endl;
615     }
616 }
617 f.close();
618 }
619
620 template<typename REAL>
621 inline void gnuplot(
622     const std::string& fname,
623     const Vector<REAL> x,
624     const Vector<REAL> y
625 )
626 {
627     std::fstream f(fname.c_str(),std::ios::out);
628     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
629     {
630         if( x.scientific() )
631         {
632             f « std::setw(x.width())
633             « i
634             « std::scientific
635             « std::showpoint
636             « std::setw( x.width() )
637             « std::setprecision( x.precision() )
638             « x[i]
639             « " "
640             « std::setw( x.width() )
641             « std::setprecision( x.precision() )
642             « y[i]
643             « std::endl;
644         }
645         else
646         {
647             f « std::setw(x.width())
648             « i
649             « std::fixed
650             « std::showpoint
651             « std::setw( x.width() )
652             « std::setprecision( x.precision() )
653             « x[i]
654             « " "
655             « std::setw( x.width() )
656             « std::setprecision( x.precision() )
657             « y[i]
658             « std::endl;
659         }
660     }
661 }
662
663 f.close();
664 }
665
666
667
668 template<typename REAL>
669 inline void readVectorFromFile (const std::string& filename, Vector<REAL> &vector)
670 {
671     std::string buffer;
672     std::ifstream fin( filename.c_str() );
673     if( fin.is_open() ){
674         while( fin ){
675             std::string sub;
676             fin » sub;
677             //std::cout « " sub = " « sub.c_str() « ": ";
678             if( sub.length()>0 ){
679                 REAL a = atof(sub.c_str());
680                 //std::cout « std::fixed « std::setw(10) « std::setprecision(5) « a;
681                 vector.push_back(a);
682             }
683         }
684         fin.close();
685     }
686     else{
687         HDNUM_ERROR("Could not open file!");
688     }
689 }
690
691

```

```
721 template<class REAL>
722 inline void zero (Vector<REAL>& x)
723 {
724     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
725         x[i] = REAL(0);
726 }
727
728 template<class REAL>
729 inline REAL norm (Vector<REAL> x)
730 {
731     REAL sum(0.0);
732     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
733         sum += x[i]*x[i];
734     return sqrt(sum);
735 }
736
737 template<class REAL>
738 inline void fill (Vector<REAL>& x, const REAL t)
739 {
740     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
741         x[i] = t;
742 }
743
744 template<class REAL>
745 inline void fill (Vector<REAL>& x, const REAL& t, const REAL& dt)
746 {
747     REAL myt(t);
748     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
749     {
750         x[i] = myt;
751         myt += dt;
752     }
753 }
754
755 template<class REAL>
756 inline void unitvector (Vector<REAL> & x, std::size_t j)
757 {
758     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
759     {
760         if (i==j)
761             x[i] = REAL(1);
762         else
763             x[i] = REAL(0);
764     }
765 }
766
767 } // end of namespace hdnum
768
769 #endif /* _VECTOR_HH */
```



# Index

begin  
  hdnum::SparseMatrix< REAL >, 59

cbegin  
  hdnum::SparseMatrix< REAL >, 59

cend  
  hdnum::SparseMatrix< REAL >, 59

colsize  
  hdnum::DenseMatrix< REAL >, 14  
  hdnum::SparseMatrix< REAL >, 59

DIRK  
  hdnum::DIRK< M, S >, 32, 33

end  
  hdnum::SparseMatrix< REAL >, 60

exceptions.hh  
  HDNUM\_ERROR, 84  
  HDNUM\_THROW, 84

fill  
  hdnum::Vector< REAL >, 74

getNeighborIndex  
  hdnum::SGrid< N, DF, dimension >, 55

getNonlinearProblem  
  newton.hh, 91

gnuplot  
  hdnum::Vector< REAL >, 74

gram\_schmidt  
  qr.hh, 130

hdnum::Banach, 7

hdnum::DenseMatrix< REAL >, 11  
  colsize, 14  
  identity, 28  
  mm, 14  
  mv, 15  
  operator\*, 16, 17  
  operator\*=: 18  
  operator(), 16  
  operator+, 18  
  operator+=, 19  
  operator-, 20  
  operator-=, 20  
  operator/=: 21  
  operator=, 21, 22  
  readMatrixFromFileDat, 28  
  readMatrixFromFileMatrixMarket, 29  
  rowsize, 22  
  sc, 22

scientific, 23

spd, 30

sr, 23

sub, 24

transpose, 24

umm, 25

umv, 25, 26

update, 27

vandermonde, 30

hdnum::DIRK< M, S >, 31  
  DIRK, 32, 33

hdnum::EE< M >, 33

hdnum::ErrorException, 34

hdnum::Exception, 35

hdnum::GenericNonlinearProblem< Lambda, Vec >, 35

hdnum::Heun2< M >, 36

hdnum::Heun3< M >, 37

hdnum::IE< M, S >, 38

hdnum::ImplicitRungeKuttaStepProblem< M >, 39

hdnum::InvalidStateException, 40

hdnum::IOError, 41

hdnum::Kutta3< M >, 41

hdnum::MathError, 42

hdnum::ModifiedEuler< M >, 43

hdnum::Newton, 44

hdnum::NotImplemented, 45

hdnum::oc::OpCounter< F >, 46

hdnum::oc::OpCounter< F >::Counters, 10

hdnum::OutOfMemoryError, 47

hdnum::RangeError, 48

hdnum::RE< M, S >, 48

hdnum::RK45< M >, 50

hdnum::RungeKutta< M, S >, 51

hdnum::RungeKutta4< M >, 53

hdnum::SGrid< N, DF, dimension >, 54  
  getNeighborIndex, 55  
  SGrid, 55

hdnum::SparseMatrix< REAL >, 56  
  begin, 59  
  cbegin, 59  
  cend, 59  
  colsize, 59  
  end, 60  
  identity, 60, 64  
  matchingIdentity, 61  
  mv, 61  
  norm\_infty, 62  
  operator\*, 62  
  operator\*=: 62

- operator/=, 62
- rowsize, 63
- scientific, 63
- SparseMatrix, 58
- umv, 63
- hdnum::SparseMatrix< REAL >::builder, 8
- hdnum::SparseMatrix< REAL >::column\_index\_iterator, 8
- hdnum::SparseMatrix< REAL >::const\_column\_index\_iterator, 9
- hdnum::SparseMatrix< REAL >::const\_row\_iterator, 9
- hdnum::SparseMatrix< REAL >::row\_iterator, 51
- hdnum::SquareRootProblem< N >, 65
- hdnum::StationarySolver< M >, 66
- hdnum::SystemError, 66
- hdnum::Timer, 67
- hdnum::TimerError, 68
- hdnum::Vector< REAL >, 68
  - fill, 74
  - gnuplot, 74
  - operator<<, 74
  - operator\*, 70
  - operator+, 71
  - operator-, 71
  - operator=, 72
  - readVectorFromFile, 75
  - scientific, 72
  - sub, 73
  - two\_norm, 73
  - unitvector, 75
- HDNUM\_ERROR
  - exceptions.hh, 84
- HDNUM\_THROW
  - exceptions.hh, 84
- identity
  - hdnum::DenseMatrix< REAL >, 28
  - hdnum::SparseMatrix< REAL >, 60, 64
- matchingIdentity
  - hdnum::SparseMatrix< REAL >, 61
- mm
  - hdnum::DenseMatrix< REAL >, 14
- modified\_gram\_schmidt
  - qr.hh, 130
- mv
  - hdnum::DenseMatrix< REAL >, 15
  - hdnum::SparseMatrix< REAL >, 61
- newton.hh
  - getNonlinearProblem, 91
- norm\_infty
  - hdnum::SparseMatrix< REAL >, 62
- operator<<
  - hdnum::Vector< REAL >, 74
- operator\*
  - hdnum::DenseMatrix< REAL >, 16, 17
  - hdnum::SparseMatrix< REAL >, 62
- hdnum::Vector< REAL >, 70
- operator\*=
  - hdnum::DenseMatrix< REAL >, 18
  - hdnum::SparseMatrix< REAL >, 62
- operator()
  - hdnum::DenseMatrix< REAL >, 16
- operator+
  - hdnum::DenseMatrix< REAL >, 18
- operator+=
  - hdnum::DenseMatrix< REAL >, 19
- operator-
  - hdnum::DenseMatrix< REAL >, 20
  - hdnum::Vector< REAL >, 71
- operator-=
  - hdnum::DenseMatrix< REAL >, 20
- operator/=
  - hdnum::DenseMatrix< REAL >, 21
  - hdnum::SparseMatrix< REAL >, 62
- operator=
  - hdnum::DenseMatrix< REAL >, 21, 22
  - hdnum::Vector< REAL >, 72
- ordertest
  - rungekutta.hh, 143
- permute\_forward
  - qr.hh, 131
- qr.hh
  - gram\_schmidt, 130
  - modified\_gram\_schmidt, 130
  - permute\_forward, 131
  - qr\_gram\_schmidt, 131
  - qr\_gram\_schmidt\_pivoting, 132
  - qr\_gram\_schmidt\_simple, 133
- qr\_gram\_schmidt
  - qr.hh, 131
- qr\_gram\_schmidt\_pivoting
  - qr.hh, 132
- qr\_gram\_schmidt\_simple
  - qr.hh, 133
- qrhousholder
  - qrhousholder.hh, 138
- qrhousholder.hh
  - qrhousholder, 138
  - qrhousholderexplicitQ, 140
- qrhousholderexplicitQ
  - qrhousholder.hh, 140
- readMatrixFromFileDat
  - hdnum::DenseMatrix< REAL >, 28
- readMatrixFromFileMatrixMarket
  - hdnum::DenseMatrix< REAL >, 29
- readVectorFromFile
  - hdnum::Vector< REAL >, 75
- rowsize
  - hdnum::DenseMatrix< REAL >, 22
  - hdnum::SparseMatrix< REAL >, 63
- rungekutta.hh

- ordertest, [143](#)
- sc
  - hdnum::DenseMatrix< REAL >, [22](#)
- scientific
  - hdnum::DenseMatrix< REAL >, [23](#)
  - hdnum::SparseMatrix< REAL >, [63](#)
  - hdnum::Vector< REAL >, [72](#)
- SGrid
  - hdnum::SGrid< N, DF, dimension >, [55](#)
- SparseMatrix
  - hdnum::SparseMatrix< REAL >, [58](#)
- spd
  - hdnum::DenseMatrix< REAL >, [30](#)
- sr
  - hdnum::DenseMatrix< REAL >, [23](#)
- src/densematrix.hh, [77](#)
- src/exceptions.hh, [83](#), [86](#)
- src/lr.hh, [87](#), [88](#)
- src/newton.hh, [91](#), [92](#)
- src/ode.hh, [96](#), [97](#)
- src/opcounter.hh, [113](#), [116](#)
- src/pde.hh, [126](#), [127](#)
- src/precision.hh, [128](#)
- src/qr.hh, [129](#), [134](#)
- src/qrhousholder.hh, [138](#), [140](#)
- src/rungekutta.hh, [142](#), [144](#)
- src/sgrid.hh, [149](#)
- src/sparsematrix.hh, [151](#)
- src/timer.hh, [161](#), [162](#)
- src/vector.hh, [163](#)
- sub
  - hdnum::DenseMatrix< REAL >, [24](#)
  - hdnum::Vector< REAL >, [73](#)
- transpose
  - hdnum::DenseMatrix< REAL >, [24](#)
- two\_norm
  - hdnum::Vector< REAL >, [73](#)
- umm
  - hdnum::DenseMatrix< REAL >, [25](#)
- umv
  - hdnum::DenseMatrix< REAL >, [25](#), [26](#)
  - hdnum::SparseMatrix< REAL >, [63](#)
- unitvector
  - hdnum::Vector< REAL >, [75](#)
- update
  - hdnum::DenseMatrix< REAL >, [27](#)
- vandermonde
  - hdnum::DenseMatrix< REAL >, [30](#)