

Heidelberg Educational Numerics Library

Version 0.27 (from 15 March 2021)

Generated by Doxygen 1.10.0

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

4.7.2.23 umv() [2/2]	27
4.7.2.24 update()	28
4.7.3 Friends And Related Symbol Documentation	28
4.7.3.1 identity()	28
4.7.3.2 readMatrixFromFileDat()	29
4.7.3.3 readMatrixFromFileMatrixMarket()	29
4.7.3.4 spd()	30
4.7.3.5 vandermonde()	31
4.8 hdnum::DIRK< M, S > Class Template Reference	31
4.8.1 Detailed Description	32
4.8.2 Constructor & Destructor Documentation	33
4.8.2.1 DIRK() [1/2]	33
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	35
4.11 hdnum::Exception Class Reference	35
4.11.1 Detailed Description	35
4.12 hdnum::GenericNonlinearProblem< Lambda, Vec > Class Template Reference	36
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	39
4.15.1 Detailed Description	39
4.16 hdnum::ImplicitRungeKuttaStepProblem< M > Class Template Reference	40
4.16.1 Detailed Description	40
4.17 hdnum::InvalidStateException Class Reference	41
4.17.1 Detailed Description	41
4.18 hdnum::IOError Class Reference	41
4.18.1 Detailed Description	42
4.19 hdnum::Kutta3< M > Class Template Reference	42
4.19.1 Detailed Description	43
4.20 hdnum::MathError Class Reference	43
4.20.1 Detailed Description	44
4.21 hdnum::ModifiedEuler< M > Class Template Reference	44
4.21.1 Detailed Description	45
4.22 hdnum::Newton Class Reference	45
4.22.1 Detailed Description	46
4.23 hdnum::NotImplemented Class Reference	46

4.23.1 Detailed Description	47
4.24 <code>hdnum::oc::OpCounter< F ></code> Class Template Reference	47
4.24.1 Detailed Description	48
4.25 <code>hdnum::OutOfMemoryError</code> Class Reference	48
4.25.1 Detailed Description	49
4.26 <code>hdnum::RangeError</code> Class Reference	49
4.26.1 Detailed Description	49
4.27 <code>hdnum::RE< M, S ></code> Class Template Reference	50
4.27.1 Detailed Description	50
4.28 <code>hdnum::RKF45< M ></code> Class Template Reference	51
4.28.1 Detailed Description	51
4.29 <code>hdnum::SparseMatrix< REAL >::row_iterator</code> Class Reference	52
4.30 <code>hdnum::RungeKutta< M, S ></code> Class Template Reference	53
4.30.1 Detailed Description	53
4.31 <code>hdnum::RungeKutta4< M ></code> Class Template Reference	54
4.31.1 Detailed Description	54
4.32 <code>hdnum::SGrid< N, DF, dimension ></code> Class Template Reference	55
4.32.1 Detailed Description	56
4.32.2 Constructor & Destructor Documentation	56
4.32.2.1 <code>SGrid()</code>	56
4.32.3 Member Function Documentation	56
4.32.3.1 <code>getNeighborIndex()</code>	56
4.33 <code>hdnum::SparseMatrix< REAL ></code> Class Template Reference	57
4.33.1 Detailed Description	59
4.33.2 Constructor & Destructor Documentation	59
4.33.2.1 <code>SparseMatrix()</code>	59
4.33.3 Member Function Documentation	60
4.33.3.1 <code>begin()</code> [1/2]	60
4.33.3.2 <code>begin()</code> [2/2]	60
4.33.3.3 <code>cbegin()</code>	60
4.33.3.4 <code>end()</code>	60
4.33.3.5 <code>colsize()</code>	61
4.33.3.6 <code>end()</code> [1/2]	61
4.33.3.7 <code>end()</code> [2/2]	61
4.33.3.8 <code>identity()</code>	61
4.33.3.9 <code>matchingIdentity()</code>	62
4.33.3.10 <code>mv()</code>	62
4.33.3.11 <code>norm_infty()</code>	62
4.33.3.12 <code>operator*()</code>	63
4.33.3.13 <code>operator*=(())</code>	63
4.33.3.14 <code>operator/=(())</code>	63
4.33.3.15 <code>rowsize()</code>	63

4.33.3.16 scientific()	64
4.33.3.17 umv()	64
4.33.4 Friends And Related Symbol Documentation	64
4.33.4.1 identity()	64
4.34 hdnum::SquareRootProblem< N > Class Template Reference	65
4.34.1 Detailed Description	66
4.35 hdnum::StationarySolver< M > Class Template Reference	66
4.35.1 Detailed Description	66
4.36 hdnum::SystemError Class Reference	67
4.36.1 Detailed Description	67
4.37 hdnum::Timer Class Reference	67
4.37.1 Detailed Description	68
4.38 hdnum::TimerError Class Reference	68
4.38.1 Detailed Description	69
4.39 hdnum::Vector< REAL > Class Template Reference	69
4.39.1 Detailed Description	71
4.39.2 Member Function Documentation	71
4.39.2.1 operator*()	71
4.39.2.2 operator+()	71
4.39.2.3 operator-()	72
4.39.2.4 operator=()	72
4.39.2.5 scientific()	73
4.39.2.6 sub()	73
4.39.2.7 two_norm()	74
4.39.3 Friends And Related Symbol Documentation	74
4.39.3.1 fill()	74
4.39.3.2 gnuplot()	75
4.39.3.3 operator<<()	75
4.39.3.4 readVectorFromFile()	75
4.39.3.5 unitvector()	76
5 File Documentation	77
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	84
5.3 exceptions.hh	85
5.4 src/lr.hh File Reference	86
5.4.1 Detailed Description	87
5.5 lr.hh	87

5.6 src/newton.hh File Reference	90
5.6.1 Detailed Description	90
5.6.2 Function Documentation	90
5.6.2.1 getNonlinearProblem()	90
5.7 newton.hh	91
5.8 src/ode.hh File Reference	95
5.8.1 Detailed Description	96
5.9 ode.hh	96
5.10 src/opcounter.hh File Reference	112
5.10.1 Detailed Description	115
5.11 opcounter.hh	115
5.12 src/pde.hh File Reference	125
5.12.1 Detailed Description	126
5.13 pde.hh	126
5.14 src/precision.hh File Reference	127
5.14.1 Detailed Description	127
5.15 precision.hh	127
5.16 src/qr.hh File Reference	128
5.16.1 Detailed Description	128
5.16.2 Function Documentation	129
5.16.2.1 gram_schmidt()	129
5.16.2.2 modified_gram_schmidt()	129
5.16.2.3 permute_forward()	130
5.16.2.4 qr_gram_schmidt()	130
5.16.2.5 qr_gram_schmidt_pivoting()	131
5.16.2.6 qr_gram_schmidt_simple()	132
5.17 qr.hh	133
5.18 src/qrhousholder.hh File Reference	137
5.18.1 Detailed Description	137
5.18.2 Function Documentation	138
5.18.2.1 qrhousholder()	138
5.18.2.2 qrhousholderexplicitQ()	139
5.19 qrhousholder.hh	139
5.20 src/rungekutta.hh File Reference	141
5.20.1 Detailed Description	142
5.20.2 Function Documentation	142
5.20.2.1 ordertest()	142
5.21 rungekutta.hh	142
5.22 sgrid.hh	148
5.23 sparsematrix.hh	150
5.24 src/timer.hh File Reference	160
5.24.1 Detailed Description	161

5.25 timer.hh	161
5.26 vector.hh	162
Index	169

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	41
hdnum::MathError	43
hdnum::NotImplemented	46
hdnum::RangeError	49
hdnum::SystemError	67
hdnum::OutOfMemoryError	48
hdnum::TimerError	68
hdnum::GenericNonlinearProblem< Lambda, Vec >	36
hdnum::Heun2< M >	36
hdnum::Heun3< M >	37
hdnum::IE< M, S >	39
hdnum::ImplicitRungeKuttaStepProblem< M >	40
hdnum::Kutta3< M >	42
hdnum::ModifiedEuler< M >	44
hdnum::Newton	45
hdnum::oc::OpCounter< F >	47
hdnum::RE< M, S >	50
hdnum::RKF45< M >	51
hdnum::SparseMatrix< REAL >::row_iterator	52
hdnum::RungeKutta< M, S >	53
hdnum::RungeKutta4< M >	54
hdnum::SGrid< N, DF, dimension >	55

hdnum::SparseMatrix< REAL >	57
hdnum::SquareRootProblem< N >	65
hdnum::StationarySolver< M >	66
hdnum::Timer	67
std::vector	
hdnum::Vector< number_type >	69
hdnum::Vector< hdnum::Vector< number_type > >	69
hdnum::Vector< size_type >	69
hdnum::Vector< REAL >	69

Chapter 2

Class Index

2.1 Class List

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

hdnum::Banach	Solve nonlinear problem using a fixed point iteration	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	Struct storing the number of operations	10
hdnum::DenseMatrix< REAL >	Class with mathematical matrix operations	11
hdnum::DIRK< M, S >	Implementation of a general Diagonal Implicit Runge-Kutta method	31
hdnum::EE< M >	Explicit Euler method as an example for an ODE solver	33
hdnum::ErrorException	General Error	34
hdnum::Exception	Base class for Exceptions	35
hdnum::GenericNonlinearProblem< Lambda, Vec >	A generic problem class that can be set up with a lambda defining $F(x)=0$	36
hdnum::Heun2< M >	Heun method (order 2 with 2 stages)	36
hdnum::Heun3< M >	Heun method (order 3 with 3 stages)	37
hdnum::IE< M, S >	Implicit Euler using Newton's method to solve nonlinear system	39
hdnum::ImplicitRungeKuttaStepProblem< M >	Nonlinear problem we need to solve to do one step of an implicit Runge Kutta method	40
hdnum::InvalidStateException	Default exception if a function was called while the object is not in a valid state for that function	41
hdnum::IOError	Default exception class for I/O errors	41
hdnum::Kutta3< M >	Kutta method (order 3 with 3 stages)	42
hdnum::MathError	Default exception class for mathematical errors	43

hdnum::ModifiedEuler< M >	Modified Euler method (order 2 with 2 stages)	44
hdnum::Newton	Solve nonlinear problem using a damped Newton method	45
hdnum::NotImplemented	Default exception for dummy implementations	46
hdnum::oc::OpCounter< F >	47
hdnum::OutOfMemoryError	Default exception if memory allocation fails	48
hdnum::RangeError	Default exception class for range errors	49
hdnum::RE< M, S >	Adaptive one-step method using Richardson extrapolation	50
hdnum::RKF45< M >	Adaptive Runge-Kutta-Fehlberg method	51
hdnum::SparseMatrix< REAL >::row_iterator	52
hdnum::RungeKutta< M, S >	Classical Runge-Kutta method (order n with n stages)	53
hdnum::RungeKutta4< M >	Classical Runge-Kutta method (order 4 with 4 stages)	54
hdnum::SGrid< N, DF, dimension >	Structured Grid for Finite Differences	55
hdnum::SparseMatrix< REAL >	Sparse matrix Class with mathematical matrix operations	57
hdnum::SquareRootProblem< N >	Example class for a nonlinear model $F(x) = 0$;	65
hdnum::StationarySolver< M >	Stationary problem solver. E.g. for elliptic problems	66
hdnum::SystemError	Default exception class for OS errors	67
hdnum::Timer	A simple stop watch	67
hdnum::TimerError	Exception thrown by the Timer class	68
hdnum::Vector< REAL >	Class with mathematical vector operations	69

Chapter 3

File Index

3.1 File List

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

src/densematrix.hh	77
src/exceptions.hh	
A few common exception classes	83
src/lr.hh	
This file implements LU decomposition	86
src/newton.hh	
Newton's method with line search	90
src/ode.hh	
Solvers for ordinary differential equations	95
src/opcounter.hh	
This file implements an operator counting class	112
src/pde.hh	
Solvers for partial differential equations	125
src/precision.hh	
Find machine precision for given float type	127
src/qr.hh	
This file implements QR decomposition using Gram-Schmidt method	128
src/qrhousholder.hh	
This file implements QR decomposition using housholder transformation	137
src/rungekutta.hh	141
src/sgrid.hh	148
src/sparsematrix.hh	150
src/timer.hh	
A simple timing class	160
src/vector.hh	162

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 [vallter](#))
- **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_infty () 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 Symbols

(Note that these are not member symbols.)

- `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 hdnun::DenseMatrix< REAL >
```

Class with mathematical matrix operations.

4.7.2 Member Function Documentation

4.7.2.1 colsize()

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

get number of columns of the matrix

Example:

```
hdnun::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 hdnun::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	<i>A</i>	constant reference to a DenseMatrix
in	<i>B</i>	constant reference to a DenseMatrix

Example:

```
hdnun::DenseMatrix<double> A(2,6,1.0);
hdnun::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;
```

```
hdnun::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 Vector
in	x	constant reference to a Vector

Example:

```
hdnun::Vector<double> x(3,10.0);
hdnun::Vector<double> y(2);
hdnun::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()

```
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 DenseMatrix
----	---	---

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 > hdnm::DenseMatrix< REAL >::operator* (
    const Vector< REAL > & x ) const [inline]

```

vector = matrix * vector

Parameters

in	x	constant reference to a Vector
----	---	--

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 DenseMatrix
----	---	---

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 & hdnnum::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 hdnnum::DenseMatrix< REAL >::operator- (
    const DenseMatrix< REAL > & x ) const [inline]

```

matrix = matrix - matrix

Parameters

in	x	constant reference to a DenseMatrix
----	---	---

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	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
```

```
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 Vector
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 hdnum::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 hdnum::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 Vector
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>i</i>	first row index of the new matrix
in	<i>j</i>	first column index of the new matrix
in	<i>rows</i>	row size of the new matrix, i.e. it has components [i,i+rows-1]
in	<i>cols</i>	column size of the new matrix, i.e. it has components [j,j+cols-1]

4.7.2.20 transpose()

```

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

```

Transposition.

Return the transposed as a new matrix.

4.7.2.21 umm()

```

template<typename REAL >
void hdnun::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 DenseMatrix
in	<i>B</i>	constant reference to a DenseMatrix

Example:

```

hdnun::DenseMatrix<double> A(2,6,1.0);
hdnun::DenseMatrix<double> B(6,3,-1.0);
hdnun::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 Vector
in	s	constant reference to a number type
in	x	constant reference to a Vector

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 = sA*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 = s*A*x =
[ 0]    20.0000000
[ 1]    20.0000000

```

4.7.2.23 umv() [2/2]

```

template<typename REAL >
template<class V >
void hdnm::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 Vector
in	x	constant reference to a Vector

Example:

```

hdnm::Vector<double> x(3,10.0);
hdnm::Vector<double> y(2,5.0);
hdnm::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 Symbol 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 DenseMatrix 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 DenseMatrix

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 DenseMatrix

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
```

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	<i>A</i>	reference to a DenseMatrix 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 DenseMatrix that shall be filled with entries
in	x	constant reference to a Vector

Example:

```
hdnm::Vector<double> x(4);
fill(x,2.0,1.0);
hdnm::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=
0      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 hdnm::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:

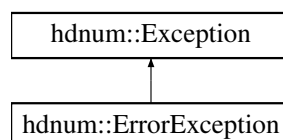
- [src/ode.hh](#)

4.10 hdnum::ErrorException Class Reference

General Error.

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

Inheritance diagram for `hdnum::ErrorException`:



Additional Inherited Members

Public Member Functions inherited from [hdnum::Exception](#)

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

4.10.1 Detailed Description

General Error.

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

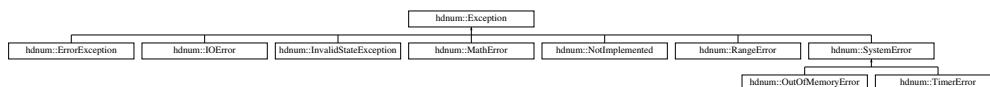
- [src/exceptions.hh](#)

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 hdnm::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 hdnm::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 Vector to a Vector
<i>Vec</i>	the type for the Vector

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

- [src/newton.hh](#)

4.13 hdnm::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 hdnm::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 implicate 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:

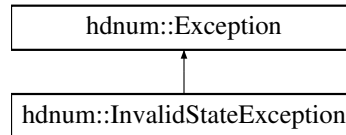
- [src/rungekutta.hh](#)

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

Public Member Functions inherited from `hdnum::Exception`

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

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:

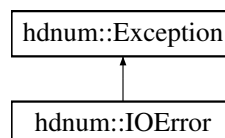
- [src/exceptions.hh](#)

4.18 `hdnum::IOError` Class Reference

Default exception class for I/O errors.

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

Inheritance diagram for `hdnum::IOError`:



Additional Inherited Members

Public Member Functions inherited from [hdnum::Exception](#)

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

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:

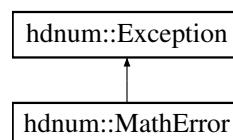
- [src/ode.hh](#)

4.20 hdnum::MathError Class Reference

Default exception class for mathematical errors.

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

Inheritance diagram for `hdnum::MathError`:



Additional Inherited Members

Public Member Functions inherited from [hdnum::Exception](#)

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

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 hdnm::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 hdnm::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:

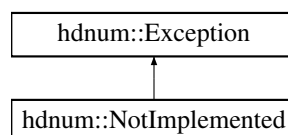
- [src/newton.hh](#)

4.23 hdnum::NotImplemented Class Reference

Default exception for dummy implementations.

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

Inheritance diagram for `hdnum::NotImplemented`:



Additional Inherited Members

Public Member Functions inherited from `hdnum::Exception`

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

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::istringstream & operator>>` (`std::istringstream &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:

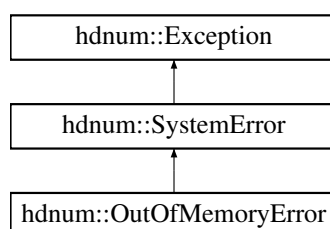
- [src/opcounter.hh](#)

4.25 hdnum::OutOfMemoryError Class Reference

Default exception if memory allocation fails.

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

Inheritance diagram for `hdnum::OutOfMemoryError`:



Additional Inherited Members

Public Member Functions inherited from `hdnum::Exception`

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

4.25.1 Detailed Description

Default exception if memory allocation fails.

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

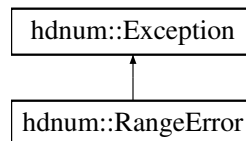
- [src/exceptions.hh](#)

4.26 `hdnum::RangeError` Class Reference

Default exception class for range errors.

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

Inheritance diagram for `hdnum::RangeError`:



Additional Inherited Members

Public Member Functions inherited from `hdnum::Exception`

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

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 vallter`)
- `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 Newton)

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 <i>df_</i> .
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 <code>evaluate(Vector<number_type> x)</code> which returns true if the node which is positioned at the coordinates of <i>x</i> 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. <code>n_dim = 0</code> 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 <code>SGrid::positive</code> or <code>SGrid::negative</code> .
in	<i>k</i>	For <code>k=1</code> 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 <code>invalid_node</code> will be returned. For <code>k=0</code> it will simply return <code>ln</code> .

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 <sparsmatrix.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_inf () 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 Symbols

(Note that these are not member symbols.)

- `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 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      0.000
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 Vector
in	<i>x</i>	constant reference to a Vector

4.33.3.11 norm_infty()

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

calculate row sum norm

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

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	x	constant reference to a Vector
----	-----	--

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 hdnum::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 $y += A*x$

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 Vector
in	<i>x</i>	constant reference to a Vector

4.33.4 Friends And Related Symbol 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

<code>in</code>	<code>A</code>	reference to a SparseMatrix 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 hdnun::SquareRootProblem< N >
```

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

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

Template Parameters

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

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

- [src/newton.hh](#)

4.35 hdnun::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 hdnun::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:

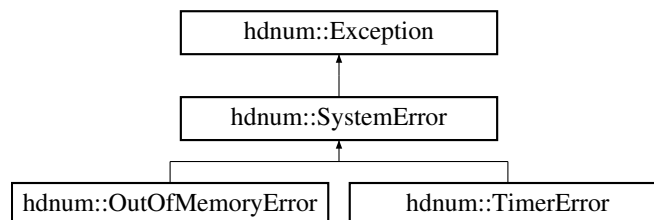
- [src/pde.hh](#)

4.36 `hdnum::SystemError` Class Reference

Default exception class for OS errors.

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

Inheritance diagram for `hdnum::SystemError`:



Additional Inherited Members

Public Member Functions inherited from [hdnum::Exception](#)

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

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

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

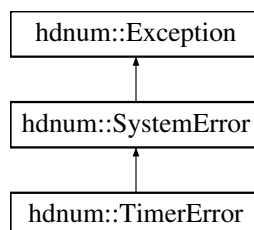
- [src/timer.hh](#)

4.38 hdnum::TimerError Class Reference

Exception thrown by the `Timer` class

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

Inheritance diagram for `hdnum::TimerError`:



Additional Inherited Members

Public Member Functions inherited from `hdnum::Exception`

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

4.38.1 Detailed Description

Exception thrown by the [Timer](#) class

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

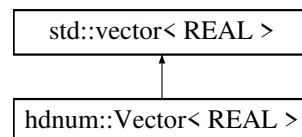
- [src/timer.hh](#)

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 Symbols

(Note that these are not member symbols.)

- **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]    12.0000000
[ 1]     3.0000000

y=
[ 0]     4.0000000
[ 1]    -1.0000000

s = x*y = 45.0000000
```

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]    12.0000000
[ 1]     3.0000000

y=
[ 0]     4.0000000
[ 1]    -1.0000000

x+y =
[ 0]    16.0000000
[ 1]     2.0000000

```

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]    12.0000000
[ 1]     3.0000000

y=
[ 0]     4.0000000
[ 1]    -1.0000000

x-y =
[ 0]     8.0000000
[ 1]     4.0000000

```

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	<i>value</i>	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 Symbol 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

<code>in</code>	<i>filename</i>	name of the text file
<code>in, out</code>	<i>vector</i>	reference to a Vector

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

```
00001 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
00002 /*
00003  * File:    densematrix.hh
00004  * Author:  ngo
00005  *
00006  * Created on April 15, 2011
00007  */
00008
00009 #ifndef DENSEMATRIX_HH
00010 #define DENSEMATRIX_HH
00011
00012 #include <cstdlib>
00013 #include <fstream>
00014 #include <iomanip>
00015 #include <iostream>
00016 #include <sstream>
00017 #include <string>
00018
00019 #include "exceptions.hh"
00020 #include "sparsematrix.hh"
00021 #include "vector.hh"
00022
00023 namespace hdnum {
00024
00025 // forward-declare the sparse matrix template to make the transforming
00026 // constructor from hdnum::SparseMatrix -> hdnum::DenseMatrix working
00027 template <typename REAL>
00028 class SparseMatrix;
00029
00032 template <typename REAL>
00033 class DenseMatrix {
00034 public:
00036     typedef std::size_t size_type;
00037     typedef typename std::vector<REAL> VType;
00038     typedef typename VType::const_iterator ConstVectorIterator;
00039     typedef typename VType::iterator VectorIterator;
00040
00041 private:
00042     VType m_data; // Matrix data is stored in an STL vector!
00043     std::size_t m_rows; // Number of Matrix rows
00044     std::size_t m_cols; // Number of Matrix columns
00045
00046     static bool bScientific;
00047     static std::size_t nIndexWidth;
00048     static std::size_t nValueWidth;
00049     static std::size_t nValuePrecision;
00050
00052     REAL myabs(REAL x) const {
00053         if (x >= REAL(0))
00054             return x;
00055         else
00056             return -x;
00057     }
00058
00060     inline REAL& at(const std::size_t row, const std::size_t col) {
00061         return m_data[row * m_cols + col];
00062     }
00063 }
```

```

00065     inline const REAL& at(const std::size_t row, const std::size_t col) const {
00066         return m_data[row * m_cols + col];
00067     }
00068
00069 public:
00071     DenseMatrix() : m_data(0, 0), m_rows(0), m_cols(0) {}
00072
00073     DenseMatrix(const std::size_t _rows, const std::size_t _cols,
00074                 const REAL def_val = 0)
00075         : m_data(_rows * _cols, def_val), m_rows(_rows), m_cols(_cols) {}
00076
00077     DenseMatrix(const std::initializer_list<std::initializer_list<REAL>>& v) {
00078         m_rows = v.size();
00079         m_cols = v.begin()->size();
00080         for (auto row : v) {
00081             if (row.size() != m_cols) {
00082                 std::cout << "Zeilen der Matrix nicht gleich lang" << std::endl;
00083                 exit(1);
00084             }
00085             for (auto elem : row) m_data.push_back(elem);
00086         }
00087     }
00088
00089     DenseMatrix(const hdnum::SparseMatrix<REAL>& other)
00090         : m_data(other.rowsize() * other.colsize(), m_rows(other.rowsize()),
00091                 m_cols(other.colsize())) {
00092         using counter_type = typename hdnum::SparseMatrix<REAL>::size_type;
00093         counter_type row_index {};
00094         for (auto& row : other) {
00095             for (auto it = row.ibegin(); it != row.iend(); it++) {
00096                 this->operator[](row_index)[it.index()] = it.value();
00097             }
00098             row_index++;
00099         }
00100     }
00101
00102     void addNewRow(const hdnum::Vector<REAL>& rowvector) {
00103         m_rows++;
00104         m_cols = rowvector.size();
00105         for (std::size_t i = 0; i < m_cols; i++) m_data.push_back(rowvector[i]);
00106     }
00107
00108     /*
00109     // copy constructor (not needed, since it inherits from the STL vector)
00110     DenseMatrix( const DenseMatrix& A )
00111     {
00112         this->m_data = A.m_data;
00113         m_rows = A.m_rows;
00114         m_cols = A.m_cols;
00115     }
00116     */
00117
00118     size_t rowsize() const { return m_rows; }
00119
00120     size_t colsize() const { return m_cols; }
00121
00122     // pretty-print output properties
00123     bool scientific() const { return bScientific; }
00124
00125     void scientific(bool b) const { bScientific = b; }
00126
00127     std::size_t iwidth() const { return nIndexWidth; }
00128
00129     std::size_t width() const { return nValueWidth; }
00130
00131     std::size_t precision() const { return nValuePrecision; }
00132
00133     void iwidth(std::size_t i) const { nIndexWidth = i; }
00134
00135     void width(std::size_t i) const { nValueWidth = i; }
00136
00137     void precision(std::size_t i) const { nValuePrecision = i; }
00138
00139     // overloaded element access operators
00140     // write access on matrix element A_ij using A(i,j)
00141     inline REAL& operator()(const std::size_t row, const std::size_t col) {
00142         assert(row < m_rows || col < m_cols);
00143         return at(row, col);
00144     }
00145
00146     inline const REAL& operator()(const std::size_t row,
00147                                   const std::size_t col) const {
00148         assert(row < m_rows || col < m_cols);
00149         return at(row, col);
00150     }
00151
00152     const ConstVectorIterator operator[](const std::size_t row) const {

```

```

00258     assert(row < m_rows);
00259     return m_data.begin() + row * m_cols;
00260 }
00261
00263 VectorIterator operator[](const std::size_t row) {
00264     assert(row < m_rows);
00265     return m_data.begin() + row * m_cols;
00266 }
00267
00290 DenseMatrix& operator=(const DenseMatrix& A) {
00291     m_data = A.m_data;
00292     m_rows = A.m_rows;
00293     m_cols = A.m_cols;
00294     return *this;
00295 }
00296
00316 DenseMatrix& operator=(const REAL value) {
00317     for (std::size_t i = 0; i < rowsize(); i++)
00318         for (std::size_t j = 0; j < colsize(); j++) (*this)(i, j) = value;
00319     return *this;
00320 }
00321
00334 DenseMatrix sub(size_type i, size_type j, size_type rows, size_type cols) {
00335     DenseMatrix A(rows, cols);
00336     DenseMatrix& self = *this;
00337     for (size_type k1 = 0; k1 < rows; k1++) {
00338         for (size_type k2 = 0; k2 < cols; k2++) {
00339             A[k1][k2] = self[k1 + i][k2 + j];
00340         }
00341     }
00342     return A;
00343 }
00344
00350 DenseMatrix transpose() const {
00351     DenseMatrix A(m_cols, m_rows);
00352     for (size_type i = 0; i < m_rows; i++) {
00353         for (size_type j = 0; j < m_cols; j++) {
00354             A[j][i] = this->operator[](i)[j];
00355         }
00356     }
00357     return A;
00358 }
00359
00360 // Basic Matrix Operations
00361
00369 DenseMatrix& operator+=(const DenseMatrix& B) {
00370     for (size_type i = 0; i < rowsize(); ++i) {
00371         for (size_type j = 0; j < colsize(); ++j) {
00372             (*this)(i, j) += B(i, j);
00373         }
00374     }
00375     return *this;
00376 }
00377
00385 DenseMatrix& operator-=(const DenseMatrix& B) {
00386     for (std::size_t i = 0; i < rowsize(); ++i)
00387         for (std::size_t j = 0; j < colsize(); ++j)
00388             (*this)(i, j) -= B(i, j);
00389     return *this;
00390 }
00391
00421 DenseMatrix& operator*=(const REAL s) {
00422     for (std::size_t i = 0; i < rowsize(); ++i)
00423         for (std::size_t j = 0; j < colsize(); ++j) (*this)(i, j) *= s;
00424     return *this;
00425 }
00426
00457 DenseMatrix& operator/=(const REAL s) {
00458     for (std::size_t i = 0; i < rowsize(); ++i)
00459         for (std::size_t j = 0; j < colsize(); ++j) (*this)(i, j) /= s;
00460     return *this;
00461 }
00462
00489 void update(const REAL s, const DenseMatrix& B) {
00490     for (std::size_t i = 0; i < rowsize(); ++i)
00491         for (std::size_t j = 0; j < colsize(); ++j)
00492             (*this)(i, j) += s * B(i, j);
00493 }
00494
00537 template <class V>
00538 void mv(Vector<V>& y, const Vector<V>& x) const {
00539     if (this->rowsize() != y.size())
00540         HDNUM_ERROR("mv: size of A and y do not match");
00541     if (this->colsize() != x.size())
00542         HDNUM_ERROR("mv: size of A and x do not match");
00543     for (std::size_t i = 0; i < rowsize(); ++i) {
00544         y[i] = 0;

```

```

00545         for (std::size_t j = 0; j < colsize(); ++j)
00546             y[i] += (*this)(i, j) * x[j];
00547     }
00548 }
00549
00597 template <class V>
00598 void umv(Vector<V>& y, const Vector<V>& x) const {
00599     if (this->rowsize() != y.size())
00600         HDNUM_ERROR("mv: size of A and y do not match");
00601     if (this->colsize() != x.size())
00602         HDNUM_ERROR("mv: size of A and x do not match");
00603     for (std::size_t i = 0; i < rowsize(); ++i) {
00604         for (std::size_t j = 0; j < colsize(); ++j)
00605             y[i] += (*this)(i, j) * x[j];
00606     }
00607 }
00608
00659 template <class V>
00660 void umv(Vector<V>& y, const V& s, const Vector<V>& x) const {
00661     if (this->rowsize() != y.size())
00662         HDNUM_ERROR("mv: size of A and y do not match");
00663     if (this->colsize() != x.size())
00664         HDNUM_ERROR("mv: size of A and x do not match");
00665     for (std::size_t i = 0; i < rowsize(); ++i) {
00666         for (std::size_t j = 0; j < colsize(); ++j)
00667             y[i] += s * (*this)(i, j) * x[j];
00668     }
00669 }
00670
00719 void mm(const DenseMatrix<REAL>& A, const DenseMatrix<REAL>& B) {
00720     if (this->rowsize() != A.rowsize())
00721         HDNUM_ERROR("mm: size incompatible");
00722     if (this->colsize() != B.colsize())
00723         HDNUM_ERROR("mm: size incompatible");
00724     if (A.colsize() != B.rowsize()) HDNUM_ERROR("mm: size incompatible");
00725
00726     for (std::size_t i = 0; i < rowsize(); i++)
00727         for (std::size_t j = 0; j < colsize(); j++) {
00728             (*this)(i, j) = 0;
00729             for (std::size_t k = 0; k < A.colsize(); k++)
00730                 (*this)(i, j) += A(i, k) * B(k, j);
00731         }
00732 }
00733
00787 void umm(const DenseMatrix<REAL>& A, const DenseMatrix<REAL>& B) {
00788     if (this->rowsize() != A.rowsize())
00789         HDNUM_ERROR("mm: size incompatible");
00790     if (this->colsize() != B.colsize())
00791         HDNUM_ERROR("mm: size incompatible");
00792     if (A.colsize() != B.rowsize()) HDNUM_ERROR("mm: size incompatible");
00793
00794     for (std::size_t i = 0; i < rowsize(); i++)
00795         for (std::size_t j = 0; j < colsize(); j++)
00796             for (std::size_t k = 0; k < A.colsize(); k++)
00797                 (*this)(i, j) += A(i, k) * B(k, j);
00798 }
00799
00833 void sc(const Vector<REAL>& x, std::size_t k) {
00834     if (this->rowsize() != x.size()) HDNUM_ERROR("cc: size incompatible");
00835
00836     for (std::size_t i = 0; i < rowsize(); i++) (*this)(i, k) = x[i];
00837 }
00838
00874 void sr(const Vector<REAL>& x, std::size_t k) {
00875     if (this->colsize() != x.size()) HDNUM_ERROR("cc: size incompatible");
00876
00877     for (std::size_t i = 0; i < colsize(); i++) (*this)(k, i) = x[i];
00878 }
00879
00881 REAL norm_infty() const {
00882     REAL norm(0.0);
00883     for (std::size_t i = 0; i < rowsize(); i++) {
00884         REAL sum(0.0);
00885         for (std::size_t j = 0; j < colsize(); j++)
00886             sum += myabs((*this)(i, j));
00887         if (sum > norm) norm = sum;
00888     }
00889     return norm;
00890 }
00891
00893 REAL norm_1() const {
00894     REAL norm(0.0);
00895     for (std::size_t j = 0; j < colsize(); j++) {
00896         REAL sum(0.0);
00897         for (std::size_t i = 0; i < rowsize(); i++)
00898             sum += myabs((*this)(i, j));
00899         if (sum > norm) norm = sum;

```



```

00900     }
00901     return norm;
00902 }
00903
00949 Vector<REAL> operator*(const Vector<REAL>& x) const {
00950     assert(x.size() == colsize());
00951
00952     Vector<REAL> y(rowsize());
00953     for (std::size_t r = 0; r < rowsize(); ++r) {
00954         for (std::size_t c = 0; c < colsize(); ++c) {
00955             y[r] += at(r, c) * x[c];
00956         }
00957     }
00958     return y;
00959 }
00960
01003 DenseMatrix operator*(const DenseMatrix& x) const {
01004     assert(colsize() == x.rowsize());
01005
01006     const std::size_t out_rows = rowsize();
01007     const std::size_t out_cols = x.colsize();
01008     DenseMatrix y(out_rows, out_cols, 0.0);
01009     for (std::size_t r = 0; r < out_rows; ++r)
01010         for (std::size_t c = 0; c < out_cols; ++c)
01011             for (std::size_t i = 0; i < colsize(); ++i)
01012                 y(r, c) += at(r, i) * x(i, c);
01013
01014     return y;
01015 }
01016
01059 DenseMatrix operator+(const DenseMatrix& x) const {
01060     assert(colsize() == x.colsize());
01061     assert(rowsize() == x.rowsize());
01062
01063     const std::size_t out_rows = rowsize();
01064     const std::size_t out_cols = x.colsize();
01065     DenseMatrix y(out_rows, out_cols, 0.0);
01066     y = *this;
01067     y += x;
01068     return y;
01069 }
01070
01113 DenseMatrix operator-(const DenseMatrix& x) const {
01114     assert(colsize() == x.colsize());
01115     assert(rowsize() == x.rowsize());
01116
01117     const std::size_t out_rows = rowsize();
01118     const std::size_t out_cols = x.colsize();
01119     DenseMatrix y(out_rows, out_cols, 0.0);
01120     y = *this;
01121     y -= x;
01122     return y;
01123 }
01124 };
01125
01126 template <typename REAL>
01127 bool DenseMatrix<REAL>::bScientific = true;
01128 template <typename REAL>
01129 std::size_t DenseMatrix<REAL>::nIndexWidth = 10;
01130 template <typename REAL>
01131 std::size_t DenseMatrix<REAL>::nValueWidth = 10;
01132 template <typename REAL>
01133 std::size_t DenseMatrix<REAL>::nValuePrecision = 3;
01134
01158 template <typename REAL>
01159 inline std::ostream& operator<<(std::ostream& s, const DenseMatrix<REAL>& A) {
01160     s << std::endl;
01161     s << " " << std::setw(A.iwidth()) << " "
01162       << " ";
01163     for (typename DenseMatrix<REAL>::size_type j = 0; j < A.colsize(); ++j)
01164         s << std::setw(A.width()) << j << " ";
01165     s << std::endl;
01166
01167     for (typename DenseMatrix<REAL>::size_type i = 0; i < A.rowsize(); ++i) {
01168         s << " " << std::setw(A.iwidth()) << i << " ";
01169         for (typename DenseMatrix<REAL>::size_type j = 0; j < A.colsize();
01170              ++j) {
01171             if (A.scientific()) {
01172                 s << std::setw(A.width()) << std::scientific << std::showpoint
01173                   << std::setprecision(A.precision()) << A[i][j] << " ";
01174             } else {
01175                 s << std::setw(A.width()) << std::fixed << std::showpoint
01176                   << std::setprecision(A.precision()) << A[i][j] << " ";
01177             }
01178         }
01179         s << std::endl;
01180     }

```

```

01181     return s;
01182 }
01183
01190 template <typename REAL>
01191 inline void fill(DenseMatrix<REAL>& A, const REAL& t) {
01192     for (typename DenseMatrix<REAL>::size_type i = 0; i < A.rowsize(); ++i)
01193         for (typename DenseMatrix<REAL>::size_type j = 0; j < A.colsize(); ++j)
01194             A[i][j] = t;
01195 }
01196
01198 template <typename REAL>
01199 inline void zero(DenseMatrix<REAL>& A) {
01200     for (std::size_t i = 0; i < A.rowsize(); ++i)
01201         for (std::size_t j = 0; j < A.colsize(); ++j) A(i, j) = REAL(0);
01202 }
01203
01237 template <class T>
01238 inline void identity(DenseMatrix<T>& A) {
01239     for (typename DenseMatrix<T>::size_type i = 0; i < A.rowsize(); ++i)
01240         for (typename DenseMatrix<T>::size_type j = 0; j < A.colsize(); ++j)
01241             if (i == j)
01242                 A[i][i] = T(1);
01243             else
01244                 A[i][j] = T(0);
01245 }
01246
01282 template <typename REAL>
01283 inline void spd(DenseMatrix<REAL>& A) {
01284     if (A.rowsize() != A.colsize() || A.rowsize() == 0)
01285         HDNUM_ERROR("need square and nonempty matrix");
01286     for (std::size_t i = 0; i < A.rowsize(); ++i)
01287         for (std::size_t j = 0; j < A.colsize(); ++j)
01288             if (i == j)
01289                 A(i, i) = REAL(4.0);
01290             else
01291                 A(i, j) = -REAL(1.0) / ((i - j) * (i - j));
01292 }
01293
01343 template <typename REAL>
01344 inline void vandermonde(DenseMatrix<REAL>& A, const Vector<REAL> x) {
01345     if (A.rowsize() != A.colsize() || A.rowsize() == 0)
01346         HDNUM_ERROR("need square and nonempty matrix");
01347     if (A.rowsize() != x.size()) HDNUM_ERROR("need A and x of same size");
01348     for (typename DenseMatrix<REAL>::size_type i = 0; i < A.rowsize(); ++i) {
01349         REAL p(1.0);
01350         for (typename DenseMatrix<REAL>::size_type j = 0; j < A.colsize();
01351             ++j) {
01352             A[i][j] = p;
01353             p *= x[i];
01354         }
01355     }
01356 }
01357
01359 template <typename REAL>
01360 inline void gnuplot(const std::string& fname, const DenseMatrix<REAL>& A) {
01361     std::ofstream f(fname.c_str(), std::ios::out);
01362     for (typename DenseMatrix<REAL>::size_type i = 0; i < A.rowsize(); ++i) {
01363         for (typename DenseMatrix<REAL>::size_type j = 0; j < A.colsize();
01364             ++j) {
01365             if (A.scientific()) {
01366                 f << std::setw(A.width()) << std::scientific << std::showpoint
01367                 << std::setprecision(A.precision()) << A[i][j];
01368             } else {
01369                 f << std::setw(A.width()) << std::fixed << std::showpoint
01370                 << std::setprecision(A.precision()) << A[i][j];
01371             }
01372         }
01373         f << std::endl;
01374     }
01375     f.close();
01376 }
01377
01407 template <typename REAL>
01408 inline void readMatrixFromFileDat(const std::string& filename,
01409     DenseMatrix<REAL>& A) {
01410     std::string buffer;
01411     std::ifstream fin(filename.c_str());
01412     std::size_t i = 0;
01413     std::size_t j = 0;
01414     if (fin.is_open()) {
01415         while (std::getline(fin, buffer)) {
01416             std::istringstream iss(buffer);
01417             hdnun::Vector<REAL> rowvector;
01418             while (iss) {
01419                 std::string sub;
01420                 iss >> sub;
01421                 // std::cout << " sub = " << sub.c_str() << " ";

```

```

01422         if (sub.length() > 0) {
01423             REAL a = atof(sub.c_str());
01424             // std::cout << std::fixed << std::setw(10) <<
01425             // std::setprecision(5) << a;
01426             rowvector.push_back(a);
01427         }
01428         j++;
01429     }
01430     if (rowvector.size() > 0) {
01431         A.addNewRow(rowvector);
01432         i++;
01433         // std::cout << std::endl;
01434     }
01435     }
01436     fin.close();
01437 } else {
01438     HDNUM_ERROR("Could not open file!");
01439 }
01440 }
01441
01475 template <typename REAL>
01476 inline void readMatrixFromFileMatrixMarket(const std::string& filename,
01477                                             DenseMatrix<REAL>& A) {
01478     std::string buffer;
01479     std::ifstream fin(filename.c_str());
01480     std::size_t i = 0;
01481     std::size_t j = 0;
01482     if (fin.is_open()) {
01483         // ignore all comments from the file (starting with %)
01484         while (fin.peek() == '%') fin.ignore(2048, '\n');
01485
01486         std::getline(fin, buffer);
01487         std::istringstream first_line(buffer);
01488         first_line >> i >> j;
01489         DenseMatrix<REAL> A_temp(i, j);
01490
01491         while (std::getline(fin, buffer)) {
01492             std::istringstream iss(buffer);
01493
01494             REAL value {};
01495             iss >> i >> j >> value;
01496             // i-1, j-1, because matrix market does not use zero based indexing
01497             A_temp(i - 1, j - 1) = value;
01498         }
01499         A = A_temp;
01500         fin.close();
01501     } else {
01502         HDNUM_ERROR("Could not open file! \"" + filename + "\"");
01503     }
01504 }
01505
01506 } // namespace hdnum
01507
01508 #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

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 Dune::Exception
<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.](#)

```

00001 #ifndef HDNUM_EXCEPTIONS_HH
00002 #define HDNUM_EXCEPTIONS_HH
00003
00004 #include <string>
00005 #include <sstream>
00006
00007 namespace hdnum {
00008
00035 class Exception {
00036 public:
00037     void message(const std::string &message);
00038     const std::string& what() const;
00039 private:
00040     std::string _message;
00041 };
00042
00043 inline void Exception::message(const std::string &message)
00044 {
00045     _message = message;
00046 }
00047
00048 inline const std::string& Exception::what() const
00049 {
00050     return _message;
00051 }
00052
00053 inline std::ostream& operator<<(std::ostream &stream, const Exception &e)
00054 {
00055     return stream << e.what();
00056 }
00057
00058 // the "format" the exception-type gets printed. __FILE__ and
00059 // __LINE__ are standard C-defines, the GNU cpp-infile claims that
00060 // C99 defines __func__ as well. __FUNCTION__ is a GNU-extension
00061 #ifndef HDNUM_DEVEL_MODE
00062 # define THROWSPEC(E) #E << " [" << __func__ << ":" << __FILE__ << ":" << __LINE__ << "]: "
00063 #else
00064 # define THROWSPEC(E) #E << ": "
00065 #endif
00066
00084 // this is the magic: use the usual do { ... } while (0) trick, create
00085 // the full message via a string stream and throw the created object
00086 #define HDNUM_THROW(E, m) do { E th_ex; std::ostringstream th_out;
00087     th_out << THROWSPEC(E) << m; th_ex.message(th_out.str()); throw th_ex; \
00088 } while (0)
00089
00099 class IOError : public Exception {};
00100
00109 class MathError : public Exception {};
00110
00122 class RangeError : public Exception {};
00123
00131 class NotImplemented : public Exception {};
00132
00139 class SystemError : public Exception {};
00140
00144 class OutOfMemoryError : public SystemError {};
00145

```

```

00149 class InvalidStateException : public Exception {};
00150
00153 class ErrorException : public Exception {};
00154
00155 // throw ErrorException with message
00156 #define HDNUM_ERROR(m) do { hdnum::ErrorException th__ex; std::ostringstream th__out; \
00157     th__out << THROWSPEC(hdnum::ErrorException) << m; \
00158     th__ex.message(th__out.str()); \
00159     std::cout << th__ex.what() << std::endl; \
00160     throw th__ex; \
00161 } while (0)
00162
00163 } // end namespace
00164
00165 #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_{ii}=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.](#)

```

00001 // -*- tab-width: 4; indent-tabs-mode: nil -*-
00002 #ifndef HDNUM_LR_HH
00003 #define HDNUM_LR_HH
00004
00005 #include "vector.hh"
00006 #include "densematrix.hh"
00007
00012 namespace hdnum {
00013
00015     template<class T>
00016     void lr (DenseMatrix<T>& A, Vector<std::size_t>& p)
00017     {
00018         if (A.rowsize()!=A.colsize() || A.rowsize()==0)
00019             HDNUM_ERROR("need square and nonempty matrix");
00020         if (A.rowsize()!=p.size())
00021             HDNUM_ERROR("permutation vector incompatible with matrix");
00022
00023         // transformation to upper triangular
00024         for (std::size_t k=0; k<A.rowsize()-1; ++k)
00025         {
00026             // find pivot element and exchange rows
00027             for (std::size_t r=k; r<A.rowsize(); ++r)
00028                 if (A[r][k]!=0)
00029                 {
00030                     p[k] = r; // store permutation in step k
00031                     if (r>k) // exchange complete row if r!=k
00032                         for (std::size_t j=0; j<A.colsize(); ++j)
00033                             {
00034                                 T temp(A[k][j]);
00035                                 A[k][j] = A[r][j];
00036                                 A[r][j] = temp;
00037                             }
00038                     break;
00039                 }
00040             if (A[k][k]==0) HDNUM_ERROR("matrix is singular");
00041
00042             // modification
00043             for (std::size_t i=k+1; i<A.rowsize(); ++i)
00044                 {
00045                     T qik(A[i][k]/A[k][k]);
00046                     A[i][k] = qik;
00047                     for (std::size_t j=k+1; j<A.colsize(); ++j)
00048                         A[i][j] -= qik * A[k][j];
00049                 }
00050         }
00051     }
00052
00054     template<class T>
00055     T abs (const T& t)
00056     {
00057         if (t<0.0)
00058             return -t;
00059         else
00060             return t;
00061     }
00062
00064     template<class T>
00065     void lr_partialpivot (DenseMatrix<T>& A, Vector<std::size_t>& p)
00066     {
00067         if (A.rowsize()!=A.colsize() || A.rowsize()==0)
00068             HDNUM_ERROR("need square and nonempty matrix");
00069         if (A.rowsize()!=p.size())
00070             HDNUM_ERROR("permutation vector incompatible with matrix");
00071
00072         // initialize permutation
00073         for (std::size_t k=0; k<A.rowsize(); ++k)
00074             p[k] = k;
00075
00076         // transformation to upper triangular
00077         for (std::size_t k=0; k<A.rowsize()-1; ++k)
00078         {
00079             // find pivot element

```

```

00080     for (std::size_t r=k+1; r<A.rowsize(); ++r)
00081         if (abs(A[r][k])>abs(A[k][k]))
00082             p[k] = r; // store permutation in step k
00083
00084     if (p[k]>k) // exchange complete row if r!=k
00085         for (std::size_t j=0; j<A.colsize(); ++j)
00086             {
00087                 T temp(A[k][j]);
00088                 A[k][j] = A[p[k]][j];
00089                 A[p[k]][j] = temp;
00090             }
00091
00092     if (A[k][k]==0) HDNUM_ERROR("matrix is singular");
00093
00094     // modification
00095     for (std::size_t i=k+1; i<A.rowsize(); ++i)
00096         {
00097             T qik(A[i][k]/A[k][k]);
00098             A[i][k] = qik;
00099             for (std::size_t j=k+1; j<A.colsize(); ++j)
00100                 A[i][j] -= qik * A[k][j];
00101         }
00102     }
00103 }
00104
00106 template<class T>
00107 void lr_fullpivot (DenseMatrix<T>& A, Vector<std::size_t>& p, Vector<std::size_t>& q)
00108 {
00109     if (A.rowsize()!=A.colsize() || A.rowsize()==0)
00110         HDNUM_ERROR("need square and nonempty matrix");
00111     if (A.rowsize()!=p.size())
00112         HDNUM_ERROR("permutation vector incompatible with matrix");
00113
00114     // initialize permutation
00115     for (std::size_t k=0; k<A.rowsize(); ++k)
00116         p[k] = q[k] = k;
00117
00118     // transformation to upper triangular
00119     for (std::size_t k=0; k<A.rowsize()-1; ++k)
00120         {
00121             // find pivot element
00122             for (std::size_t r=k; r<A.rowsize(); ++r)
00123                 for (std::size_t s=k; s<A.colsize(); ++s)
00124                     if (abs(A[r][s])>abs(A[k][k]))
00125                         {
00126                             p[k] = r; // store permutation in step k
00127                             q[k] = s;
00128                         }
00129
00130             if (p[k]>k) // exchange complete row if r!=k
00131                 for (std::size_t j=0; j<A.colsize(); ++j)
00132                     {
00133                         T temp(A[k][j]);
00134                         A[k][j] = A[p[k]][j];
00135                         A[p[k]][j] = temp;
00136                     }
00137             if (q[k]>k) // exchange complete column if s!=k
00138                 for (std::size_t i=0; i<A.rowsize(); ++i)
00139                     {
00140                         T temp(A[i][k]);
00141                         A[i][k] = A[i][q[k]];
00142                         A[i][q[k]] = temp;
00143                     }
00144
00145             if (std::abs(A[k][k])==0) HDNUM_ERROR("matrix is singular");
00146
00147             // modification
00148             for (std::size_t i=k+1; i<A.rowsize(); ++i)
00149                 {
00150                     T qik(A[i][k]/A[k][k]);
00151                     A[i][k] = qik;
00152                     for (std::size_t j=k+1; j<A.colsize(); ++j)
00153                         A[i][j] -= qik * A[k][j];
00154                 }
00155         }
00156 }
00157
00159 template<class T>
00160 void permute_forward (const Vector<std::size_t>& p, Vector<T>& b)
00161 {
00162     if (b.size()!=p.size())
00163         HDNUM_ERROR("permutation vector incompatible with rhs");
00164
00165     for (std::size_t k=0; k<b.size()-1; ++k)
00166         if (p[k]!=k) std::swap(b[k],b[p[k]]);
00167 }
00168

```



```

00170     template<class T>
00171     void permute_backward (const Vector<std::size_t>& q, Vector<T>& z)
00172     {
00173         if (z.size()!=q.size())
00174             HDNUM_ERROR("permutation vector incompatible with z");
00175
00176         for (int k=z.size()-2; k>=0; --k)
00177             if (q[k]!=std::size_t(k)) std::swap(z[k],z[q[k]]);
00178     }
00179
00180     template<class T>
00181     void row_equilibrate (DenseMatrix<T>& A, Vector<T>& s)
00182     {
00183         if (A.rowsize()*A.colsize()==0)
00184             HDNUM_ERROR("need nonempty matrix");
00185         if (A.rowsize()!=s.size())
00186             HDNUM_ERROR("scaling vector incompatible with matrix");
00187
00188         // equilibrate row sums
00189         for (std::size_t k=0; k<A.rowsize(); ++k)
00190         {
00191             s[k] = T(0.0);
00192             for (std::size_t j=0; j<A.colsize(); ++j)
00193                 s[k] += abs(A[k][j]);
00194             if (std::abs(s[k])==0) HDNUM_ERROR("row sum is zero");
00195             for (std::size_t j=0; j<A.colsize(); ++j)
00196                 A[k][j] /= s[k];
00197         }
00198     }
00199
00200     template<class T>
00201     void apply_equilibrate (Vector<T>& s, Vector<T>& b)
00202     {
00203         if (s.size()!=b.size())
00204             HDNUM_ERROR("s and b incompatible");
00205
00206         // equilibrate row sums
00207         for (std::size_t k=0; k<b.size(); ++k)
00208             b[k] /= s[k];
00209     }
00210
00211     template<class T>
00212     void solveL (const DenseMatrix<T>& A, Vector<T>& x, const Vector<T>& b)
00213     {
00214         if (A.rowsize()!=A.colsize() || A.rowsize()==0)
00215             HDNUM_ERROR("need square and nonempty matrix");
00216         if (A.rowsize()!=b.size())
00217             HDNUM_ERROR("right hand side incompatible with matrix");
00218
00219         for (std::size_t i=0; i<A.rowsize(); ++i)
00220         {
00221             T rhs(b[i]);
00222             for (std::size_t j=0; j<i; j++)
00223                 rhs -= A[i][j] * x[j];
00224             x[i] = rhs;
00225         }
00226     }
00227
00228     template<class T>
00229     void solveR (const DenseMatrix<T>& A, Vector<T>& x, const Vector<T>& b)
00230     {
00231         if (A.rowsize()!=A.colsize() || A.rowsize()==0)
00232             HDNUM_ERROR("need square and nonempty matrix");
00233         if (A.rowsize()!=b.size())
00234             HDNUM_ERROR("right hand side incompatible with matrix");
00235
00236         for (int i=A.rowsize()-1; i>=0; --i)
00237         {
00238             T rhs(b[i]);
00239             for (std::size_t j=i+1; j<A.colsize(); j++)
00240                 rhs -= A[i][j] * x[j];
00241             x[i] = rhs/A[i][i];
00242         }
00243     }
00244
00245     template<class T>
00246     void linsolve (DenseMatrix<T>& A, Vector<T>& x, Vector<T>& b)
00247     {
00248         if (A.rowsize()!=A.colsize() || A.rowsize()==0)
00249             HDNUM_ERROR("need square and nonempty matrix");
00250         if (A.rowsize()!=b.size())
00251             HDNUM_ERROR("right hand side incompatible with matrix");
00252
00253         Vector<T> s(x.size());
00254         Vector<std::size_t> p(x.size());
00255         Vector<std::size_t> q(x.size());
00256         row_equilibrate(A,s);
00257     }

```

```

00262     lr_fullpivot (A,p,q);
00263     apply_equilibrate(s,b);
00264     permute_forward(p,b);
00265     solveL(A,b,b);
00266     solveR(A,x,b);
00267     permute_backward(q,x);
00268 }
00269
00270 }
00271 #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.](#)

```

00001 // -*- tab-width: 4; indent-tabs-mode: nil -*-
00002 #ifndef HDNUM_NEWTON_HH
00003 #define HDNUM_NEWTON_HH
00004
00005 #include "lr.hh"
00006 #include <type_traits>
00007
00012 namespace hdnum {
00013
00020     template<class N>
00021     class SquareRootProblem
00022     {
00023     public:
00025         typedef std::size_t size_type;
00026
00028         typedef N number_type;
00029
00031         SquareRootProblem (number_type a_)
00032             : a(a_)
00033         {}
00034
00036         std::size_t size () const
00037         {
00038             return 1;
00039         }
00040
00042         void F (const Vector<N>& x, Vector<N>& result) const
00043         {
00044             result[0] = x[0]*x[0] - a;
00045         }
00046
00048         void F_x (const Vector<N>& x, DenseMatrix<N>& result) const
00049         {
00050             result[0][0] = number_type(2.0)*x[0];
00051         }
00052
00053     private:
00054         number_type a;
00055     };
00056
00057
00063     template<typename Lambda, typename Vec>
00064     class GenericNonlinearProblem
00065     {
00066     public:
00067         Lambda lambda; // lambda defining the problem "lambda(x)=0"
00068         size_t s;
00069         typename Vec::value_type eps;
00070
00072         typedef std::size_t size_type;
00073
00075         typedef typename Vec::value_type number_type;
00076
00078         GenericNonlinearProblem (const Lambda& l_, const Vec& x_, number_type eps_ = 1e-7)
00079             : lambda(l_), s(x_.size()), eps(eps_)
00080         {}
00081
00083         std::size_t size () const
00084         {
00085             return s;
00086         }
00087
00089         void F (const Vec& x, Vec& result) const
00090         {
00091             result = lambda(x);
00092         }
00093
00095         void F_x (const Vec& x, DenseMatrix<number_type>& result) const
00096         {

```

```

00097     Vec Fx(x.size());
00098     F(x,Fx);
00099     Vec z(x);
00100     Vec Fz(x.size());
00101
00102     // numerische Jacobimatrix
00103     for (int j=0; j<result.colsize(); ++j)
00104     {
00105         auto zj = z[j];
00106         auto dz = (1.0+abs(zj))*eps;
00107         z[j] += dz;
00108         F(z,Fz);
00109         for (int i=0; i<result.rowsize(); i++)
00110             result[i][j] = (Fz[i]-Fx[i])/dz;
00111         z[j] = zj;
00112     }
00113 }
00114 };
00115
00123 template<typename F, typename X>
00124 GenericNonlinearProblem<F,X> getNonlinearProblem (const F& f, const X& x, typename X::value_type eps
= 1e-7)
00125 {
00126     return GenericNonlinearProblem<F,X>(f,x,eps);
00127 }
00128
00135 class Newton
00136 {
00137     typedef std::size_t size_type;
00138
00139 public:
00141     Newton ()
00142         : maxit(25), linesearchsteps(10), verbosity(0),
00143           reduction(1e-14), abslimit(1e-30), converged(false)
00144     {}
00145
00147     void set_maxit (size_type n)
00148     {
00149         maxit = n;
00150     }
00151
00152     void set_sigma (double sigma_)
00153     {
00154     }
00155
00156
00158     void set_linesearchsteps (size_type n)
00159     {
00160         linesearchsteps = n;
00161     }
00162
00164     void set_verbosity (size_type n)
00165     {
00166         verbosity = n;
00167     }
00168
00170     void set_abslimit (double l)
00171     {
00172         abslimit = l;
00173     }
00174
00176     void set_reduction (double l)
00177     {
00178         reduction = l;
00179     }
00180
00182     template<class M>
00183     void solve (const M& model, Vector<typename M::number_type> & x) const
00184     {
00185         typedef typename M::number_type N;
00186         // In complex case, we still need to use real valued numbers for residual norms etc.
00187         using Real = typename std::conditional<std::is_same<std::complex<double>, N>::value, double,
N>::type;
00188         Vector<N> r(model.size()); // residual
00189         DenseMatrix<N> A(model.size(),model.size()); // Jacobian matrix
00190         Vector<N> y(model.size()); // temporary solution in line search
00191         Vector<N> z(model.size()); // solution of linear system
00192         Vector<N> s(model.size()); // scaling factors
00193         Vector<size_type> p(model.size()); // row permutations
00194         Vector<size_type> q(model.size()); // column permutations
00195
00196         model.F(x,r); // compute nonlinear residual
00197         Real R0(std::abs(norm(r))); // norm of initial residual
00198         Real R(R0); // current residual norm
00199         if (verbosity>=1)
00200         {
00201             std::cout << "Newton "

```

```

00202         << "   norm=" << std::scientific << std::showpoint
00203         << std::setprecision(4) << R0
00204         << std::endl;
00205     }
00206
00207     converged = false;
00208     for (size_type i=1; i<=maxit; i++)           // do Newton iterations
00209     {
00210         // check absolute size of residual
00211         if (R<=abslimit)
00212         {
00213             converged = true;
00214             return;
00215         }
00216
00217         // solve Jacobian system for update
00218         model.F_x(x,A);                          // compute Jacobian matrix
00219         row_equilibrate(A,s);                     // equilibrate rows
00220         lr_fullpivot(A,p,q);                      // LR decomposition of A
00221         z = N(0.0);                               // clear solution
00222         apply_equilibrate(s,r);                   // equilibration of right hand side
00223         permute_forward(p,r);                     // permutation of right hand side
00224         solvel(A,r,r);                            // forward substitution
00225         solveR(A,z,r);                            // backward substitution
00226         permute_backward(q,z);                    // backward permutation
00227
00228         // line search
00229         Real lambda(1.0);                          // start with lambda=1
00230         for (size_type k=0; k<linesearchsteps; k++)
00231         {
00232             y = x;
00233             y.update(-lambda,z);                   // y = x+lambda*z
00234             model.F(y,r);                          // r = F(y)
00235             Real newR(std::abs(norm(r)));           // compute norm
00236             if (verbosity>=3)
00237             {
00238                 std::cout << "   line search " << std::setw(2) << k
00239                 << " lambda=" << std::scientific << std::showpoint
00240                 << std::setprecision(4) << lambda
00241                 << " norm=" << std::scientific << std::showpoint
00242                 << std::setprecision(4) << newR
00243                 << " red=" << std::scientific << std::showpoint
00244                 << std::setprecision(4) << newR/R
00245                 << std::endl;
00246             }
00247             if (newR<(1.0-0.25*lambda)*R)           // check convergence
00248             {
00249                 if (verbosity>=2)
00250                 {
00251                     std::cout << " step" << std::setw(3) << i
00252                     << " norm=" << std::scientific << std::showpoint
00253                     << std::setprecision(4) << newR
00254                     << " red=" << std::scientific << std::showpoint
00255                     << std::setprecision(4) << newR/R
00256                     << std::endl;
00257                 }
00258                 x = y;
00259                 R = newR;
00260                 break;                               // continue with Newton loop
00261             }
00262             else lambda *= 0.5;                       // reduce damping factor
00263             if (k==linesearchsteps-1)
00264             {
00265                 if (verbosity>=3)
00266                     std::cout << "   line search not converged within " << linesearchsteps << " steps" <<
std::endl;
00267                 return;
00268             }
00269         }
00270
00271         // check convergence
00272         if (R<=reduction*R0)
00273         {
00274             if (verbosity>=1)
00275             {
00276                 std::cout << "Newton converged in " << i << " steps"
00277                 << " reduction=" << std::scientific << std::showpoint
00278                 << std::setprecision(4) << R/R0
00279                 << std::endl;
00280             }
00281             iterations_taken = i;
00282             converged = true;
00283             return;
00284         }
00285         if (i==maxit)
00286         {
00287             iterations_taken = i;

```

```

00288         if (verbosity>=1)
00289             std::cout << "Newton not converged within " << maxit << " iterations" << std::endl;
00290     }
00291 }
00292 }
00293
00294 bool has_converged () const
00295 {
00296     return converged;
00297 }
00298 size_type iterations() const {
00299     return iterations_taken;
00300 }
00301 }
00302
00303 private:
00304     size_type maxit;
00305     mutable size_type iterations_taken = -1;
00306     size_type linesearchsteps;
00307     size_type verbosity;
00308     double reduction;
00309     double abslimit;
00310     mutable bool converged;
00311 };
00312
00313
00314
00315
00323 class Banach
00324 {
00325     typedef std::size_t size_type;
00326
00327 public:
00329     Banach ()
00330         : maxit(25), linesearchsteps(10), verbosity(0),
00331           reduction(1e-14), abslimit(1e-30), sigma(1.0), converged(false)
00332     {}
00333
00335     void set_maxit (size_type n)
00336     {
00337         maxit = n;
00338     }
00339
00341     void set_sigma (double sigma_)
00342     {
00343         sigma = sigma_;
00344     }
00345
00347     void set_linesearchsteps (size_type n)
00348     {
00349         linesearchsteps = n;
00350     }
00351
00353     void set_verbosity (size_type n)
00354     {
00355         verbosity = n;
00356     }
00357
00359     void set_abslimit (double l)
00360     {
00361         abslimit = l;
00362     }
00363
00365     void set_reduction (double l)
00366     {
00367         reduction = l;
00368     }
00369
00371     template<class M>
00372     void solve (const M& model, Vector<typename M::number_type>& x) const
00373     {
00374         typedef typename M::number_type N;
00375         Vector<N> r(model.size()); // residual
00376         Vector<N> y(model.size()); // temporary solution in line search
00377
00378         model.F(x,r); // compute nonlinear residual
00379         N R0(norm(r)); // norm of initial residual
00380         N R(R0); // current residual norm
00381         if (verbosity>=1)
00382         {
00383             std::cout << "Banach "
00384                 << " norm=" << std::scientific << std::showpoint
00385                 << std::setprecision(4) << R0
00386                 << std::endl;
00387         }
00388
00389         converged = false;

```

```

00390     for (size_type i=1; i<=maxit; i++)                // do iterations
00391     {
00392         // check absolute size of residual
00393         if (R<=abslimit)
00394         {
00395             converged = true;
00396             return;
00397         }
00398
00399         // next iterate
00400         y = x;
00401         y.update(-sigma,r);                          // y = x+lambda*z
00402         model.F(y,r);                                 // r = F(y)
00403         N newR(norm(r));                              // compute norm
00404         if (verbosity>=2)
00405         {
00406             std::cout << " " << std::setw(3) << i
00407             << " norm=" << std::scientific << std::showpoint
00408             << std::setprecision(4) << newR
00409             << " red=" << std::scientific << std::showpoint
00410             << std::setprecision(4) << newR/R
00411             << std::endl;
00412         }
00413         x = y;                                        // accept new iterate
00414         R = newR;                                    // remember new norm
00415
00416         // check convergence
00417         if (R<=reduction*R0 || R<=abslimit)
00418         {
00419             if (verbosity>=1)
00420             {
00421                 std::cout << "Banach converged in " << i << " steps"
00422                 << " reduction=" << std::scientific << std::showpoint
00423                 << std::setprecision(4) << R/R0
00424                 << std::endl;
00425             }
00426             converged = true;
00427             return;
00428         }
00429     }
00430 }
00431
00432 bool has_converged () const
00433 {
00434     return converged;
00435 }
00436
00437 private:
00438     size_type maxit;
00439     size_type linesearchsteps;
00440     size_type verbosity;
00441     double reduction;
00442     double abslimit;
00443     double sigma;
00444     mutable bool converged;
00445 };
00446
00447 } // namespace hdnun
00448
00449 #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.](#)

```

00001 // -*- tab-width: 4; indent-tabs-mode: nil -*-
00002 #ifndef HDNUM_ODE_HH
00003 #define HDNUM_ODE_HH
00004
00005 #include<vector>
00006 #include "newton.hh"
00007
00012 namespace hdnum {
00013
00022     template<class M>
00023     class EE
00024     {
00025     public:
00027         typedef typename M::size_type size_type;
00028
00030         typedef typename M::time_type time_type;
00031
00033         typedef typename M::number_type number_type;
00034
00036         EE (const M& model_)

```



```

00037     : model(model_), u(model.size()), f(model.size())
00038     {
00039     model.initialize(t,u);
00040     dt = 0.1;
00041     }
00042
00044 void set_dt (time_type dt_)
00045 {
00046     dt = dt_;
00047 }
00048
00050 void step ()
00051 {
00052     model.f(t,u,f); // evaluate model
00053     u.update(dt,f); // advance state
00054     t += dt; // advance time
00055 }
00056
00058 void set_state (time_type t_, const Vector<number_type>& u_)
00059 {
00060     t = t_;
00061     u = u_;
00062 }
00063
00065 const Vector<number_type>& get_state () const
00066 {
00067     return u;
00068 }
00069
00071 time_type get_time () const
00072 {
00073     return t;
00074 }
00075
00077 time_type get_dt () const
00078 {
00079     return dt;
00080 }
00081
00083 size_type get_order () const
00084 {
00085     return 1;
00086 }
00087
00088 private:
00089     const M& model;
00090     time_type t, dt;
00091     Vector<number_type> u;
00092     Vector<number_type> f;
00093 };
00094
00103 template<class M>
00104 class ModifiedEuler
00105 {
00106 public:
00108     typedef typename M::size_type size_type;
00109
00111     typedef typename M::time_type time_type;
00112
00114     typedef typename M::number_type number_type;
00115
00117     ModifiedEuler (const M& model_)
00118     : model(model_), u(model.size()), w(model.size()), k1(model.size()), k2(model.size())
00119     {
00120         c2 = 0.5;
00121         a21 = 0.5;
00122         b2 = 1.0;
00123         model.initialize(t,u);
00124         dt = 0.1;
00125     }
00126
00128 void set_dt (time_type dt_)
00129 {
00130     dt = dt_;
00131 }
00132
00134 void step ()
00135 {
00136     // stage 1
00137     model.f(t,u,k1);
00138
00139     // stage 2
00140     w = u;
00141     w.update(dt*a21,k1);
00142     model.f(t+c2*dt,w,k2);
00143
00144     // final

```

```

00145     u.update(dt*b2,k2);
00146     t += dt;
00147 }
00148
00150 void set_state (time_type t_, const Vector<number_type>& u_)
00151 {
00152     t = t_;
00153     u = u_;
00154 }
00155
00157 const Vector<number_type>& get_state () const
00158 {
00159     return u;
00160 }
00161
00163 time_type get_time () const
00164 {
00165     return t;
00166 }
00167
00169 time_type get_dt () const
00170 {
00171     return dt;
00172 }
00173
00175 size_type get_order () const
00176 {
00177     return 2;
00178 }
00179
00180 private:
00181     const M& model;
00182     time_type t, dt;
00183     time_type c2,a21,b2;
00184     Vector<number_type> u,w;
00185     Vector<number_type> k1,k2;
00186 };
00187
00188
00197 template<class M>
00198 class Heun2
00199 {
00200 public:
00202     typedef typename M::size_type size_type;
00203
00205     typedef typename M::time_type time_type;
00206
00208     typedef typename M::number_type number_type;
00209
00211     Heun2 (const M& model_)
00212         : model(model_), u(model.size()), w(model.size()), k1(model.size()), k2(model.size())
00213     {
00214         c2 = 1.0;
00215         a21 = 1.0;
00216         b1 = 0.5;
00217         b2 = 0.5;
00218         model.initialize(t,u);
00219         dt = 0.1;
00220     }
00221
00223     void set_dt (time_type dt_)
00224     {
00225         dt = dt_;
00226     }
00227
00229     void step ()
00230     {
00231         // stage 1
00232         model.f(t,u,k1);
00233
00234         // stage 2
00235         w = u;
00236         w.update(dt*a21,k1);
00237         model.f(t+c2*dt,w,k2);
00238
00239         // final
00240         u.update(dt*b1,k1);
00241         u.update(dt*b2,k2);
00242         t += dt;
00243     }
00244
00246     void set_state (time_type t_, const Vector<number_type>& u_)
00247     {
00248         t = t_;
00249         u = u_;
00250     }
00251

```

```

00253     const Vector<number_type>& get_state () const
00254     {
00255         return u;
00256     }
00257
00259     time_type get_time () const
00260     {
00261         return t;
00262     }
00263
00265     time_type get_dt () const
00266     {
00267         return dt;
00268     }
00269
00271     size_type get_order () const
00272     {
00273         return 2;
00274     }
00275
00276 private:
00277     const M& model;
00278     time_type t, dt;
00279     time_type c2,a21,b1,b2;
00280     Vector<number_type> u,w;
00281     Vector<number_type> k1,k2;
00282 };
00283
00284
00293 template<class M>
00294 class Heun3
00295 {
00296 public:
00298     typedef typename M::size_type size_type;
00299
00301     typedef typename M::time_type time_type;
00302
00304     typedef typename M::number_type number_type;
00305
00307     Heun3 (const M& model_)
00308         : model(model_), u(model.size()), w(model.size()), k1(model.size()),
00309           k2(model.size()), k3(model.size())
00310     {
00311         c2 = time_type(1.0)/time_type(3.0);
00312         c3 = time_type(2.0)/time_type(3.0);
00313         a21 = time_type(1.0)/time_type(3.0);
00314         a32 = time_type(2.0)/time_type(3.0);
00315         b1 = 0.25;
00316         b2 = 0.0;
00317         b3 = 0.75;
00318         model.initialize(t,u);
00319         dt = 0.1;
00320     }
00321
00323     void set_dt (time_type dt_)
00324     {
00325         dt = dt_;
00326     }
00327
00329     void step ()
00330     {
00331         // stage 1
00332         model.f(t,u,k1);
00333
00334         // stage 2
00335         w = u;
00336         w.update(dt*a21,k1);
00337         model.f(t+c2*dt,w,k2);
00338
00339         // stage 3
00340         w = u;
00341         w.update(dt*a32,k2);
00342         model.f(t+c3*dt,w,k3);
00343
00344         // final
00345         u.update(dt*b1,k1);
00346         u.update(dt*b3,k3);
00347         t += dt;
00348     }
00349
00351     void set_state (time_type t_, const Vector<number_type>& u_)
00352     {
00353         t = t_;
00354         u = u_;
00355     }
00356
00358     const Vector<number_type>& get_state () const

```

```

00359     {
00360         return u;
00361     }
00362
00363     time_type get_time () const
00364     {
00365         return t;
00366     }
00367
00368     time_type get_dt () const
00369     {
00370         return dt;
00371     }
00372
00373     size_type get_order () const
00374     {
00375         return 3;
00376     }
00377
00378 private:
00379     const M& model;
00380     time_type t, dt;
00381     time_type c2, c3, a21, a31, a32, b1, b2, b3;
00382     Vector<number_type> u, w;
00383     Vector<number_type> k1, k2, k3;
00384 };
00385
00386 template<class M>
00387 class Kutta3
00388 {
00389 public:
00390     typedef typename M::size_type size_type;
00391
00392     typedef typename M::time_type time_type;
00393
00394     typedef typename M::number_type number_type;
00395
00396     Kutta3 (const M& model_)
00397         : model(model_), u(model.size()), w(model.size()), k1(model.size()),
00398           k2(model.size()), k3(model.size())
00399     {
00400         c2 = 0.5;
00401         c3 = 1.0;
00402         a21 = 0.5;
00403         a31 = -1.0;
00404         a32 = 2.0;
00405         b1 = time_type(1.0)/time_type(6.0);
00406         b2 = time_type(4.0)/time_type(6.0);
00407         b3 = time_type(1.0)/time_type(6.0);
00408         model.initialize(t, u);
00409         dt = 0.1;
00410     }
00411
00412     void set_dt (time_type dt_)
00413     {
00414         dt = dt_;
00415     }
00416
00417     void step ()
00418     {
00419         // stage 1
00420         model.f(t, u, k1);
00421
00422         // stage 2
00423         w = u;
00424         w.update(dt*a21, k1);
00425         model.f(t+c2*dt, w, k2);
00426
00427         // stage 3
00428         w = u;
00429         w.update(dt*a31, k1);
00430         w.update(dt*a32, k2);
00431         model.f(t+c3*dt, w, k3);
00432
00433         // final
00434         u.update(dt*b1, k1);
00435         u.update(dt*b2, k2);
00436         u.update(dt*b3, k3);
00437         t += dt;
00438     }
00439
00440     void set_state (time_type t_, const Vector<number_type>& u_)
00441     {
00442         t = t_;
00443         u = u_;
00444     }
00445
00446
00447
00448
00449
00450
00451
00452
00453
00454
00455
00456
00457
00458
00459
00460
00461
00462
00463

```

```

00465     const Vector<number_type>& get_state () const
00466     {
00467         return u;
00468     }
00469
00471     time_type get_time () const
00472     {
00473         return t;
00474     }
00475
00477     time_type get_dt () const
00478     {
00479         return dt;
00480     }
00481
00483     size_type get_order () const
00484     {
00485         return 3;
00486     }
00487
00488 private:
00489     const M& model;
00490     time_type t, dt;
00491     time_type c2,c3,a21,a31,a32,b1,b2,b3;
00492     Vector<number_type> u,w;
00493     Vector<number_type> k1,k2,k3;
00494 };
00495
00504 template<class M>
00505 class RungeKutta4
00506 {
00507 public:
00509     typedef typename M::size_type size_type;
00510
00512     typedef typename M::time_type time_type;
00513
00515     typedef typename M::number_type number_type;
00516
00518     RungeKutta4 (const M& model_)
00519         : model(model_), u(model.size()), w(model.size()), k1(model.size()),
00520           k2(model.size()), k3(model.size()), k4(model.size())
00521     {
00522         c2 = 0.5;
00523         c3 = 0.5;
00524         c4 = 1.0;
00525         a21 = 0.5;
00526         a32 = 0.5;
00527         a43 = 1.0;
00528         b1 = time_type(1.0)/time_type(6.0);
00529         b2 = time_type(2.0)/time_type(6.0);
00530         b3 = time_type(2.0)/time_type(6.0);
00531         b4 = time_type(1.0)/time_type(6.0);
00532         model.initialize(t,u);
00533         dt = 0.1;
00534     }
00535
00537     void set_dt (time_type dt_)
00538     {
00539         dt = dt_;
00540     }
00541
00543     void step ()
00544     {
00545         // stage 1
00546         model.f(t,u,k1);
00547
00548         // stage 2
00549         w = u;
00550         w.update(dt*a21,k1);
00551         model.f(t+c2*dt,w,k2);
00552
00553         // stage 3
00554         w = u;
00555         w.update(dt*a32,k2);
00556         model.f(t+c3*dt,w,k3);
00557
00558         // stage 4
00559         w = u;
00560         w.update(dt*a43,k3);
00561         model.f(t+c4*dt,w,k4);
00562
00563         // final
00564         u.update(dt*b1,k1);
00565         u.update(dt*b2,k2);
00566         u.update(dt*b3,k3);
00567         u.update(dt*b4,k4);
00568         t += dt;

```

```

00569     }
00570
00572     void set_state (time_type t_, const Vector<number_type>& u_)
00573     {
00574         t = t_;
00575         u = u_;
00576     }
00577
00579     const Vector<number_type>& get_state () const
00580     {
00581         return u;
00582     }
00583
00585     time_type get_time () const
00586     {
00587         return t;
00588     }
00589
00591     time_type get_dt () const
00592     {
00593         return dt;
00594     }
00595
00597     size_type get_order () const
00598     {
00599         return 4;
00600     }
00601
00602 private:
00603     const M& model;
00604     time_type t, dt;
00605     time_type c2, c3, c4, a21, a32, a43, b1, b2, b3, b4;
00606     Vector<number_type> u, w;
00607     Vector<number_type> k1, k2, k3, k4;
00608 };
00609
00614 template<class M>
00615 class RKF45
00616 {
00617 public:
00619     typedef typename M::size_type size_type;
00620
00622     typedef typename M::time_type time_type;
00623
00625     typedef typename M::number_type number_type;
00626
00628     RKF45 (const M& model_)
00629     : model(model_), u(model.size()), w(model.size()), ww(model.size()), k1(model.size()),
00630       k2(model.size()), k3(model.size()), k4(model.size()), k5(model.size()), k6(model.size()),
00631       steps(0), rejected(0)
00632     {
00633         TOL = time_type(0.0001);
00634         rho = time_type(0.8);
00635         alpha = time_type(0.25);
00636         beta = time_type(4.0);
00637         dt_min = 1E-12;
00638
00639         c2 = time_type(1.0)/time_type(4.0);
00640         c3 = time_type(3.0)/time_type(8.0);
00641         c4 = time_type(12.0)/time_type(13.0);
00642         c5 = time_type(1.0);
00643         c6 = time_type(1.0)/time_type(2.0);
00644
00645         a21 = time_type(1.0)/time_type(4.0);
00646
00647         a31 = time_type(3.0)/time_type(32.0);
00648         a32 = time_type(9.0)/time_type(32.0);
00649
00650         a41 = time_type(1932.0)/time_type(2197.0);
00651         a42 = time_type(-7200.0)/time_type(2197.0);
00652         a43 = time_type(7296.0)/time_type(2197.0);
00653
00654         a51 = time_type(439.0)/time_type(216.0);
00655         a52 = time_type(-8.0);
00656         a53 = time_type(3680.0)/time_type(513.0);
00657         a54 = time_type(-845.0)/time_type(4104.0);
00658
00659         a61 = time_type(-8.0)/time_type(27.0);
00660         a62 = time_type(2.0);
00661         a63 = time_type(-3544.0)/time_type(2565.0);
00662         a64 = time_type(1859.0)/time_type(4104.0);
00663         a65 = time_type(-11.0)/time_type(40.0);
00664
00665         b1 = time_type(25.0)/time_type(216.0);
00666         b2 = time_type(0.0);
00667         b3 = time_type(1408.0)/time_type(2565.0);
00668         b4 = time_type(2197.0)/time_type(4104.0);

```

```

00669     b5 = time_type(-1.0)/time_type(5.0);
00670
00671     bb1 = time_type(16.0)/time_type(135.0);
00672     bb2 = time_type(0.0);
00673     bb3 = time_type(6656.0)/time_type(12825.0);
00674     bb4 = time_type(28561.0)/time_type(56430.0);
00675     bb5 = time_type(-9.0)/time_type(50.0);
00676     bb6 = time_type(2.0)/time_type(55.0);
00677
00678     model.initialize(t,u);
00679     dt = 0.1;
00680 }
00681
00682 void set_dt (time_type dt_)
00683 {
00684     dt = dt_;
00685 }
00686
00687 void set_TOL (time_type TOL_)
00688 {
00689     TOL = TOL_;
00690 }
00691
00692 void step ()
00693 {
00694     steps++;
00695
00696     // stage 1
00697     model.f(t,u,k1);
00698
00699     // stage 2
00700     w = u;
00701     w.update(dt*a21,k1);
00702     model.f(t+c2*dt,w,k2);
00703
00704     // stage 3
00705     w = u;
00706     w.update(dt*a31,k1);
00707     w.update(dt*a32,k2);
00708     model.f(t+c3*dt,w,k3);
00709
00710     // stage 4
00711     w = u;
00712     w.update(dt*a41,k1);
00713     w.update(dt*a42,k2);
00714     w.update(dt*a43,k3);
00715     model.f(t+c4*dt,w,k4);
00716
00717     // stage 5
00718     w = u;
00719     w.update(dt*a51,k1);
00720     w.update(dt*a52,k2);
00721     w.update(dt*a53,k3);
00722     w.update(dt*a54,k4);
00723     model.f(t+c5*dt,w,k5);
00724
00725     // stage 6
00726     w = u;
00727     w.update(dt*a61,k1);
00728     w.update(dt*a62,k2);
00729     w.update(dt*a63,k3);
00730     w.update(dt*a64,k4);
00731     w.update(dt*a65,k5);
00732     model.f(t+c6*dt,w,k6);
00733
00734     // compute order 4 approximation
00735     w = u;
00736     w.update(dt*b1,k1);
00737     w.update(dt*b2,k2);
00738     w.update(dt*b3,k3);
00739     w.update(dt*b4,k4);
00740     w.update(dt*b5,k5);
00741
00742     // compute order 5 approximation
00743     ww = u;
00744     ww.update(dt*bb1,k1);
00745     ww.update(dt*bb2,k2);
00746     ww.update(dt*bb3,k3);
00747     ww.update(dt*bb4,k4);
00748     ww.update(dt*bb5,k5);
00749     ww.update(dt*bb6,k6);
00750
00751     // estimate local error
00752     w -= ww;
00753     time_type error(norm(w));
00754     time_type dt_opt(dt*pow(rho*TOL/error,0.2));
00755     dt_opt = std::min(beta*dt,std::max(alpha*dt,dt_opt));
00756

```

```

00759     //std::cout << "est. error=" << error << " dt_opt=" << dt_opt << std::endl;
00760
00761     if (error<=TOL)
00762     {
00763         t += dt;
00764         u = ww;
00765         dt = dt_opt;
00766     }
00767     else
00768     {
00769         rejected++;
00770         dt = dt_opt;
00771         if (dt>dt_min) step();
00772     }
00773 }
00774
00775 const Vector<number_type>& get_state () const
00776 {
00777     return u;
00778 }
00779
00780 time_type get_time () const
00781 {
00782     return t;
00783 }
00784
00785 time_type get_dt () const
00786 {
00787     return dt;
00788 }
00789
00790 size_type get_order () const
00791 {
00792     return 5;
00793 }
00794
00795 void get_info () const
00796 {
00797     std::cout << "RE: steps=" << steps << " rejected=" << rejected << std::endl;
00798 }
00799
00800 private:
00801 const M& model;
00802 time_type t, dt;
00803 time_type TOL, rho, alpha, beta, dt_min;
00804 time_type c2, c3, c4, c5, c6;
00805 time_type a21, a31, a32, a41, a42, a43, a51, a52, a53, a54, a61, a62, a63, a64, a65;
00806 time_type b1, b2, b3, b4, b5; // 4th order
00807 time_type bb1, bb2, bb3, bb4, bb5, bb6; // 5th order
00808 Vector<number_type> u, w, ww;
00809 Vector<number_type> k1, k2, k3, k4, k5, k6;
00810 mutable size_type steps, rejected;
00811 };
00812
00813 template<class M, class S>
00814 class RE
00815 {
00816 public:
00817     typedef typename M::size_type size_type;
00818
00819     typedef typename M::time_type time_type;
00820
00821     typedef typename M::number_type number_type;
00822
00823     RE (const M& model_, S& solver_)
00824     : model(model_), solver(solver_), u(model.size()),
00825       wlow(model.size()), which(model.size()), ww(model.size()),
00826       steps(0), rejected(0)
00827     {
00828         model.initialize(t, u); // initialize state
00829         dt = 0.1; // set initial time step
00830         two_power_m = 1.0;
00831         for (size_type i=0; i<solver.get_order(); i++)
00832             two_power_m *= 2.0;
00833         TOL = time_type(0.0001);
00834         rho = time_type(0.8);
00835         alpha = time_type(0.25);
00836         beta = time_type(4.0);
00837         dt_min = 1E-12;
00838     }
00839
00840     void set_dt (time_type dt_)
00841     {
00842         dt = dt_;
00843     }
00844 }

```



```

00862 void set_TOL (time_type TOL_)
00863 {
00864     TOL = TOL_;
00865 }
00866
00868 void step ()
00869 {
00870     // count steps done
00871     steps++;
00872
00873     // do 1 step with 2*dt
00874     time_type H(2.0*dt);
00875     solver.set_state(t,u);
00876     solver.set_dt(H);
00877     solver.step();
00878     wlow = solver.get_state();
00879
00880     // do 2 steps with dt
00881     solver.set_state(t,u);
00882     solver.set_dt(dt);
00883     solver.step();
00884     solver.step();
00885     whigh = solver.get_state();
00886
00887     // estimate local error
00888     ww = wlow;
00889     ww -= whigh;
00890     time_type error(norm(ww)/(pow(H,1.0+solver.get_order())*(1.0-1.0/two_power_m)));
00891     time_type dt_opt(pow(rho*TOL/error,1.0/((time_type)solver.get_order())));
00892     dt_opt = std::min(beta*dt, std::max(alpha*dt, dt_opt));
00893     //std::cout << "est. error=" << error << " dt_opt=" << dt_opt << std::endl;
00894
00895     if (dt<=dt_opt)
00896     {
00897         t += H;
00898         u = whigh;
00899         u *= two_power_m;
00900         u -= wlow;
00901         u /= two_power_m-1.0;
00902         dt = dt_opt;
00903     }
00904     else
00905     {
00906         rejected++;
00907         dt = dt_opt;
00908         if (dt>dt_min) step();
00909     }
00910 }
00911
00913 const Vector<number_type>& get_state () const
00914 {
00915     return u;
00916 }
00917
00919 time_type get_time () const
00920 {
00921     return t;
00922 }
00923
00925 time_type get_dt () const
00926 {
00927     return dt;
00928 }
00929
00931 size_type get_order () const
00932 {
00933     return solver.get_order()+1;
00934 }
00935
00937 void get_info () const
00938 {
00939     std::cout << "RE: steps=" << steps << " rejected=" << rejected << std::endl;
00940 }
00941
00942 private:
00943     const M& model;
00944     S& solver;
00945     time_type t, dt;
00946     time_type two_power_m;
00947     Vector<number_type> u,wlow,whigh,ww;
00948     time_type TOL,rho,alpha,beta,dt_min;
00949     mutable size_type steps, rejected;
00950 };
00951
00952
00962 template<class M, class S>
00963 class IE

```

```

00964 {
00966 // h_n f(t_n, y_n) - y_n + y_{n-1} = 0
00967 class NonlinearProblem
00968 {
00969 public:
00971     typedef typename M::size_type size_type;
00972
00974     typedef typename M::number_type number_type;
00975
00977     NonlinearProblem (const M& model_, const Vector<number_type>& yold_,
00978                       typename M::time_type tnew_, typename M::time_type dt_)
00979         : model(model_), yold(yold_), tnew(tnew_), dt(dt_)
00980     {}
00981
00983     std::size_t size () const
00984     {
00985         return model.size();
00986     }
00987
00989     void F (const Vector<number_type>& x, Vector<number_type>& result) const
00990     {
00991         model.f(tnew,x,result);
00992         result *= dt;
00993         result -= x;
00994         result += yold;
00995     }
00996
00998     void F_x (const Vector<number_type>& x, DenseMatrix<number_type>& result) const
00999     {
01000         model.f_x(tnew,x,result);
01001         result *= dt;
01002         for (size_type i=0; i<model.size(); i++) result[i][i] -= number_type(1.0);
01003     }
01004
01005     void set_tnew_dt (typename M::time_type tnew_, typename M::time_type dt_)
01006     {
01007         tnew = tnew_;
01008         dt = dt_;
01009     }
01010
01011 private:
01012     const M& model;
01013     const Vector<number_type>& yold;
01014     typename M::time_type tnew;
01015     typename M::time_type dt;
01016 };
01017
01018 public:
01020     typedef typename M::size_type size_type;
01021
01023     typedef typename M::time_type time_type;
01024
01026     typedef typename M::number_type number_type;
01027
01029     IE (const M& model_, const S& newton_)
01030         : verbosity(0), model(model_), newton(newton_), u(model.size()), unew(model.size())
01031     {
01032         model.initialize(t,u);
01033         dt = dtmax = 0.1;
01034     }
01035
01037     void set_dt (time_type dt_)
01038     {
01039         dt = dtmax = dt_;
01040     }
01041
01043     void set_verbosity (size_type verbosity_)
01044     {
01045         verbosity = verbosity_;
01046     }
01047
01049     void step ()
01050     {
01051         if (verbosity>=2)
01052             std::cout << "IE: step" << " t=" << t << " dt=" << dt << std::endl;
01053         NonlinearProblem nlp(model,u,t+dt,dt);
01054         bool reduced = false;
01055         error = false;
01056         while (1)
01057         {
01058             unew = u;
01059             newton.solve(nlp,unew);
01060             if (newton.has_converged())
01061             {
01062                 u = unew;
01063                 t += dt;
01064                 if (!reduced && dt<dtmax-1e-13)

```

```

01065         {
01066             dt = std::min(2.0*dt,dtmax);
01067             if (verbosity>0)
01068                 std::cout << "IE: increasing time step to " << dt << std::endl;
01069         }
01070         return;
01071     }
01072     else
01073     {
01074         if (dt<1e-12)
01075         {
01076             HDNUM_ERROR("time step too small in implicit Euler");
01077             error = true;
01078             break;
01079         }
01080         dt *= 0.5;
01081         reduced = true;
01082         nlp.set_tnew_dt(t+dt,dt);
01083         if (verbosity>0) std::cout << "IE: reducing time step to " << dt << std::endl;
01084     }
01085 }
01086 }
01087
01088 bool get_error () const
01089 {
01090     return error;
01091 }
01092
01093 void set_state (time_type t_, const Vector<number_type>& u_)
01094 {
01095     t = t_;
01096     u = u_;
01097 }
01098
01099 const Vector<number_type>& get_state () const
01100 {
01101     return u;
01102 }
01103
01104 time_type get_time () const
01105 {
01106     return t;
01107 }
01108
01109 time_type get_dt () const
01110 {
01111     return dt;
01112 }
01113
01114 size_type get_order () const
01115 {
01116     return 1;
01117 }
01118
01119 void get_info () const
01120 {
01121 }
01122
01123 private:
01124     size_type verbosity;
01125     const M& model;
01126     const S& newton;
01127     time_type t, dt, dtmax;
01128     number_type reduction;
01129     size_type linesearchsteps;
01130     Vector<number_type> u;
01131     Vector<number_type> unew;
01132     mutable bool error;
01133 };
01134
01135 template<class M, class S>
01136 class DIRK
01137 {
01138     public:
01139         typedef typename M::size_type size_type;
01140         typedef typename M::time_type time_type;
01141         typedef typename M::number_type number_type;
01142         typedef DenseMatrix<number_type> ButcherTableau;
01143     private:
01144         static ButcherTableau initTableau(const std::string method)
01145         {

```

```

01175     if(method.find("Implicit Euler") != std::string::npos){
01176         ButcherTableau butcher(2,2,0.0);
01177         butcher[1][1] = 1;
01178         butcher[0][1] = 1;
01179         butcher[0][0] = 1;
01180
01181         return butcher;
01182     }
01183     else if(method.find("Alexander") != std::string::npos){
01184         ButcherTableau butcher(3,3,0.0);
01185         const number_type alpha = 1. - sqrt(2.)/2.;
01186         butcher[0][0] = alpha;
01187         butcher[0][1] = alpha;
01188
01189         butcher[1][0] = 1.;
01190         butcher[1][1] = 1. - alpha;
01191         butcher[1][2] = alpha;
01192
01193         butcher[2][1] = 1. - alpha;
01194         butcher[2][2] = alpha;
01195
01196         return butcher;
01197     }
01198     else if(method.find("Crouzieux") != std::string::npos){
01199         ButcherTableau butcher(3,3,0.0);
01200         const number_type beta = 1./2./sqrt(3);
01201         butcher[0][0] = 0.5 + beta;
01202         butcher[0][1] = 0.5 + beta;
01203
01204         butcher[1][0] = 0.5 - beta;
01205         butcher[1][1] = -1. / sqrt(3);
01206         butcher[1][2] = 0.5 + beta;
01207
01208         butcher[2][1] = 0.5;
01209         butcher[2][2] = 0.5;
01210
01211         return butcher;
01212     }
01213     else if(method.find("Midpoint Rule") != std::string::npos){
01214         ButcherTableau butcher(2,2,0.0);
01215         butcher[0][0] = 0.5;
01216         butcher[0][1] = 0.5;
01217         butcher[1][1] = 1;
01218
01219         return butcher;
01220     }
01221     else if(method.find("Fractional Step Theta") != std::string::npos){
01222         ButcherTableau butcher(5,5,0.0);
01223         const number_type theta = 1 - sqrt(2.)/2.;
01224         const number_type alpha = 2. - sqrt(2.);
01225         const number_type beta = 1. - alpha;
01226         butcher[1][0] = theta;
01227         butcher[1][1] = beta * theta;
01228         butcher[1][2] = alpha * theta;
01229
01230         butcher[2][0] = 1.-theta;
01231         butcher[2][1] = beta * theta;
01232         butcher[2][2] = alpha * (1.-theta);
01233         butcher[2][3] = alpha * theta;
01234
01235         butcher[3][0] = 1.;
01236         butcher[3][1] = beta * theta;
01237         butcher[3][2] = alpha * (1.-theta);
01238         butcher[3][3] = (alpha + beta) * theta;
01239         butcher[3][4] = alpha * theta;
01240
01241         butcher[4][1] = beta * theta;
01242         butcher[4][2] = alpha * (1.-theta);
01243         butcher[4][3] = (alpha + beta) * theta;
01244         butcher[4][4] = alpha * theta;
01245
01246         return butcher;
01247     }
01248     else{
01249         HDNUM_ERROR("Order not available for Runge Kutta solver.");
01250     }
01251 }
01252
01253 static int initOrder(const std::string method)
01254 {
01255     if(method.find("Implicit Euler") != std::string::npos){
01256         return 1;
01257     }
01258     else if(method.find("Alexander") != std::string::npos){
01259         return 2;
01260     }
01261     else if(method.find("Crouzieux") != std::string::npos){

```

```

01262     return 3;
01263 }
01264 else if(method.find("Midpoint Rule") != std::string::npos){
01265     return 2;
01266 }
01267 else if(method.find("Fractional Step Theta") != std::string::npos){
01268     return 2;
01269 }
01270 else{
01271     HDNUM_ERROR("Order not available for Runge Kutta solver.");
01272 }
01273 }
01274
01275
01277 // h_n f(t_n, y_n) - y_n + y_{n-1} = 0
01278 class NonlinearProblem
01279 {
01280 public:
01282     typedef typename M::size_type size_type;
01283
01285     typedef typename M::number_type number_type;
01286
01288     NonlinearProblem (const M& model_, const Vector<number_type>& yold_,
01289                     typename M::time_type told_, typename M::time_type dt_,
01290                     const ButcherTableau & butcher_, const int rk_step_,
01291                     const std::vector< Vector<number_type> > & k_)
01292     : model(model_), yold(yold_), told(told_),
01293       dt(dt_), butcher(butcher_), rk_step(rk_step_), k_old(model.size(),0)
01294     {
01295         for(int i=0; i<rk_step; ++i)
01296             k_old.update(butcher[rk_step][1+i] * dt, k_[i]);
01297     }
01298
01300     std::size_t size () const
01301     {
01302         return model.size();
01303     }
01304
01306     void F (const Vector<number_type>& x, Vector<number_type>& result) const
01307     {
01308         result = k_old;
01309
01310         Vector<number_type> current_z(x);
01311         current_z.update(1.,yold);
01312
01313         const number_type tnew = told + butcher[rk_step][0] * dt;
01314
01315         Vector<number_type> current_k(model.size(),0.);
01316         model.f(tnew,current_z,current_k);
01317         result.update(butcher[rk_step][rk_step+1] * dt, current_k);
01318
01319         result.update(-1.,x);
01320     }
01321
01323     void F_x (const Vector<number_type>& x, DenseMatrix<number_type>& result) const
01324     {
01325         const number_type tnew = told + butcher[rk_step][0] * dt;
01326
01327         Vector<number_type> current_z(x);
01328         current_z.update(1.,yold);
01329
01330         model.f_x(tnew,current_z,result);
01331
01332         result *= dt * butcher[rk_step][rk_step+1];
01333
01334         for (size_type i=0; i<model.size(); i++) result[i][i] -= number_type(1.0);
01335     }
01336
01337     void set_told_dt (typename M::time_type told_, typename M::time_type dt_)
01338     {
01339         told = told_;
01340         dt = dt_;
01341     }
01342
01343 private:
01344     const M& model;
01345     const Vector<number_type>& yold;
01346     typename M::time_type told;
01347     typename M::time_type dt;
01348     const ButcherTableau & butcher;
01349     const int rk_step;
01350     Vector<number_type> k_old;
01351 };
01352
01353 public:
01354
01357     DIRK (const M& model_, const S& newton_, const ButcherTableau & butcher_, const int order_)

```

```

01358     : verbosity(0), butcher(butcher_), model(model_), newton(newton_),
01359       u(model.size()), order(order_)
01360   {
01361     model.initialize(t,u);
01362     dt = dtmax = 0.1;
01363   }
01364
01365   DIRK (const M& model_, const S& newton_, const std::string method)
01366     : verbosity(0), butcher(initTableau(method)), model(model_), newton(newton_), u(model.size()),
01367       order(initOrder(method))
01370   {
01371     model.initialize(t,u);
01372     dt = dtmax = 0.1;
01373   }
01374
01375
01376   void set_dt (time_type dt_)
01377   {
01378     dt = dtmax = dt_;
01379   }
01380
01381
01382   void set_verbosity (size_type verbosity_)
01383   {
01384     verbosity = verbosity_;
01385   }
01386
01387
01388   void step ()
01389   {
01390     {
01391       const size_type R = butcher.colsize()-1;
01392
01393       bool reduced = false;
01394       error = false;
01395       if(verbosity>=2)
01396         std::cout << "DIRK: step to" << " t+dt=" << t+dt << " dt=" << dt << std::endl;
01397
01398       while (1)
01399       {
01400         bool converged = true;
01401
01402         // Perform R Runge-Kutta steps
01403         std::vector< Vector<number_type> > k;
01404         for(size_type i=0; i<R; ++i) {
01405           if (verbosity>=2)
01406             std::cout << "DIRK: step nr " << i << std::endl;
01407
01408           Vector<number_type> current_z(model.size(),0.0);
01409
01410           // Set starting value of k_i
01411           // model.f(t,u,current_k);
01412
01413           // Solve nonlinear problem
01414           NonlinearProblem nlp(model,u,t,dt,butcher,i,k);
01415
01416           newton.solve(nlp,current_z);
01417
01418           converged = converged && newton.has_converged();
01419           if(!converged)
01420             break;
01421
01422           current_z.update(1., u);
01423           const number_type t_i = t + butcher[i][0] * dt;
01424           Vector<number_type>current_k(model.size(),0.);
01425           model.f(t_i,current_z,current_k);
01426
01427           k.push_back( current_k );
01428         }
01429
01430         if (converged)
01431         {
01432           if(verbosity >= 2)
01433             std::cout << "DIRK finished" << std::endl;
01434
01435           // Update to new solution
01436           for(size_type i=0; i<R; ++i)
01437             u.update(dt*butcher[R][1+i],k[i]);
01438
01439           t += dt;
01440           if (!reduced && dt<dtmax-1e-13)
01441           {
01442             dt = std::min(2.0*dt,dtmax);
01443             if (verbosity>0)
01444               std::cout << "DIRK: increasing time step to " << dt << std::endl;
01445           }
01446           return;
01447         }
01448       }
01449       else

```

```

01450         {
01451             if (dt<1e-12)
01452             {
01453                 HDNUM_ERROR("time step too small in implicit Euler");
01454                 error = true;
01455                 break;
01456             }
01457             dt *= 0.5;
01458             reduced = true;
01459             if (verbosity>0) std::cout << "DIRK: reducing time step to " << dt << std::endl;
01460         }
01461     }
01462 }
01463
01465 bool get_error () const
01466 {
01467     return error;
01468 }
01469
01471 void set_state (time_type t_, const Vector<number_type>& u_)
01472 {
01473     t = t_;
01474     u = u_;
01475 }
01476
01478 const Vector<number_type>& get_state () const
01479 {
01480     return u;
01481 }
01482
01484 time_type get_time () const
01485 {
01486     return t;
01487 }
01488
01490 time_type get_dt () const
01491 {
01492     return dt;
01493 }
01494
01496 size_type get_order () const
01497 {
01498     return order;
01499 }
01500
01502 void get_info () const
01503 {
01504 }
01505
01506 private:
01507     size_type verbosity;
01508     const DenseMatrix<number_type> butcher;
01509     const M& model;
01510     const S& newton;
01511     time_type t, dt, dtmax;
01512     number_type reduction;
01513     size_type linesearchsteps;
01514     Vector<number_type> u;
01515     int order;
01516     mutable bool error;
01517 };
01518
01520 template<class T, class N>
01521 inline void gnuplot (const std::string& fname, const std::vector<T> t, const std::vector<Vector<N> >
u)
01522 {
01523     std::fstream f(fname.c_str(),std::ios::out);
01524     for (typename std::vector<T>::size_type n=0; n<t.size(); n++)
01525     {
01526         f << std::scientific << std::showpoint
01527           << std::setprecision(16) << t[n];
01528         for (typename Vector<N>::size_type i=0; i<u[n].size(); i++)
01529             f << " " << std::scientific << std::showpoint
01530               << std::setprecision(u[n].precision()) << u[n][i];
01531         f << std::endl;
01532     }
01533     f.close();
01534 }
01535
01537 template<class T, class N>
01538 inline void gnuplot (const std::string& fname, const std::vector<T> t, const std::vector<Vector<N> >
u, const std::vector<T> dt)
01539 {
01540     std::fstream f(fname.c_str(),std::ios::out);
01541     for (typename std::vector<T>::size_type n=0; n<t.size(); n++)
01542     {
01543         f << std::scientific << std::showpoint

```

```

01544         « std::setprecision(16) « t[n];
01545     for (typename Vector<N>::size_type i=0; i<u[n].size(); i++)
01546         f « " " « std::scientific « std::showpoint
01547         « std::setprecision(u[n].precision()) « u[n][i];
01548     f « " " « std::scientific « std::showpoint
01549     « std::setprecision(16) « dt[n];
01550     f « std::endl;
01551     }
01552     f.close();
01553 }
01554
01555 } // namespace hdnum
01556
01557 #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)

- `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.](#)

```

00001 // -*- tab-width: 4; indent-tabs-mode: nil -*-
00002 #ifndef __OPCOUNTER__
00003 #define __OPCOUNTER__
00004
00005 #include <type_traits>
00006 #include <iostream>
00007 #include <cmath>
00008 #include <cstdlib>
00009
00014 namespace hdnum {
00015     namespace oc {
00016         template<typename F>
00017         class OpCounter;
00018     }
00019 }
00020
00021 namespace hdnum {
00022     namespace oc {
00028         template<typename F>
00029         class OpCounter
00030         {
00031
00032         public:
00033
00034             using size_type = std::size_t;
00035
00036             using value_type = F;
00037
00038             OpCounter()
00039             : _v()
00040             {}
00041
00042             template<typename T>
00043             OpCounter(const T& t, typename std::enable_if<std::is_same<T,int>::value and
!std::is_same<F,int>::value>::type* = nullptr)

```

```

00044         : _v(t)
00045     {}
00046
00047     OpCounter(const F& f)
00048         : _v(f)
00049     {}
00050
00051     OpCounter(F&& f)
00052         : _v(f)
00053     {}
00054
00055     explicit OpCounter(const char* s)
00056         : _v(strtod(s,nullptr))
00057     {}
00058
00059     OpCounter& operator=(const char* s)
00060     {
00061         _v = strtod(s,nullptr);
00062         return *this;
00063     }
00064
00065     explicit operator F() const
00066     {
00067         return _v;
00068     }
00069
00070     OpCounter& operator=(const F& f)
00071     {
00072         _v = f;
00073         return *this;
00074     }
00075
00076     OpCounter& operator=(F&& f)
00077     {
00078         _v = f;
00079         return *this;
00080     }
00081
00082     friend std::ostream& operator<(std::ostream& os, const OpCounter& f)
00083     {
00084         os << "OC(" << f._v << ")";
00085         return os;
00086     }
00087
00088     friend std::stringstream& operator>(std::stringstream& iss, OpCounter& f)
00089     {
00090         iss >> f._v;
00091         return iss;
00092     }
00093
00094     F* data()
00095     {
00096         return &_v;
00097     }
00098
00099     const F* data() const
00100     {
00101         return &_v;
00102     }
00103
00104     F _v;
00105
00106     struct Counters {
00107
00108         size_type addition_count;
00109         size_type multiplication_count;
00110         size_type division_count;
00111         size_type exp_count;
00112         size_type pow_count;
00113         size_type sin_count;
00114         size_type sqrt_count;
00115         size_type comparison_count;
00116
00117         Counters()
00118             : addition_count(0)
00119               , multiplication_count(0)
00120               , division_count(0)
00121               , exp_count(0)
00122               , pow_count(0)
00123               , sin_count(0)
00124               , sqrt_count(0)
00125               , comparison_count(0)
00126         {}
00127
00128         void reset()
00129         {
00130             addition_count = 0;

```

```

00132     multiplication_count = 0;
00133     division_count = 0;
00134     exp_count = 0;
00135     pow_count = 0;
00136     sin_count = 0;
00137     sqrt_count = 0;
00138     comparison_count = 0;
00139 }
00140
00142 template<typename Stream>
00143 void reportOperations(Stream& os, bool doReset = false)
00144 {
00145     os << "additions: " << addition_count << std::endl
00146         << "multiplications: " << multiplication_count << std::endl
00147         << "divisions: " << division_count << std::endl
00148         << "exp: " << exp_count << std::endl
00149         << "pow: " << pow_count << std::endl
00150         << "sin: " << sin_count << std::endl
00151         << "sqrt: " << sqrt_count << std::endl
00152         << "comparisons: " << comparison_count << std::endl
00153         << std::endl
00154         << "total: " << addition_count + multiplication_count + division_count + exp_count +
pow_count + sin_count + sqrt_count + comparison_count << std::endl;
00155
00156     if (doReset)
00157         reset();
00158 }
00159
00161 size_type totalOperationCount(bool doReset=false)
00162 {
00163     if (doReset)
00164         reset();
00165
00166     return addition_count + multiplication_count + division_count + exp_count + pow_count +
sin_count + sqrt_count + comparison_count;
00167 }
00168
00169 Counters& operator+=(const Counters& rhs)
00170 {
00171     addition_count += rhs.addition_count;
00172     multiplication_count += rhs.multiplication_count;
00173     division_count += rhs.division_count;
00174     exp_count += rhs.exp_count;
00175     pow_count += rhs.pow_count;
00176     sin_count += rhs.sin_count;
00177     sqrt_count += rhs.sqrt_count;
00178     comparison_count += rhs.comparison_count;
00179     return *this;
00180 }
00181
00182 Counters operator-(const Counters& rhs)
00183 {
00184     Counters r;
00185     r.addition_count = addition_count - rhs.addition_count;
00186     r.multiplication_count = multiplication_count - rhs.multiplication_count;
00187     r.division_count = division_count - rhs.division_count;
00188     r.exp_count = exp_count - rhs.exp_count;
00189     r.pow_count = pow_count - rhs.pow_count;
00190     r.sin_count = sin_count - rhs.sin_count;
00191     r.sqrt_count = sqrt_count - rhs.sqrt_count;
00192     r.comparison_count = comparison_count - rhs.comparison_count;
00193     return r;
00194 }
00195
00196 };
00197
00198 static void additions(std::size_t n)
00199 {
00200     counters.addition_count += n;
00201 }
00202
00203 static void multiplications(std::size_t n)
00204 {
00205     counters.multiplication_count += n;
00206 }
00207
00208 static void divisions(std::size_t n)
00209 {
00210     counters.division_count += n;
00211 }
00212
00213 static void reset()
00214 {
00215     counters.reset();
00216 }
00217
00219 template<typename Stream>

```

```

00220     static void reportOperations(Stream& os, bool doReset = false)
00221     {
00222         counters.reportOperations(os, doReset);
00223     }
00224
00226     static size_type totalOperationCount(bool doReset=false)
00227     {
00228         return counters.totalOperationCount(doReset);
00229     }
00230
00231     static Counters counters;
00232
00233 };
00234
00235 template<typename F>
00236 typename OpCounter<F>::Counters OpCounter<F>::counters;
00237
00238 // *****
00239 // negation
00240 // *****
00241
00242 template<typename F>
00243 OpCounter<F> operator-(const OpCounter<F>& a)
00244 {
00245     ++OpCounter<F>::counters.addition_count;
00246     return {-a._v};
00247 }
00248
00249 // *****
00250 // addition
00251 // *****
00252
00253 template<typename F>
00254 OpCounter<F> operator+(const OpCounter<F>& a, const OpCounter<F>& b)
00255 {
00256     ++OpCounter<F>::counters.addition_count;
00257     return {a._v + b._v};
00258 }
00259
00260 template<typename F>
00261 OpCounter<F> operator+(const OpCounter<F>& a, const F& b)
00262 {
00263     ++OpCounter<F>::counters.addition_count;
00264     return {a._v + b};
00265 }
00266
00267 template<typename F>
00268 OpCounter<F> operator+(const F& a, const OpCounter<F>& b)
00269 {
00270     ++OpCounter<F>::counters.addition_count;
00271     return {a + b._v};
00272 }
00273
00274 template<typename F, typename T>
00275 typename std::enable_if<
00276     std::is_arithmetic<T>::value,
00277     OpCounter<F>
00278 >::type
00279 operator+(const OpCounter<F>& a, const T& b)
00280 {
00281     ++OpCounter<F>::counters.addition_count;
00282     return {a._v + b};
00283 }
00284
00285 template<typename F, typename T>
00286 typename std::enable_if<
00287     std::is_arithmetic<T>::value,
00288     OpCounter<F>
00289 >::type
00290 operator+(const T& a, const OpCounter<F>& b)
00291 {
00292     ++OpCounter<F>::counters.addition_count;
00293     return {a + b._v};
00294 }
00295
00296 template<typename F>
00297 OpCounter<F>& operator+=(OpCounter<F>& a, const OpCounter<F>& b)
00298 {
00299     ++OpCounter<F>::counters.addition_count;
00300     a._v += b._v;
00301     return a;
00302 }
00303
00304 template<typename F>
00305 OpCounter<F>& operator+=(OpCounter<F>& a, const F& b)
00306 {

```

```

00308     ++OpCounter<F>::counters.addition_count;
00309     a._v += b;
00310     return a;
00311 }
00312
00313 template<typename F, typename T>
00314 typename std::enable_if<
00315     std::is_arithmetic<T>::value,
00316     OpCounter<F>&
00317 >::type
00318 operator+=(OpCounter<F>& a, const T& b)
00319 {
00320     ++OpCounter<F>::counters.addition_count;
00321     a._v += b;
00322     return a;
00323 }
00324
00325 // *****
00326 // subtraction
00327 // *****
00328
00329 template<typename F>
00330 OpCounter<F> operator-(const OpCounter<F>& a, const OpCounter<F>& b)
00331 {
00332     ++OpCounter<F>::counters.addition_count;
00333     return {a._v - b._v};
00334 }
00335
00336 template<typename F>
00337 OpCounter<F> operator-(const OpCounter<F>& a, const F& b)
00338 {
00339     ++OpCounter<F>::counters.addition_count;
00340     return {a._v - b};
00341 }
00342
00343 template<typename F>
00344 OpCounter<F> operator-(const F& a, const OpCounter<F>& b)
00345 {
00346     ++OpCounter<F>::counters.addition_count;
00347     return {a - b._v};
00348 }
00349
00350 template<typename F, typename T>
00351 typename std::enable_if<
00352     std::is_arithmetic<T>::value,
00353     OpCounter<F>
00354 >::type
00355 operator-(const OpCounter<F>& a, const T& b)
00356 {
00357     ++OpCounter<F>::counters.addition_count;
00358     return {a._v - b};
00359 }
00360
00361 template<typename F, typename T>
00362 typename std::enable_if<
00363     std::is_arithmetic<T>::value,
00364     OpCounter<F>
00365 >::type
00366 operator-(const T& a, const OpCounter<F>& b)
00367 {
00368     ++OpCounter<F>::counters.addition_count;
00369     return {a - b._v};
00370 }
00371
00372 template<typename F>
00373 OpCounter<F>& operator--(OpCounter<F>& a, const OpCounter<F>& b)
00374 {
00375     ++OpCounter<F>::counters.addition_count;
00376     a._v -= b._v;
00377     return a;
00378 }
00379
00380 template<typename F>
00381 OpCounter<F>& operator--(OpCounter<F>& a, const F& b)
00382 {
00383     ++OpCounter<F>::counters.addition_count;
00384     a._v -= b;
00385     return a;
00386 }
00387
00388 template<typename F, typename T>
00389 typename std::enable_if<
00390     std::is_arithmetic<T>::value,
00391     OpCounter<F>&
00392 >::type
00393 operator==(OpCounter<F>& a, const T& b)
00394 {

```

```

00395     ++OpCounter<F>::counters.addition_count;
00396     a._v -= b;
00397     return a;
00398 }
00399
00400
00401 // *****
00402 // multiplication
00403 // *****
00404
00405 template<typename F>
00406 OpCounter<F> operator*(const OpCounter<F>& a, const OpCounter<F>& b)
00407 {
00408     ++OpCounter<F>::counters.multiplication_count;
00409     return {a._v * b._v};
00410 }
00411
00412 template<typename F>
00413 OpCounter<F> operator*(const OpCounter<F>& a, const F& b)
00414 {
00415     ++OpCounter<F>::counters.multiplication_count;
00416     return {a._v * b};
00417 }
00418
00419 template<typename F>
00420 OpCounter<F> operator*(const F& a, const OpCounter<F>& b)
00421 {
00422     ++OpCounter<F>::counters.multiplication_count;
00423     return {a * b._v};
00424 }
00425
00426 template<typename F, typename T>
00427 typename std::enable_if<
00428     std::is_arithmetic<T>::value,
00429     OpCounter<F>
00430 >::type
00431 operator*(const OpCounter<F>& a, const T& b)
00432 {
00433     ++OpCounter<F>::counters.multiplication_count;
00434     return {a._v * b};
00435 }
00436
00437 template<typename F, typename T>
00438 typename std::enable_if<
00439     std::is_arithmetic<T>::value,
00440     OpCounter<F>
00441 >::type
00442 operator*(const T& a, const OpCounter<F>& b)
00443 {
00444     ++OpCounter<F>::counters.multiplication_count;
00445     return {a * b._v};
00446 }
00447
00448 template<typename F>
00449 OpCounter<F>& operator*=(OpCounter<F>& a, const OpCounter<F>& b)
00450 {
00451     ++OpCounter<F>::counters.multiplication_count;
00452     a._v *= b._v;
00453     return a;
00454 }
00455
00456 template<typename F>
00457 OpCounter<F>& operator*=(OpCounter<F>& a, const F& b)
00458 {
00459     ++OpCounter<F>::counters.multiplication_count;
00460     a._v *= b;
00461     return a;
00462 }
00463
00464 template<typename F, typename T>
00465 typename std::enable_if<
00466     std::is_arithmetic<T>::value,
00467     OpCounter<F>&
00468 >::type
00469 operator*=(OpCounter<F>& a, const T& b)
00470 {
00471     ++OpCounter<F>::counters.multiplication_count;
00472     a._v *= b;
00473     return a;
00474 }
00475
00476
00477 // *****
00478 // division
00479 // *****
00480
00481 template<typename F>

```



```

00482     OpCounter<F> operator/(const OpCounter<F>& a, const OpCounter<F>& b)
00483     {
00484         ++OpCounter<F>::counters.division_count;
00485         return {a._v / b._v};
00486     }
00487
00488     template<typename F>
00489     OpCounter<F> operator/(const OpCounter<F>& a, const F& b)
00490     {
00491         ++OpCounter<F>::counters.division_count;
00492         return {a._v / b};
00493     }
00494
00495     template<typename F>
00496     OpCounter<F> operator/(const F& a, const OpCounter<F>& b)
00497     {
00498         ++OpCounter<F>::counters.division_count;
00499         return {a / b._v};
00500     }
00501
00502     template<typename F, typename T>
00503     typename std::enable_if<
00504         std::is_arithmetic<T>::value,
00505         OpCounter<F>
00506     >::type
00507     operator/(const OpCounter<F>& a, const T& b)
00508     {
00509         ++OpCounter<F>::counters.division_count;
00510         return {a._v / b};
00511     }
00512
00513     template<typename F, typename T>
00514     typename std::enable_if<
00515         std::is_arithmetic<T>::value,
00516         OpCounter<F>
00517     >::type
00518     operator/(const T& a, const OpCounter<F>& b)
00519     {
00520         ++OpCounter<F>::counters.division_count;
00521         return {a / b._v};
00522     }
00523
00524     template<typename F>
00525     OpCounter<F>& operator/=(OpCounter<F>& a, const OpCounter<F>& b)
00526     {
00527         ++OpCounter<F>::counters.division_count;
00528         a._v /= b._v;
00529         return a;
00530     }
00531
00532     template<typename F>
00533     OpCounter<F>& operator/=(OpCounter<F>& a, const F& b)
00534     {
00535         ++OpCounter<F>::counters.division_count;
00536         a._v /= b;
00537         return a;
00538     }
00539
00540     template<typename F, typename T>
00541     typename std::enable_if<
00542         std::is_arithmetic<T>::value,
00543         OpCounter<F>&
00544     >::type
00545     operator/=(OpCounter<F>& a, const T& b)
00546     {
00547         ++OpCounter<F>::counters.division_count;
00548         a._v /= b;
00549         return a;
00550     }
00551
00552
00553     // *****
00554     // comparisons
00555     // *****
00556
00557
00558     // *****
00559     // less
00560     // *****
00561
00562
00563     template<typename F>
00564     bool operator<(const OpCounter<F>& a, const OpCounter<F>& b)
00565     {
00566         ++OpCounter<F>::counters.comparison_count;
00567         return {a._v < b._v};
00568     }

```

```

00569
00570     template<typename F>
00571     bool operator<(const OpCounter<F>& a, const F& b)
00572     {
00573         ++OpCounter<F>::counters.comparison_count;
00574         return {a._v < b};
00575     }
00576
00577     template<typename F>
00578     bool operator<(const F& a, const OpCounter<F>& b)
00579     {
00580         ++OpCounter<F>::counters.comparison_count;
00581         return {a < b._v};
00582     }
00583
00584     template<typename F, typename T>
00585     bool operator<(const OpCounter<F>& a, const T& b)
00586     {
00587         ++OpCounter<F>::counters.comparison_count;
00588         return {a._v < b};
00589     }
00590
00591     template<typename F, typename T>
00592     bool operator<(const T& a, const OpCounter<F>& b)
00593     {
00594         ++OpCounter<F>::counters.comparison_count;
00595         return {a < b._v};
00596     }
00597
00598
00599     // *****
00600     // less_or_equals
00601     // *****
00602
00603     template<typename F>
00604     bool operator<=(const OpCounter<F>& a, const OpCounter<F>& b)
00605     {
00606         ++OpCounter<F>::counters.comparison_count;
00607         return {a._v <= b._v};
00608     }
00609
00610     template<typename F>
00611     bool operator<=(const OpCounter<F>& a, const F& b)
00612     {
00613         ++OpCounter<F>::counters.comparison_count;
00614         return {a._v <= b};
00615     }
00616
00617     template<typename F>
00618     bool operator<=(const F& a, const OpCounter<F>& b)
00619     {
00620         ++OpCounter<F>::counters.comparison_count;
00621         return {a <= b._v};
00622     }
00623
00624     template<typename F, typename T>
00625     bool operator<=(const OpCounter<F>& a, const T& b)
00626     {
00627         ++OpCounter<F>::counters.comparison_count;
00628         return {a._v <= b};
00629     }
00630
00631     template<typename F, typename T>
00632     bool operator<=(const T& a, const OpCounter<F>& b)
00633     {
00634         ++OpCounter<F>::counters.comparison_count;
00635         return {a <= b._v};
00636     }
00637
00638
00639     // *****
00640     // greater
00641     // *****
00642
00643     template<typename F>
00644     bool operator>(const OpCounter<F>& a, const OpCounter<F>& b)
00645     {
00646         ++OpCounter<F>::counters.comparison_count;
00647         return {a._v > b._v};
00648     }
00649
00650     template<typename F>
00651     bool operator>(const OpCounter<F>& a, const F& b)
00652     {
00653         ++OpCounter<F>::counters.comparison_count;
00654         return {a._v > b};
00655     }

```

```

00656
00657     template<typename F>
00658     bool operator>(const F& a, const OpCounter<F>& b)
00659     {
00660         ++OpCounter<F>::counters.comparison_count;
00661         return {a > b._v};
00662     }
00663
00664     template<typename F, typename T>
00665     bool operator>(const OpCounter<F>& a, const T& b)
00666     {
00667         ++OpCounter<F>::counters.comparison_count;
00668         return {a._v > b};
00669     }
00670
00671     template<typename F, typename T>
00672     bool operator>(const T& a, const OpCounter<F>& b)
00673     {
00674         ++OpCounter<F>::counters.comparison_count;
00675         return {a > b._v};
00676     }
00677
00678
00679     // *****
00680     // greater_or_equals
00681     // *****
00682
00683     template<typename F>
00684     bool operator>=(const OpCounter<F>& a, const OpCounter<F>& b)
00685     {
00686         ++OpCounter<F>::counters.comparison_count;
00687         return {a._v >= b._v};
00688     }
00689
00690     template<typename F>
00691     bool operator>=(const OpCounter<F>& a, const F& b)
00692     {
00693         ++OpCounter<F>::counters.comparison_count;
00694         return {a._v >= b};
00695     }
00696
00697     template<typename F>
00698     bool operator>=(const F& a, const OpCounter<F>& b)
00699     {
00700         ++OpCounter<F>::counters.comparison_count;
00701         return {a >= b._v};
00702     }
00703
00704     template<typename F, typename T>
00705     bool operator>=(const OpCounter<F>& a, const T& b)
00706     {
00707         ++OpCounter<F>::counters.comparison_count;
00708         return {a._v >= b};
00709     }
00710
00711     template<typename F, typename T>
00712     bool operator>=(const T& a, const OpCounter<F>& b)
00713     {
00714         ++OpCounter<F>::counters.comparison_count;
00715         return {a >= b._v};
00716     }
00717
00718
00719     // *****
00720     // inequals
00721     // *****
00722
00723     template<typename F>
00724     bool operator!=(const OpCounter<F>& a, const OpCounter<F>& b)
00725     {
00726         ++OpCounter<F>::counters.comparison_count;
00727         return {a._v != b._v};
00728     }
00729
00730     template<typename F>
00731     bool operator!=(const OpCounter<F>& a, const F& b)
00732     {
00733         ++OpCounter<F>::counters.comparison_count;
00734         return {a._v != b};
00735     }
00736
00737     template<typename F>
00738     bool operator!=(const F& a, const OpCounter<F>& b)
00739     {
00740         ++OpCounter<F>::counters.comparison_count;
00741         return {a != b._v};
00742     }

```

```

00743
00744     template<typename F, typename T>
00745     bool operator!=(const OpCounter<F>& a, const T& b)
00746     {
00747         ++OpCounter<F>::counters.comparison_count;
00748         return {a._v != b};
00749     }
00750
00751     template<typename F, typename T>
00752     bool operator!=(const T& a, const OpCounter<F>& b)
00753     {
00754         ++OpCounter<F>::counters.comparison_count;
00755         return {a != b._v};
00756     }
00757
00758     // *****
00759     // equals
00760     // *****
00761
00762     template<typename F>
00763     bool operator==(const OpCounter<F>& a, const OpCounter<F>& b)
00764     {
00765         ++OpCounter<F>::counters.comparison_count;
00766         return {a._v == b._v};
00767     }
00768
00769     template<typename F>
00770     bool operator==(const OpCounter<F>& a, const F& b)
00771     {
00772         ++OpCounter<F>::counters.comparison_count;
00773         return {a._v == b};
00774     }
00775
00776     template<typename F>
00777     bool operator==(const F& a, const OpCounter<F>& b)
00778     {
00779         ++OpCounter<F>::counters.comparison_count;
00780         return {a == b._v};
00781     }
00782
00783     template<typename F, typename T>
00784     bool operator==(const OpCounter<F>& a, const T& b)
00785     {
00786         ++OpCounter<F>::counters.comparison_count;
00787         return {a._v == b};
00788     }
00789
00790     template<typename F, typename T>
00791     bool operator==(const T& a, const OpCounter<F>& b)
00792     {
00793         ++OpCounter<F>::counters.comparison_count;
00794         return {a == b._v};
00795     }
00796
00797
00798
00799
00800     // *****
00801     // functions
00802     // *****
00803
00804     template<typename F>
00805     OpCounter<F> exp(const OpCounter<F>& a)
00806     {
00807         ++OpCounter<F>::counters.exp_count;
00808         return {std::exp(a._v)};
00809     }
00810
00811     template<typename F>
00812     OpCounter<F> pow(const OpCounter<F>& a, const OpCounter<F>& b)
00813     {
00814         ++OpCounter<F>::counters.pow_count;
00815         return {std::pow(a._v,b._v)};
00816     }
00817
00818     template<typename F>
00819     OpCounter<F> pow(const OpCounter<F>& a, const F& b)
00820     {
00821         ++OpCounter<F>::counters.pow_count;
00822         return {std::pow(a._v,b)};
00823     }
00824
00825     template<typename F, typename T>
00826     OpCounter<F> pow(const OpCounter<F>& a, const T& b)
00827     {
00828         ++OpCounter<F>::counters.pow_count;
00829         return {std::pow(a._v,b)};

```

```

00830     }
00831
00832     template<typename F>
00833     OpCounter<F> pow(const F& a, const OpCounter<F>& b)
00834     {
00835         ++OpCounter<F>::counters.pow_count;
00836         return {std::pow(a,b._v)};
00837     }
00838
00839     template<typename F, typename T>
00840     OpCounter<F> pow(const T& a, const OpCounter<F>& b)
00841     {
00842         ++OpCounter<F>::counters.pow_count;
00843         return {std::pow(a,b._v)};
00844     }
00845
00846     template<typename F>
00847     OpCounter<F> sin(const OpCounter<F>& a)
00848     {
00849         ++OpCounter<F>::counters.sin_count;
00850         return {std::sin(a._v)};
00851     }
00852
00853     template<typename F>
00854     OpCounter<F> cos(const OpCounter<F>& a)
00855     {
00856         ++OpCounter<F>::counters.sin_count;
00857         return {std::cos(a._v)};
00858     }
00859
00860     template<typename F>
00861     OpCounter<F> sqrt(const OpCounter<F>& a)
00862     {
00863         ++OpCounter<F>::counters.sqrt_count;
00864         return {std::sqrt(a._v)};
00865     }
00866
00867     template<typename F>
00868     OpCounter<F> abs(const OpCounter<F>& a)
00869     {
00870         ++OpCounter<F>::counters.comparison_count;
00871         return {std::abs(a._v)};
00872     }
00873 }
00874 }
00875
00876 #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.](#)

```

00001 // -*- tab-width: 4; indent-tabs-mode: nil -*-
00002 #ifndef HDNUM_PDE_HH
00003 #define HDNUM_PDE_HH
00004
00005 #include<vector>
00006 #include "newton.hh"
00007
00012 namespace hdnum {
00013
00022     template<class M>
00023     class StationarySolver
00024     {
00025     public:
00027         typedef typename M::size_type size_type;
00028
00030         typedef typename M::time_type time_type;
00031
00033         typedef typename M::number_type number_type;
00034
00036         StationarySolver (const M& model_)
00037             : model(model_), x(model.size())
00038         {
00039         }
00040
00042         void solve ()
00043         {
00044             const size_t n_dofs = model.size();
00045
00046             DenseMatrix<number_type> A(n_dofs,n_dofs,0.);
00047             Vector<number_type> b(n_dofs,0.);
00048
00049             Vector<number_type> s(n_dofs); // scaling factors
00050             Vector<size_t> p(n_dofs); // row permutations
00051             Vector<size_t> q(n_dofs); // column permutations
00052
00053             number_type t = 0.;
00054
00055             x = 0.;
00056
00057             model.f_x(t, x, A);
00058             model.f(t, x, b);
00059
00060             b*=-1.;
00061
00062             row_equilibrate(A,s); // equilibrate rows
00063             lr_fullpivot(A,p,q); // LR decomposition of A
00064             apply_equilibrate(s,b); // equilibration of right hand side
00065             permute_forward(p,b); // permutation of right hand side
00066             solveL(A,b,b); // forward substitution
00067             solveR(A,x,b); // backward substitution
00068             permute_backward(q,x); // backward permutation
00069         }
00070
00072         const Vector<number_type>& get_state () const
00073         {
00074             return x;
00075         }
00076
00078         size_type get_order () const
00079         {
00080             return 2;
00081         }
00082
00083     private:
00084         const M& model;
00085         Vector<number_type> x;
00086     };
00087
00088
00090     template<class N, class G>
00091     inline void pde_gnuplot2d (const std::string& fname, const Vector<N> solution,
00092                               const G & grid)

```

```

00093 {
00094
00095     const std::vector<Vector<N> > coords = grid.getNodeCoordinates();
00096     Vector<typename G::size_type> gsize = grid.getGridSize();
00097
00098     std::fstream f(fname.c_str(),std::ios::out);
00099     // f << "set dgrid3d ";
00100
00101     // f << gsize[0] << ", " << gsize[1] << std::endl;
00102
00103     // f << "set hidden3d" << std::endl;
00104     f << "set ticslevel 0" << std::endl;
00105     f << "splot \"-\\" using 1:2:3 with points" << std::endl;
00106     f << "#" << std::endl;
00107     for (typename Vector<N>::size_type n=0; n<solution.size(); n++)
00108     {
00109         for (typename Vector<N>::size_type d=0; d<coords[n].size(); d++){
00110             f << std::scientific << std::showpoint
00111                 << std::setprecision(16) << coords[n][d] << " ";
00112         }
00113
00114         f << std::scientific << std::showpoint
00115             << std::setprecision(solution.precision()) << solution[n];
00116
00117         f << std::endl;
00118     }
00119     f << "end" << std::endl;
00120     f << "pause -1" << std::endl;
00121     f.close();
00122 }
00123
00124
00125 }
00126 #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.](#)

```

00001 // -*- tab-width: 4; indent-tabs-mode: nil -*-
00002 #ifndef HDNUM_PRECISION_HH
00003 #define HDNUM_PRECISION_HH
00004
00009 namespace hdnum {
00010
00011     // find largest eps such that 0.5 + eps > 0.5
00012     template<typename X>
00013     int precision (X& eps)
00014     {
00015         X x,large,largex,two;
00016         large = 0.5;
00017         two = 2.0;
00018         x = 0.5;
00019         largex = large+x;

```

```

00020     int i(0);
00021     while (largex>large)
00022     {
00023         eps = x;
00024         i = i+1;
00025         //          std::cout << i << " " << std::scientific << std::showpoint
00026         //          << std::setprecision(15) << largex << " " << x << std::endl;
00027         x = x/two;
00028         largex = largex+x;
00029     }
00030     return i;
00031 }
00032
00033 } // namespace hdnum
00034
00035 #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 > hdnun::gram_schmidt (
    const DenseMatrix< T > & A )
```

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

Template Parameters

<i>hdnum::DenseMatrix<T></i>	A
------------------------------------	---

Example:

```
hdnum::DenseMatrix<double> A({{2, 9},
                             {1, -5}});
hdnum::DenseMatrix<double> Q(hdnun::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 > hdnun::modified_gram_schmidt (
    const DenseMatrix< T > & A )
```

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

Template Parameters

<i>hdnum::DenseMatrix<T></i>	A
------------------------------------	---

Example:

```
hdnum::DenseMatrix<double> A({{2, 9},
                             {1, -5}});
hdnum::DenseMatrix<double> Q(hdnun::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<T></code>	<code>A</code>
--	----------------

Parameters

<code>hdnum::Vector<int></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<T></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<T></code>	Q
<code>T</code>	threshold (optional)

Parameters

<code>hdnum::Vector<int></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<T></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.](#)

```
00001 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
00002 /*
00003  * File: qr.hh
00004  * Author: Raphael Vogt <cx238@stud.uni-heidelberg.de>
00005  *
00006  * Created on August 30, 2020
00007  */
00008
00009 #ifndef HDNUM_QR_HH
00010 #define HDNUM_QR_HH
00011
00012 #include <cmath>
00013 #include <utility>
00014
00015 #include "densematrix.hh"
00016 #include "vector.hh"
00017
00022 namespace hdnum {
00023
00052 template <class T>
00053 DenseMatrix<T> gram_schmidt(const DenseMatrix<T>& A) {
00054     DenseMatrix<T> Q(A);
00055
00056     // for all columns except the first
00057     for (int k = 1; k < Q.colsize(); k++) {
00058         // orthogonalize column k against all previous
00059         for (int j = 0; j < k; j++) {
00060             // compute factor
00061             T sum_nom(0.0);
00062             T sum_denom(0.0);
00063             for (int i = 0; i < Q.rowsize(); i++) {
00064                 sum_nom += A[i][k] * Q[i][j];
00065                 sum_denom += Q[i][j] * Q[i][j];
00066             }
00067             // modify
00068             T alpha = sum_nom / sum_denom;
00069             for (int i = 0; i < Q.rowsize(); i++) Q[i][k] -= alpha * Q[i][j];
00070         }
00071     }
00072     for (int j = 0; j < Q.colsize(); j++) {
00073         // compute norm of column j
00074         T sum(0.0);
00075         for (int i = 0; i < Q.rowsize(); i++) sum += Q[i][j] * Q[i][j];
```

```

00076         sum = sqrt(sum);
00077         // scale
00078         for (int i = 0; i < Q.rowsize(); i++) Q[i][j] = Q[i][j] / sum;
00079     }
00080     return Q;
00081 }
00082
00111 template <class T>
00112 DenseMatrix<T> modified_gram_schmidt(const DenseMatrix<T>& A) {
00113     DenseMatrix<T> Q(A);
00114
00115     for (int k = 0; k < Q.colsize(); k++) {
00116         // modify all later columns with column k
00117         for (int j = k + 1; j < Q.colsize(); j++) {
00118             // compute factor
00119             T sum_nom(0.0);
00120             T sum_denom(0.0);
00121             for (int i = 0; i < Q.rowsize(); i++) {
00122                 sum_nom += Q[i][j] * Q[i][k];
00123                 sum_denom += Q[i][k] * Q[i][k];
00124             }
00125             // modify
00126             T alpha = sum_nom / sum_denom;
00127             for (int i = 0; i < Q.rowsize(); i++) Q[i][j] -= alpha * Q[i][k];
00128         }
00129     }
00130     for (int j = 0; j < Q.colsize(); j++) {
00131         // compute norm of column j
00132         T sum(0.0);
00133         for (int i = 0; i < Q.rowsize(); i++) sum += Q[i][j] * Q[i][j];
00134         sum = sqrt(sum);
00135         // scale
00136         for (int i = 0; i < Q.rowsize(); i++) Q[i][j] = Q[i][j] / sum;
00137     }
00138     return Q;
00139 }
00140
00182 template <class T>
00183 DenseMatrix<T> qr_gram_schmidt_simple(DenseMatrix<T>& Q) {
00184     // save matrix A, before it's replaced with Q
00185     DenseMatrix<T> A(Q);
00186
00187     // create matrix R
00188     DenseMatrix<T> R(Q.colsize(), Q.colsize());
00189
00190     // start orthogonalizing
00191     for (int k = 0; k < Q.colsize(); k++) {
00192         // modify all later columns with column k
00193         for (int j = k + 1; j < Q.colsize(); j++) {
00194             // compute factor
00195             T sum_nom(0.0);
00196             T sum_denom(0.0);
00197             for (int i = 0; i < Q.rowsize(); i++) {
00198                 sum_nom += Q(i, j) * Q(i, k);
00199                 sum_denom += Q(i, k) * Q(i, k);
00200             }
00201
00202             T alpha = sum_nom / sum_denom;
00203             for (int i = 0; i < Q.rowsize(); i++) Q(i, j) -= alpha * Q(i, k);
00204         }
00205     }
00206
00207     // add values to R, except main diagonal
00208     for (int i = 1; i < R.colsize(); i++) {
00209         for (int j = 0; j < i; j++) {
00210             T sum_nom(0.0);
00211             T sum_l2nom(0.0);
00212             for (int k = 0; k < Q.rowsize(); k++) {
00213                 sum_nom += A(k, i) * Q(k, j);
00214                 sum_l2nom += Q(k, j) * Q(k, j);
00215             }
00216             sum_l2nom = sqrt(sum_l2nom);
00217             // add element
00218             R(j, i) = sum_nom / sum_l2nom;
00219         }
00220     }
00221
00222     // add missing values and scale
00223     for (int j = 0; j < Q.colsize(); j++) {
00224         // compute norm of column j
00225         T sum(0.0);
00226         for (int i = 0; i < Q.rowsize(); i++) sum += Q(i, j) * Q(i, j);
00227         sum = sqrt(sum);
00228         // add main diagonal to R
00229         R(j, j) = sum;
00230         // scale Q
00231         for (int i = 0; i < Q.rowsize(); i++) Q(i, j) = Q(i, j) / sum;

```

```

00232     }
00233     return R;
00234 }
00235
00277 template <class T>
00278 DenseMatrix<T> qr_gram_schmidt(DenseMatrix<T>& Q) {
00279     // create matrix R
00280     DenseMatrix<T> R(Q.colsize(), Q.colsize());
00281
00282     // start orthogonalizing
00283     for (int k = 0; k < Q.colsize(); k++) {
00284         // compute norm of column k
00285         T sum_denom(0.0);
00286         for (int i = 0; i < Q.rowsize(); i++) {
00287             sum_denom += Q(i, k) * Q(i, k);
00288         }
00289
00290         // fill the main diagonal of R with elements
00291         sum_denom = sqrt(sum_denom);
00292         R(k, k) = sum_denom;
00293
00294         // scale column k to the main diagonal
00295         for (int i = 0; i < Q.rowsize(); i++) {
00296             Q(i, k) /= R(k, k);
00297         }
00298
00299         // modify all later columns with column k
00300         for (int j = k + 1; j < Q.colsize(); j++) {
00301             // compute norm of column j
00302             T sum_nom(0.0);
00303             for (int i = 0; i < Q.rowsize(); i++) {
00304                 sum_nom += Q(i, k) * Q(i, j);
00305             }
00306             // insert missing elements to R
00307             R(k, j) = sum_nom;
00308
00309             // orthogonalize column j
00310             for (int i = 0; i < Q.rowsize(); i++) {
00311                 Q(i, j) -= Q(i, k) * R(k, j);
00312             }
00313         }
00314     }
00315     return R;
00316 }
00317
00381 template <class T>
00382 DenseMatrix<T> qr_gram_schmidt_pivoting(DenseMatrix<T>& Q, Vector<int>& p, int& rank, T
threshold=0.00000000001) {
00383     // check if permutation vector has the right size
00384     if (p.size() != Q.colsize()) {
00385         HDNUM_ERROR("Permutation Vector incompatible with Matrix!");
00386     }
00387
00388     // initialize permutation vector
00389     for (int i = 0; i < p.size(); i++) {
00390         p[i] = i;
00391     }
00392
00393     // initialize rank
00394     rank = 0;
00395
00396     // save matrix A, before it's replaced with Q
00397     DenseMatrix<T> A(Q);
00398
00399     // create Matrix R
00400     hdnum::DenseMatrix<T> R(A.colsize(), A.colsize());
00401
00402     // start orthogonalizing
00403     for (int k = 0; k < Q.colsize(); k++) {
00404         // find column with highest norm
00405         // compute norm of column k
00406         T norm_k(0.0);
00407         for (int r = 0; r < Q.rowsize(); r++) {
00408             norm_k += Q(r, k) * Q(r, k);
00409         }
00410         norm_k = sqrt(norm_k);
00411
00412         // compare norm of column k to the following column norms
00413         for (int c = k+1; c < Q.colsize(); c++) {
00414             T norm(0.0);
00415             for (int r = 0; r < Q.rowsize(); r++) {
00416                 norm += Q(r, c) * Q(r, c);
00417             }
00418             norm = sqrt(norm);
00419             // store permutation
00420             if (norm > norm_k) {
00421                 p[k] = c;

```

```

00422     }
00423     }
00424
00425     // swap columns if necessary
00426     if (p[k] > k) {
00427         for (int r = 0; r < Q.rowsize(); r++) {
00428             T temp_Q = Q(r, k);
00429             Q(r, k) = Q(r, p[k]);
00430             Q(r, p[k]) = temp_Q;
00431         }
00432         p[p[k]] = k;
00433
00434         // compute norm of the new column k
00435         norm_k = 0;
00436         for (int i = 0; i < Q.rowsize(); i++) {
00437             norm_k += Q(i, k) * Q(i, k);
00438         }
00439         norm_k = sqrt(norm_k);
00440     }
00441
00442     // if norm of column k > threshold -> column k is linear independent
00443     if (norm_k > threshold) {
00444         rank++;
00445     } else {
00446         break;
00447     }
00448
00449     // modify all later columns with column k
00450     for (int j = k + 1; j < Q.colsize(); j++) {
00451         // compute factor
00452         T sum_nom(0.0);
00453         T sum_denom(0.0);
00454         for (int i = 0; i < Q.rowsize(); i++) {
00455             sum_nom += Q(i, j) * Q(i, k);
00456             sum_denom += Q(i, k) * Q(i, k);
00457         }
00458
00459         T alpha = sum_nom / sum_denom;
00460         for (int i = 0; i < Q.rowsize(); i++) Q(i, j) -= alpha * Q(i, k);
00461     }
00462 }
00463
00464 // add values to R, except main diagonal
00465 for (int i = 1; i < R.colsize(); i++) {
00466     for (int j = 0; j < i; j++) {
00467         T sum_nom(0.0);
00468         T sum_l2nom(0.0);
00469         for (int k = 0; k < Q.rowsize(); k++) {
00470             sum_nom += A(k, p[i]) * Q(k, j);
00471             sum_l2nom += Q(k, j) * Q(k, j);
00472         }
00473         sum_l2nom = sqrt(sum_l2nom);
00474         // add element
00475         R(j, i) = sum_nom / sum_l2nom;
00476     }
00477 }
00478
00479 // add missing values and scale
00480 for (int j = 0; j < Q.colsize(); j++) {
00481     // compute norm of column j
00482     T sum(0.0);
00483     for (int i = 0; i < Q.rowsize(); i++) sum += Q(i, j) * Q(i, j);
00484     sum = sqrt(sum);
00485     // add main diagonal to R
00486     R(j, j) = sum;
00487     // scale Q
00488     for (int i = 0; i < Q.rowsize(); i++) Q(i, j) = Q(i, j) / sum;
00489 }
00490
00491 return R;
00492 }
00493
00524 template<typename T>
00525 void permute_forward(DenseMatrix<T>& A, Vector<int>& p) {
00526     // check if permutation vector has the right size
00527     if (p.size() != A.colsize()) {
00528         HDNUM_ERROR("Permutation Vector incompatible with Matrix!");
00529     }
00530
00531     // permute the columns
00532     for (int k = 0; k < p.size(); k++) {
00533         if (p[k] != k) {
00534             // swap column
00535             for (int j=0; j < A.rowsize(); j++) {
00536                 T temp_A = A(j, k);
00537                 A(j, k) = A(j, p[k]);
00538                 A(j, p[k]) = temp_A;

```



```

00539         }
00540         // swap inside permutation vector
00541         int temp_p = p[k];
00542         p[k] = p[temp_p];
00543         p[temp_p] = temp_p;
00544     }
00545 }
00546 }
00547
00548 } // namespace hdnun
00549
00550 #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 > hdnun::creat_I_matrix (size_t n)`
- `template<typename REAL >`
`size_t hdnun::sgn (REAL val)`
Function that return the sign of a number.
- `template<class REAL >`
`void hdnun::qrhousholder (DenseMatrix< REAL > &A, hdnun::Vector< REAL > &v)`
Function that calculate the QR decoposition in place the elements of A will be replaced with the elements of v_{\leftarrow} 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 > hdnun::qrhousholderexplicitQ (DenseMatrix< REAL > &A, hdnun::Vector< REAL > &v, bool show_Hi=false)`
Function that calculate the QR decoposition in place and return Q the elements of A will be replaced with the elements of v_{\leftarrow} 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 qrhouholder()

```
template<class REAL >
void hdnum::qrhouholder (
    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} 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 <code>hdnum::Vector</code>

5.18.2.2 qrhouholderexplicitQ()

```
template<class REAL >
DenseMatrix< REAL > hdnum::qrhouholderexplicitQ (
    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 <code>hdnum::Vector</code>

Returns

Q matrix

5.19 qrhouholder.hh

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

```
00001 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
00002 /*
00003  * File:   qrhouholder
00004  * Author: Ahmad Fadl <abohmaid@windowslive.com>
00005  *
00006  * Created on August 25, 2020
00007  */
00008
00009 #ifndef HDNUM_QRHOUSHOLDER_HH
00010 #define HDNUM_QRHOUSHOLDER_HH
00011 #include <cmath>
00012 #include <cstdlib>
00013 #include <fstream>
00014 #include <iomanip>
00015 #include <iostream>
00016 #include <sstream>
00017 #include <string>
00018
00019 #include "densematrix.hh"
00020 #include "vector.hh"
00024 namespace hdnum {
00025 template <class REAL>
00026 DenseMatrix<REAL> creat_I_matrix(size_t n) {
00027     DenseMatrix<REAL> res(n, n, 0);
00028     for (size_t i = 0; i < n; i++) {
00029         res(i, i) = 1;
00030     }
00031     return res;
00032 }
00033
00036 template <typename REAL>
00037 size_t sgn(REAL val) {
00038     return (REAL(0) < val) - (val < REAL(0));
```

```

00039 }
00049 template <class REAL>
00050 void qrhouholder(DenseMatrix<REAL>& A, hdnum::Vector<REAL>& v) {
00051     auto m = A.rowsize();
00052     auto n = A.colsize();
00053     for (size_t j = 0; j < n; j++) {
00054         REAL s = 0;
00055         for (size_t i = j; i < m; i++) {
00056             s = s + pow(A(i, j), 2);
00057         }
00058         s = sqrt(s);
00059         v[j] = (-1.0) * sgn(A(j, j)) * s;
00060         REAL fak = sqrt(s * (s + std::abs(A(j, j))));
00061         A(j, j) = A(j, j) - v[j];
00062         for (size_t k = j; k < m; k++) {
00063             A(k, j) = A(k, j) / fak;
00064         }
00065         for (size_t i = j + 1; i < n; i++) {
00066             s = 0;
00067             for (size_t k = j; k < m; k++) {
00068                 s = s + (A(k, j) * A(k, i));
00069             }
00070             for (size_t k = j; k < m; k++) {
00071                 A(k, i) = A(k, i) - (A(k, j) * s);
00072             }
00073         }
00074         // normalize the vi vectors again
00075         for (size_t i = m; i >= 0; i--) {
00076             A(i, j) = A(i, j) * fak;
00077             if (i == j) {
00078                 break;
00079             }
00080         }
00081     }
00082 }
00093 template <class REAL>
00094 DenseMatrix<REAL> qrhouholderexplicitQ(DenseMatrix<REAL>& A,
00095                                         hdnum::Vector<REAL>& v,
00096                                         bool show_Hi = false) {
00097     auto m = A.rowsize();
00098     auto n = A.colsize();
00099     auto I = creat_I_matrix<REAL>(std::max(m, n));
00100
00101     DenseMatrix<REAL> Q(m, m, 0);
00102     for (size_t j = 0; j < n; j++) {
00103         REAL s = 0;
00104         for (size_t i = j; i < m; i++) {
00105             s = s + pow(A(i, j), 2);
00106         }
00107         s = sqrt(s);
00108         v[j] = (-1.0) * sgn(A(j, j)) * s;
00109         REAL fak = sqrt(s * (s + std::abs(A(j, j))));
00110         A(j, j) = A(j, j) - v[j];
00111         for (size_t k = j; k < m; k++) {
00112             A(k, j) = A(k, j) / fak;
00113         }
00114         for (size_t i = j + 1; i < n; i++) {
00115             s = 0;
00116             for (size_t k = j; k < m; k++) {
00117                 s = s + (A(k, j) * A(k, i));
00118             }
00119             for (size_t k = j; k < m; k++) {
00120                 A(k, i) = A(k, i) - (A(k, j) * s);
00121             }
00122         }
00123         // normalize the vi vectors again
00124         for (size_t i = m; i >= 0; i--) {
00125             A(i, j) = A(i, j) * fak;
00126             if (i == j) {
00127                 break;
00128             }
00129         }
00130     }
00131     // create qi and multiply them
00132     if (m >= n) {
00133         for (size_t j = 0; j < n; j++) {
00134             DenseMatrix<REAL> TempQ(m, m, 0.0);
00135             DenseMatrix<REAL> vl(m, 1, 0.0);
00136             DenseMatrix<REAL> vlt(1, m, 0.0);
00137             hdnum::Vector<double> v__i(m, 0);
00138             for (size_t i = 0; i < m; i++) {
00139                 if (i < j) {
00140                     vl(i, 0) = 0;
00141                 }
00142                 v__i[i] = 0;
00143                 continue;
00144             }

```

```

00145         v1(i, 0) = A(i, j);
00146
00147         v__i[i] = A(i, j);
00148     }
00149     vlt = v1.transpose();
00150
00151     TempQ = (v1 * vlt);
00152
00153     TempQ *= (-2.0);
00154
00155     TempQ /= v__i.two_norm_2();
00156
00157     TempQ += I;
00158     if (show_Hi) {
00159         std::cout << "H[" << j + 1 << "]" << TempQ;
00160     }
00161     if (j == 0) {
00162         Q = TempQ;
00163     }
00164     if (j > 0) {
00165         Q = Q * TempQ;
00166     }
00167 }
00168 }
00169 if (n > m) {
00170     for (size_t j = 0; j < m; j++) {
00171         DenseMatrix<REAL> TempQ(m, m, 0.0);
00172         DenseMatrix<REAL> v1(m, 1, 0.0);
00173         DenseMatrix<REAL> vlt(1, m, 0.0);
00174         hdnum::Vector<double> v__i(m, 0);
00175         for (size_t i = 0; i < m; i++) {
00176             if (i < j) {
00177                 v1(i, 0) = 0;
00178
00179                 v__i[i] = 0;
00180                 continue;
00181             }
00182             v1(i, 0) = A(i, j);
00183
00184             v__i[i] = A(i, j);
00185         }
00186         vlt = v1.transpose();
00187
00188         TempQ = (v1 * vlt);
00189
00190         TempQ *= (-2.0);
00191
00192         TempQ /= v__i.two_norm_2();
00193
00194         TempQ += I;
00195         if (show_Hi) {
00196             std::cout << "H[" << j + 1 << "]" << TempQ;
00197         }
00198         if (j == 0) {
00199             Q = TempQ;
00200         }
00201         if (j > 0) {
00202             Q = Q * TempQ;
00203         }
00204     }
00205 }
00206 return Q;
00207 }
00208 } // namespace hdnum
00209 #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.](#)

```
00001 // -*- tab-width: 4; indent-tabs-mode: nil -*-
00002 #ifndef HDNUM_RUNGEKUTTA_HH
00003 #define HDNUM_RUNGEKUTTA_HH
00004
00005 #include "vector.hh"
00006 #include "newton.hh"
00007
00012 namespace hdnum {
00015     template<class M>
```

```

00016 class ImplicitRungeKuttaStepProblem
00017 {
00018 public:
00020     typedef typename M::size_type size_type;
00021
00023     typedef typename M::time_type time_type;
00024
00026     typedef typename M::number_type number_type;
00027
00029     ImplicitRungeKuttaStepProblem (const M& model_,
00030                                   DenseMatrix<number_type> A_,
00031                                   Vector<number_type> b_,
00032                                   Vector<number_type> c_,
00033                                   time_type t_,
00034                                   Vector<number_type> u_,
00035                                   time_type dt_)
00036     : model(model_) , u(model.size())
00037     {
00038         A = A_;
00039         b = b_;
00040         c = c_;
00041         s = A_.rowsize ();
00042         dt = dt_;
00043         n = model.size();
00044         t = t_;
00045         u = u_;
00046     }
00047
00049     std::size_t size () const
00050     {
00051         return n*s;
00052     }
00053
00055     void F (const Vector<number_type>& x, Vector<number_type>& result) const
00056     {
00057         Vector<Vector<number_type> > xx (s);
00058         for (int i = 0; i < s; i++)
00059         {
00060             xx[i].resize(n, number_type(0));
00061             for(int k = 0; k < n; k++)
00062             {
00063                 xx[i][k] = x[i*n + k];
00064             }
00065         }
00066         Vector<Vector<number_type> > f (s);
00067         for (int i = 0; i < s; i++)
00068         {
00069             f[i].resize(n, number_type(0));
00070             model.f(t + c[i] * dt, u + xx[i], f[i]);
00071         }
00072         Vector<Vector<number_type> > hr (s);
00073         for (int i = 0; i < s; i++)
00074         {
00075             hr[i].resize(n, number_type(0));
00076         }
00077         for (int i = 0; i < s; i++)
00078         {
00079             Vector<number_type> sum (n, number_type(0));
00080             for (int j = 0; j < s; j++)
00081             {
00082                 sum.update(dt*A[i][j], f[j]);
00083             }
00084             hr[i] = xx[i] - sum;
00085         }
00086         //translating hr into result
00087         for (int i = 0; i < s; i++)
00088         {
00089             for (int j = 0; j < n; j++)
00090             {
00091                 result[i*n + j] = hr[i][j];
00092             }
00093         }
00094     }
00095
00097     void F_x (const Vector<number_type>& x, DenseMatrix<number_type>& result) const
00098     {
00099         Vector<Vector<number_type> > xx (s);
00100         for (int i = 0; i < s; i++)
00101         {
00102             xx[i].resize(n);
00103             for(int k = 0; k < n; k++)
00104             {
00105                 xx[i][k] = x[i*n + k];
00106             }
00107         }
00108         DenseMatrix<number_type> I (n, n, 0.0);
00109         for (int i = 0; i < n; i++)

```

```

00110     {
00111         I[i][i] = 1.0;
00112     }
00113     for (int i = 0; i < s; i++)
00114     {
00115         for (int j = 0; j < s; j++)
00116         {
00117             DenseMatrix<number_type> J (n, n, number_type(0));
00118             DenseMatrix<number_type> H (n, n, number_type(0));
00119             model.f_x(t+c[j]*dt, u + xx[j],H);
00120             J.update(-dt*A[i][j],H);
00121             if(i==j) //add I on diagonal
00122             {
00123                 J+=I;
00124             }
00125             for (int k = 0; k < n; k++)
00126             {
00127                 for (int l = 0; l < n; l++)
00128                 {
00129                     result[n * i + k][n * j + l] = J[k][l];
00130                 }
00131             }
00132         }
00133     }
00134 }
00135
00136 private:
00137     const M& model;
00138     time_type t, dt;
00139     Vector<number_type> u;
00140     int n, s; // dimension of matrix A and model.size
00141     DenseMatrix<number_type> A; // A, b, c as in the butcher tableau
00142     Vector<number_type> b;
00143     Vector<number_type> c;
00144 };
00145
00146
00156 template<class M, class S = Newton>
00157 class RungeKutta
00158 {
00159 public:
00161     typedef typename M::size_type size_type;
00162
00164     typedef typename M::time_type time_type;
00165
00167     typedef typename M::number_type number_type;
00168
00170     RungeKutta (const M& model_,
00171                 DenseMatrix<number_type> A_,
00172                 Vector<number_type> b_,
00173                 Vector<number_type> c_)
00174     : model(model_), u(model.size()), w(model.size()), K(A_.rowsize ())
00175     {
00176         A = A_;
00177         b = b_;
00178         c = c_;
00179         s = A_.rowsize ();
00180         n = model.size();
00181         model.initialize(t,u);
00182         dt = 0.1;
00183         for (int i = 0; i < s; i++)
00184         {
00185             K[i].resize(n, number_type(0));
00186         }
00187         sigma = 0.01;
00188         verbosity = 0;
00189
00190         if (A_.rowsize()!=A_.colsize())
00191             HDNUM_ERROR("need square and nonempty matrix");
00192         if (A_.rowsize()!=b_.size())
00193             HDNUM_ERROR("vector incompatible with matrix");
00194         if (A_.colsize()!=c_.size())
00195             HDNUM_ERROR("vector incompatible with matrix");
00196     }
00197
00199     RungeKutta (const M& model_,
00200                 DenseMatrix<number_type> A_,
00201                 Vector<number_type> b_,
00202                 Vector<number_type> c_,
00203                 number_type sigma_)
00204     : model(model_), u(model.size()), w(model.size()), K(A_.rowsize ())
00205     {
00206         A = A_;
00207         b = b_;
00208         c = c_;
00209         s = A_.rowsize ();
00210         n = model.size();

```



```

00211     model.initialize(t,u);
00212     dt = 0.1;
00213     for (int i = 0; i < s; i++)
00214     {
00215         K[i].resize(n, number_type(0));
00216     }
00217     sigma = sigma_;
00218     verbosity = 0;
00219     if (A_.rowsize() != A_.colsize())
00220     HDNUM_ERROR("need square and nonempty matrix");
00221     if (A_.rowsize() != b_.size())
00222     HDNUM_ERROR("vector incompatible with matrix");
00223     if (A_.colsize() != c_.size())
00224     HDNUM_ERROR("vector incompatible with matrix");
00225 }
00226
00228 void set_dt (time_type dt_)
00229 {
00230     dt = dt_;
00231 }
00232
00234 bool check_explicit ()
00235 {
00236     bool is_explicit = true;
00237     for (int i = 0; i < s; i++)
00238     {
00239         for (int j = i; j < s; j++)
00240         {
00241             if (A[i][j] != 0.0)
00242             {
00243                 is_explicit = false;
00244             }
00245         }
00246     }
00247     return is_explicit;
00248 }
00249
00251 void step ()
00252 {
00253     if (check_explicit())
00254     {
00255         // compute k_1
00256         w = u;
00257         model.f(t, w, K[0]);
00258         for (int i = 0; i < s; i++)
00259         {
00260             Vector<number_type> sum (K[0].size(), 0.0);
00261             sum.update(b[0], K[0]);
00262             //compute k_i
00263             for (int j = 0; j < i+1; j++)
00264             {
00265                 sum.update(A[i][j],K[j]);
00266             }
00267             Vector<number_type> wert = w.update(dt,sum);
00268             model.f(t + c[i]*dt, wert, K[i]);
00269             u.update(dt *b[i], K[i]);
00270         }
00271     }
00272     if (not check_explicit())
00273     {
00274         // In the implicit case we need to solve a nonlinear problem
00275         // to do a time step.
00276         ImplicitRungeKuttaStepProblem<M> problem(model, A, b, c, t, u, dt);
00277         bool last_row_eq_b = true;
00278         for (int i = 0; i<s; i++)
00279         {
00280             if (A[s-1][i] != b[i])
00281             {
00282                 last_row_eq_b = false;
00283             }
00284         }
00285
00286         // Solve nonlinear problem and determine coefficients
00287         S solver;
00288         solver.set_maxit(2000);
00289         solver.set_verbosity(verbosity);
00290         solver.set_reduction(1e-10);
00291         solver.set_abslimit(1e-10);
00292         solver.set_linesearchsteps(10);
00293         solver.set_sigma(0.01);
00294         Vector<number_type> zij (s*n,0.0);
00295         solver.solve(problem,zij);
00296
00297         DenseMatrix<number_type> Ainv (s,s,number_type(0));
00298         if (not last_row_eq_b)
00299         {
00300

```

```

00301     // Compute LR decomposition of A
00302     Vector<number_type> w (s, number_type(0));
00303     Vector<number_type> x (s, number_type(0));
00304     Vector<number_type> z (s, number_type(0));
00305     Vector<std::size_t> p(s);
00306     Vector<std::size_t> q(s);
00307     DenseMatrix<number_type> Temp (s,s,0.0);
00308     Temp = A;
00309     row_equilibrate(Temp,w);
00310     lr_fullpivot(Temp,p,q);
00311
00312     // Use LR decomposition to calculate inverse of A
00313     for (int i=0; i<s; i++)
00314     {
00315         Vector<number_type> e (s, number_type(0));
00316         e[i]=number_type(1);
00317         apply_equilibrate(w,e);
00318         permute_forward(p,e);
00319         solveL(Temp,e,e);
00320         solveR(Temp,z,e);
00321         permute_backward(q,z);
00322         for (int j = 0; j < s; j++)
00323         {
00324             Ainv[j][i] = z[j];
00325         }
00326     }
00327 }
00328
00329 Vector<Vector<number_type> > Z (s, 0.0);
00330 for(int i=0; i<s; i++)
00331 {
00332     Vector<number_type> zero(n,number_type(0));
00333     Z[i] = zero;
00334     for (int j = 0; j < n; j++)
00335     {
00336         Z[i][j] = zij[i*n+j];
00337     }
00338 }
00339 if (last_row_eq_b)
00340 {
00341     u += Z[s-1];
00342 }
00343 else
00344 {
00345     // compute ki
00346     Vector<number_type> zero(n,number_type(0));
00347     for (int i = 0; i < s; i++)
00348     {
00349         K[i] = zero;
00350         for (int j=0; j < s; j++)
00351         {
00352             K[i].update(Ainv[i][j],Z[j]);
00353         }
00354         K[i]*= (1.0/dt);
00355
00356         // compute u
00357         u.update(dt*b[i], K[i]);
00358     }
00359 }
00360 }
00361 t = t+dt;
00362 }
00363
00364 void set_state (time_type t_, const Vector<number_type>& u_)
00365 {
00366     t = t_;
00367     u = u_;
00368 }
00369
00370
00371 const Vector<number_type>& get_state () const
00372 {
00373     return u;
00374 }
00375
00376
00377 time_type get_time () const
00378 {
00379     return t;
00380 }
00381
00382
00383 time_type get_dt () const
00384 {
00385     return dt;
00386 }
00387
00388
00389 void set_verbosity(int verbosity_)
00390 {
00391     verbosity = verbosity_;
00392 }

```

```

00393     }
00394
00395 private:
00396     const M& model;
00397     time_type t, dt;
00398     Vector<number_type> u;
00399     Vector<number_type> w;
00400     Vector<Vector<number_type> > K; // save ki
00401     int n; //
00402     dimension of matrix A
00403     int s;
00404     DenseMatrix<number_type> A; // A, b, c as in the butcher
00405     tableau
00406     Vector<number_type> b;
00407     Vector<number_type> c;
00408     number_type sigma;
00409     int verbosity;
00410 };
00411
00412 template<class M, class S>
00413 void ordertest(const M& model,
00414               S solver,
00415               typename M::number_type T,
00416               typename M::number_type h_0,
00417               int l)
00418 {
00419     // Get types
00420     typedef typename M::time_type time_type;
00421     typedef typename M::number_type number_type;
00422
00423     // error_array[i] = ||u(T)-u_i(T)||
00424     number_type error_array[l];
00425
00426     Vector<number_type> exact_solution;
00427     model.exact_solution(T, exact_solution);
00428
00429     for (int i=0; i<l; i++)
00430     {
00431         // Set initial time and value
00432         time_type t_start;
00433         Vector<number_type> initial_solution(1);
00434         model.initialize(t_start, initial_solution);
00435         solver.set_state(t_start, initial_solution);
00436
00437         // Initial time step
00438         time_type dt = h_0/pow(2,i) ;
00439         solver.set_dt(dt);
00440
00441         // Time loop
00442         while (solver.get_time()<T-2*solver.get_dt())
00443         {
00444             solver.step();
00445         }
00446
00447         // Last steps
00448         if (solver.get_time()<T-solver.get_dt())
00449         {
00450             solver.set_dt((T-solver.get_time())/2.0);
00451             for(int i=0; i<2; i++)
00452             {
00453                 solver.step();
00454             }
00455         }
00456         else
00457         {
00458             solver.set_dt(T-solver.get_time());
00459             solver.step();
00460         }
00461
00462         // Error
00463         Vector<number_type> state = solver.get_state();
00464         error_array[i] = norm(exact_solution-state);
00465
00466         if(i==0)
00467         {
00468             std::cout << "dt: "
00469                       << std::scientific << std::showpoint << std::setprecision(8)
00470                       << dt
00471                       << " "
00472                       << "Error: "
00473                       << error_array[0] << std::endl;
00474         }
00475         if(i>0)
00476         {
00477             number_type rate = log(error_array[i-1]/error_array[i])/log(2);
00478             std::cout << "dt: "

```

```

00489         << std::scientific << std::showpoint << std::setprecision(8)
00490         << dt
00491         << " "
00492         << "Error: "
00493         << error_array[i]
00494         << " "
00495         << "Rate: "
00496         << rate << std::endl;
00497     }
00498 }
00499 }
00500
00501 } // namespace hdnun
00502
00503 #endif

```

5.22 sgrid.hh

```

00001 #ifndef HDNUM_SGRID_HH
00002 #define HDNUM_SGRID_HH
00003 #include <limits>
00004 #include <assert.h>
00005
00006 namespace hdnun {
00013     template<class N, class DF, int dimension>
00014     class SGrid
00015     {
00016     public:
00017
00019         typedef std::size_t size_type;
00020
00022         typedef N number_type;
00023
00025         typedef DF DomainFunction;
00026
00027         enum { dim = dimension };
00028
00030         static const int positive = 1;
00031         static const int negative = -1;
00032
00033     private:
00034
00036         const Vector<number_type> extent;
00037         const Vector<size_type> size;
00038         const DomainFunction & df;
00039         Vector<number_type> h;
00040         Vector<size_type> offsets;
00041         std::vector<size_type> node_map;
00042         std::vector<size_type> grid_map;
00043         std::vector<bool> inside_map;
00044         std::vector<bool> boundary_map;
00045
00046         size_t n_nodes;
00047
00048         inline Vector<size_type> index2grid(size_type index) const
00049         {
00050             Vector<size_type> c(dim);
00051             for(int d=dim-1; d>=0; --d){
00052                 c[d] = index / offsets[d];
00053                 index -= c[d] * offsets[d];
00054             }
00055             return c;
00056         }
00057
00058         inline Vector<number_type> grid2world(const Vector<size_type> & c) const
00059         {
00060             Vector<number_type> w(dim);
00061             for(int d=dim-1; d>=0; --d)
00062                 w[d] = c[d] * h[d];
00063             return w;
00064         }
00065
00066         inline Vector<number_type> index2world(size_type index) const
00067         {
00068             Vector<number_type> w(dim);
00069             Vector<size_type> c = index2grid(index);
00070             return grid2world(c);
00071         }
00072
00073     public:
00074
00075

```

```

00077     const size_type invalid_node;
00078
00093     SGrid(const Vector<number_type> extent_,
00094           const Vector<size_type> size_,
00095           const DomainFunction & df_)
00096     : extent(extent_), size(size_), df(df_),
00097       h(dim), offsets(dim),
00098       invalid_node(std::numeric_limits<size_type>::max())
00099     {
00100         // Determine total number of nodes, increment offsets, and cell
00101         // widths.
00102         n_nodes = 1;
00103         offsets.resize(dim);
00104         h.resize(dim);
00105         for(int d=0; d<dim; ++d){
00106             n_nodes *= size[d];
00107             offsets[d] = d==0 ? 1 : size[d-1] * offsets[d-1];
00108             h[d] = extent[d] / number_type(size[d]-1);
00109         }
00110
00111         // Initialize maps.
00112         node_map.resize(0);
00113         inside_map.resize(n_nodes);
00114         grid_map.resize(n_nodes);
00115         boundary_map.resize(0);
00116         boundary_map.resize(n_nodes, false);
00117
00118         for(size_type n=0; n<n_nodes; ++n){
00119             Vector<size_type> c = index2grid(n);
00120             Vector<number_type> x = grid2world(c);
00121
00122             inside_map[n] = df.evaluate(x);
00123             if(inside_map[n]){
00124                 node_map.push_back(n);
00125                 grid_map[n] = node_map.size()-1;
00126             }
00127             else
00128                 grid_map[n] = invalid_node;
00129         }
00130
00131         // Find boundary nodes
00132         for(size_type n=0; n<node_map.size(); ++n){
00133             for(int d=0; d<dim; ++d){
00134                 for(int s=0; s<2; ++s){
00135                     const int side = s*2-1;
00136                     const size_type neighbor = getNeighborIndex(n,d,side,1);
00137                     if(neighbor == invalid_node)
00138                         boundary_map[node_map[n]] = true;
00139                 }
00140             }
00141         }
00142
00143     }
00144
00164     size_type getNeighborIndex(const size_type ln, const size_type n_dim, const int n_side, const int
k = 1) const
00165     {
00166         const size_type n = node_map[ln];
00167         const Vector<size_type> c = index2grid(n);
00168         size_type neighbors[2];
00169         neighbors[0] = c[n_dim];
00170         neighbors[1] = size[n_dim]-c[n_dim]-1;
00171
00172         assert(n_side == 1 || n_side == -1);
00173         if(size_type(k) > neighbors[(n_side+1)/2])
00174             return invalid_node;
00175
00176         const size_type neighbor = n + offsets[n_dim] * n_side * k;
00177
00178         if(!inside_map[neighbor])
00179             return invalid_node;
00180
00181         return grid_map[neighbor];
00182     }
00183
00187     bool isBoundaryNode(const size_type ln) const
00188     {
00189         return boundary_map[node_map[ln]];
00190     }
00191
00195     size_type getNumberOfNodes() const
00196     {
00197         return node_map.size();
00198     }
00199
00200     Vector<size_type> getGridSize() const
00201     {

```

```

00202     return size;
00203 }
00204
00207 Vector<number_type> getCellWidth() const
00208 {
00209     return h;
00210 }
00211
00215 Vector<number_type> getCoordinates(const size_type ln) const
00216 {
00217     return index2world(node_map[ln]);
00218 }
00219
00220 std::vector<Vector<number_type> > getNodeCoordinates() const
00221 {
00222     std::vector<Vector<number_type> > coords;
00223     for(size_type n=0; n<node_map.size(); ++n){
00224         coords.push_back(Vector<number_type>(dim));
00225         coords.back() = index2world(node_map[n]);
00226     }
00227     return coords;
00228 }
00229 };
00230 };
00231
00232 }
00233
00234 #endif // HDNUM_SGRID_HH

```

5.23 sparsematrix.hh

```

00001 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
00002 /*
00003  * File:   sparsematrix.hh
00004  * Author: Christian Heusel <christian@heusel.eu>
00005  *
00006  * Created on August 25, 2020
00007  */
00008
00009 #ifndef SPARSEMATRIX_HH
00010 #define SPARSEMATRIX_HH
00011
00012 #include <algorithm>
00013 #include <complex>
00014 #include <functional>
00015 #include <iomanip>
00016 #include <iostream>
00017 #include <map>
00018 #include <numeric>
00019 #include <string>
00020 #include <type_traits>
00021 #include <vector>
00022
00023 #include "densematrix.hh"
00024 #include "vector.hh"
00025
00026 namespace hdnum {
00027
00030 template <typename REAL>
00031 class SparseMatrix {
00032 public:
00034     using size_type = std::size_t;
00035
00037     using column_iterator = typename std::vector<REAL>::iterator;
00039     using const_column_iterator = typename std::vector<REAL>::const_iterator;
00040
00041 private:
00042     // Matrix data is stored in an STL vector!
00043     std::vector<REAL> _data;
00044
00045     // The non-null indices are stored in STL vectors with the size_type!
00046     // Explanation on how the mapping works can be found here:
00047     // https://de.wikipedia.org/wiki/Compressed_Row_Storage
00048     std::vector<size_type> _colIndices;
00049     std::vector<size_type> _rowPtr;
00050
00051     size_type m_rows = 0; // Number of Matrix rows
00052     size_type m_cols = 0; // Number of Matrix columns
00053
00054     static bool bScientific;
00055     static size_type nIndexWidth;
00056     static size_type nValueWidth;
00057     static size_type nValuePrecision;

```

```

00058     static const REAL _zero;
00059
00060     // !function that converts container contents into
00061     // { 1, 2, 3, 4 }
00062     template <typename T>
00063     [[nodiscard]] std::string comma_fold(T container) const {
00064         return "{ " +
00065             std::accumulate(
00066                 std::next(container.cbegin()), container.cend(),
00067                 std::to_string(container[0]), // start with first element
00068                 [](const std::string &a, REAL b) {
00069                     return a + ", " + std::to_string(b);
00070                 }) +
00071             " }";
00072     }
00073
00074     // This code was copied from StackOverflow to generalize a check whether a
00075     // template is a specialization i.e. for std::complex
00076     // https://stackoverflow.com/questions/31762958/check-if-class-is-a-template-specialization
00077     template <class T, template <class...> class Template>
00078     struct is_specialization : std::false_type {};
00079
00080     template <template <class...> class Template, class... Args>
00081     struct is_specialization<Template<Args...>, Template> : std::true_type {};
00082
00083     bool checkIfAccessIsInBounds(const size_type row_index,
00084                                 const size_type col_index) const {
00085         if (not (row_index < m_rows)) {
00086             HDNUM_ERROR("Out of bounds access: row too big! -> " +
00087                 std::to_string(row_index) + " is not < " +
00088                 std::to_string(m_rows));
00089             return false;
00090         }
00091         if (not (col_index < m_cols)) {
00092             HDNUM_ERROR("Out of bounds access: column too big! -> " +
00093                 std::to_string(col_index) + " is not < " +
00094                 std::to_string(m_cols));
00095             return false;
00096         }
00097         return true;
00098     }
00099
00100 public:
00101     SparseMatrix() = default;
00102
00103     SparseMatrix(const size_type _rows, const size_type _cols)
00104         : _rowPtr(_rows + 1), m_rows(_rows), m_cols(_cols) {}
00105
00106     [[nodiscard]] size_type rowsize() const { return m_rows; }
00107
00108     [[nodiscard]] size_type colsize() const { return m_cols; }
00109
00110     [[nodiscard]] bool scientific() const { return bScientific; }
00111
00112     class column_index_iterator {
00113     public:
00114         using self_type = column_index_iterator;
00115
00116         // conform to the iterator traits
00117         // https://en.cppreference.com/w/cpp/iterator/iterator_traits
00118         using difference_type = std::ptrdiff_t;
00119         using value_type = std::pair<REAL &, size_type const &>;
00120         using pointer = value_type *;
00121         using reference = value_type &;
00122         using iterator_category = std::bidirectional_iterator_tag;
00123
00124         column_index_iterator(typename std::vector<REAL>::iterator valIter,
00125                               std::vector<size_type>::iterator colIndicesIter)
00126             : _valIter(valIter), _colIndicesIter(colIndicesIter) {}
00127
00128         // prefix
00129         self_type &operator++() {
00130             _valIter++;
00131             _colIndicesIter++;
00132             return *this;
00133         }
00134
00135         // postfix
00136         self_type &operator++(int junk) {
00137             self_type cached = *this;
00138             _valIter++;
00139             _colIndicesIter++;
00140             return cached;
00141         }
00142
00143         [[nodiscard]] value_type operator*() {
00144             return std::make_pair(std::ref(*_valIter),

```

```

00193         std::cref(*_colIndicesIter));
00194     }
00195     // [[nodiscard]] value_type operator->() {
00196     //     return std::make_pair(std::ref(*_valIter),
00197     //         std::cref(*_colIndicesIter));
00198     // }
00199
00200     [[nodiscard]] typename value_type::first_type value() {
00201         return std::ref(*_valIter);
00202     }
00203
00204     [[nodiscard]] typename value_type::second_type index() {
00205         return std::cref(*_colIndicesIter);
00206     }
00207
00208     [[nodiscard]] bool operator==(const self_type &other) {
00209         return (_valIter == other._valIter) and
00210             (_colIndicesIter == other._colIndicesIter);
00211     }
00212     [[nodiscard]] bool operator!=(const self_type &other) {
00213         return not (*this == other);
00214     }
00215
00216 private:
00217     typename std::vector<REAL>::iterator _valIter;
00218     std::vector<size_type>::iterator _colIndicesIter;
00219 };
00220
00221 class const_column_index_iterator {
00222 public:
00223     using self_type = const_column_index_iterator;
00224
00225     // conform to the iterator traits
00226     // https://en.cppreference.com/w/cpp/iterator/iterator_traits
00227     using difference_type = std::ptrdiff_t;
00228     using value_type = std::pair<REAL const &, size_type const &>;
00229     using pointer = value_type *;
00230     using reference = value_type &;
00231     using iterator_category = std::bidirectional_iterator_tag;
00232
00233     const_column_index_iterator(
00234         typename std::vector<REAL>::const_iterator valIter,
00235         std::vector<size_type>::const_iterator colIndicesIter)
00236         : _valIter(valIter), _colIndicesIter(colIndicesIter) {}
00237
00238     // prefix
00239     self_type &operator++() {
00240         _valIter++;
00241         _colIndicesIter++;
00242         return *this;
00243     }
00244
00245     // postfix
00246     self_type operator++(int junk) {
00247         self_type cached = *this;
00248         _valIter++;
00249         _colIndicesIter++;
00250         return cached;
00251     }
00252
00253     [[nodiscard]] value_type operator*() {
00254         return std::make_pair(std::ref(*_valIter),
00255             std::cref(*_colIndicesIter));
00256     }
00257     // TODO: This is wrong
00258     // [[nodiscard]] value_type operator->() {
00259     //     return std::make_pair(*_valIter, *_colIndicesIter);
00260     // }
00261
00262     [[nodiscard]] typename value_type::first_type value() {
00263         return std::ref(*_valIter);
00264     }
00265
00266     [[nodiscard]] typename value_type::second_type index() {
00267         return std::cref(*_colIndicesIter);
00268     }
00269
00270     [[nodiscard]] bool operator==(const self_type &other) {
00271         return (_valIter == other._valIter) and
00272             (_colIndicesIter == other._colIndicesIter);
00273     }
00274     [[nodiscard]] bool operator!=(const self_type &other) {
00275         return not (*this == other);
00276     }
00277
00278 private:
00279     typename std::vector<REAL>::const_iterator _valIter;

```



```

00280     std::vector<size_type>::const_iterator _colIndicesIter;
00281 };
00282
00283 class row_iterator {
00284 public:
00285     using self_type = row_iterator;
00286
00287     // conform to the iterator traits
00288     // https://en.cppreference.com/w/cpp/iterator/iterator_traits
00289     using difference_type = std::ptrdiff_t;
00290     using value_type = self_type;
00291     using pointer = self_type *;
00292     using reference = self_type &;
00293     using iterator_category = std::random_access_iterator_tag;
00294
00295     row_iterator(std::vector<size_type>::iterator rowPtrIter,
00296                 std::vector<size_type>::iterator colIndicesIter,
00297                 typename std::vector<REAL>::iterator valIter)
00298     : _rowPtrIter(rowPtrIter), _colIndicesIter(colIndicesIter),
00299       _valIter(valIter) {}
00300
00301     [[nodiscard]] column_iterator begin() {
00302         return column_iterator((_valIter + *_rowPtrIter));
00303     }
00304     [[nodiscard]] column_iterator end() {
00305         return column_iterator((_valIter + *(_rowPtrIter + 1)));
00306     }
00307
00308     [[nodiscard]] column_index_iterator ibegin() {
00309         return column_index_iterator((_valIter + *_rowPtrIter),
00310                                     (_colIndicesIter + *_rowPtrIter));
00311     }
00312     [[nodiscard]] column_index_iterator iend() {
00313         return column_index_iterator(
00314             (_valIter + *(_rowPtrIter + 1)),
00315             (_colIndicesIter + *(_rowPtrIter + 1)));
00316     }
00317
00318     // prefix
00319     self_type &operator++() {
00320         _rowPtrIter++;
00321         return *this;
00322     }
00323
00324     // postfix
00325     self_type operator++(int junk) {
00326         self_type cached = *this;
00327         _rowPtrIter++;
00328         return cached;
00329     }
00330
00331     self_type &operator+=(difference_type offset) {
00332         _rowPtrIter += offset;
00333         return *this;
00334     }
00335
00336     self_type &operator-=(difference_type offset) {
00337         _rowPtrIter -= offset;
00338         return *this;
00339     }
00340
00341     // iter - n
00342     self_type operator-(difference_type offset) {
00343         self_type cache(*this);
00344         cache -= offset;
00345         return cache;
00346     }
00347
00348     // iter + n
00349     self_type operator+(difference_type offset) {
00350         self_type cache(*this);
00351         cache += offset;
00352         return cache;
00353     }
00354     // n + iter
00355     friend self_type operator+(const difference_type &offset,
00356                               const self_type &sec) {
00357         self_type cache(sec);
00358         cache += offset;
00359         return cache;
00360     }
00361
00362     reference operator[](difference_type offset) {
00363         return *(*this + offset);
00364     }
00365
00366     bool operator<(const self_type &other) {

```

```

00367         return other - (*this) > 0; //
00368     }
00369
00370     bool operator>(const self_type &other) {
00371         return other < (*this); //
00372     }
00373
00374     [[nodiscard]] self_type &operator*() { return *this; }
00375     // [[nodiscard]] self_type &operator->() { return *this; }
00376
00377     [[nodiscard]] bool operator==(const self_type &rhs) {
00378         return _rowPtrIter == rhs._rowPtrIter;
00379     }
00380     [[nodiscard]] bool operator!=(const self_type &rhs) {
00381         return _rowPtrIter != rhs._rowPtrIter;
00382     }
00383
00384 private:
00385     std::vector<size_type>::iterator _rowPtrIter;
00386     std::vector<size_type>::iterator _colIndicesIter;
00387     typename std::vector<REAL>::iterator _valIter;
00388 };
00389
00390 class const_row_iterator {
00391 public:
00392     using self_type = const_row_iterator;
00393
00394     // conform to the iterator traits
00395     // https://en.cppreference.com/w/cpp/iterator/iterator_traits
00396     using difference_type = std::ptrdiff_t;
00397     using value_type = self_type;
00398     using pointer = self_type *;
00399     using reference = self_type &;
00400     using iterator_category = std::bidirectional_iterator_tag;
00401
00402     const_row_iterator(
00403         std::vector<size_type>::const_iterator rowPtrIter,
00404         std::vector<size_type>::const_iterator colIndicesIter,
00405         typename std::vector<REAL>::const_iterator valIter)
00406         : _rowPtrIter(rowPtrIter), _colIndicesIter(colIndicesIter),
00407           _valIter(valIter) {}
00408
00409     [[nodiscard]] const_column_iterator begin() const {
00410         return const_column_iterator((_valIter + *_rowPtrIter));
00411     }
00412     [[nodiscard]] const_column_iterator end() const {
00413         return const_column_iterator((_valIter + *(_rowPtrIter + 1)));
00414     }
00415
00416     [[nodiscard]] const_column_index_iterator ibegin() const {
00417         return const_column_index_iterator(
00418             (_valIter + *_rowPtrIter), (_colIndicesIter + *_rowPtrIter));
00419     }
00420     [[nodiscard]] const_column_index_iterator iend() const {
00421         return const_column_index_iterator(
00422             (_valIter + *(_rowPtrIter + 1)),
00423             (_colIndicesIter + *(_rowPtrIter + 1)));
00424     }
00425
00426     [[nodiscard]] const_column_iterator cbegin() const {
00427         return this->begin();
00428     }
00429     [[nodiscard]] const_column_iterator cend() const {
00430         return this->end(); //
00431     }
00432
00433     // prefix
00434     self_type &operator++() {
00435         _rowPtrIter++;
00436         return *this;
00437     }
00438
00439     // postfix
00440     self_type &operator++(int junk) {
00441         self_type cached = *this;
00442         _rowPtrIter++;
00443         return cached;
00444     }
00445
00446     self_type &operator+=(difference_type offset) {
00447         _rowPtrIter += offset;
00448         return *this;
00449     }
00450
00451     self_type &operator-=(difference_type offset) {
00452         _rowPtrIter -= offset;
00453         return *this;

```

```

00454     }
00455
00456     // iter - n
00457     self_type operator-(difference_type offset) {
00458         self_type cache(*this);
00459         cache -= offset;
00460         return cache;
00461     }
00462
00463     // iter + n
00464     self_type operator+(difference_type offset) {
00465         self_type cache(*this);
00466         cache += offset;
00467         return cache;
00468     }
00469     // n + iter
00470     friend self_type operator+(const difference_type &offset,
00471                               const self_type &sec) {
00472         self_type cache(sec);
00473         cache += offset;
00474         return cache;
00475     }
00476
00477     reference operator[](difference_type offset) {
00478         return *(*this + offset);
00479     }
00480
00481     bool operator<(const self_type &other) {
00482         return other - (*this) > 0; //
00483     }
00484
00485     bool operator>(const self_type &other) {
00486         return other < (*this); //
00487     }
00488
00489     [[nodiscard]] self_type &operator*() { return *this; }
00490     // [[nodiscard]] self_type &operator->() { return this; }
00491
00492     [[nodiscard]] bool operator==(const self_type &rhs) {
00493         return _rowPtrIter == rhs._rowPtrIter;
00494     }
00495     [[nodiscard]] bool operator!=(const self_type &rhs) {
00496         return _rowPtrIter != rhs._rowPtrIter;
00497     }
00498
00499     private:
00500     std::vector<size_type>::const_iterator _rowPtrIter;
00501     std::vector<size_type>::const_iterator _colIndicesIter;
00502     typename std::vector<REAL>::const_iterator _valIter;
00503 };
00504
00523 [[nodiscard]] row_iterator begin() {
00524     return row_iterator(_rowPtr.begin(), _colIndices.begin(),
00525                        _data.begin());
00526 }
00527
00546 [[nodiscard]] row_iterator end() {
00547     return row_iterator(_rowPtr.end() - 1, _colIndices.begin(),
00548                        _data.begin());
00549 }
00550
00556 [[nodiscard]] const_row_iterator cbegin() const {
00557     return const_row_iterator(_rowPtr.cbegin(), _colIndices.cbegin(),
00558                              _data.cbegin());
00559 }
00560
00566 [[nodiscard]] const_row_iterator cend() const {
00567     return const_row_iterator(_rowPtr.cend() - 1, _colIndices.cbegin(),
00568                              _data.cbegin());
00569 }
00570
00572 [[nodiscard]] const_row_iterator begin() const { return this->cbegin(); }
00573 [[nodiscard]] const_row_iterator end() const { return this->cend(); }
00574
00600 void scientific(bool b) const { bScientific = b; }
00601
00603 size_type iwidth() const { return nIndexWidth; }
00604
00606 size_type width() const { return nValueWidth; }
00607
00609 size_type precision() const { return nValuePrecision; }
00610
00612 void iwidth(size_type i) const { nIndexWidth = i; }
00613
00615 void width(size_type i) const { nValueWidth = i; }
00616
00618 void precision(size_type i) const { nValuePrecision = i; }

```

```

00619
00620 column_iterator find(const size_type row_index,
00621                    const size_type col_index) const {
00622     checkIfAccessIsInBounds(row_index, col_index);
00623
00624     using value_pair = typename const_column_index_iterator::value_type;
00625     auto row = const_row_iterator(_rowPtr.begin() + row_index,
00626                                 _colIndices.begin(), _data.begin());
00627     return std::find_if(row.ibegin(), row.iend(),
00628                       [col_index](value_pair el) {
00629                             // only care for the index here since the value
00630                             // is unknown
00631                             return el.second == col_index;
00632                         });
00633 }
00634
00635 bool exists(const size_type row_index, const size_type col_index) const {
00636     auto row = const_row_iterator(_rowPtr.begin() + row_index,
00637                                 _colIndices.begin(), _data.begin());
00638     return find(row_index, col_index) != row.iend();
00639 }
00640
00641 REAL &get(const size_type row_index, const size_type col_index) {
00642     checkIfAccessIsInBounds(row_index, col_index);
00643     // look for the entry
00644     using value_pair = typename const_column_index_iterator::value_type;
00645     auto row = row_iterator(_rowPtr.begin() + row_index,
00646                            _colIndices.begin(), _data.begin());
00647
00648     auto result =
00649         std::find_if(row.ibegin(), row.iend(), [col_index](value_pair el) {
00650             // only care for the index here
00651             // since the value is unknown
00652             // anyways
00653             return el.second == col_index;
00654         });
00655     // we found something within the right row
00656     if (result != row.iend()) {
00657         return result.value();
00658     }
00659     throw std::out_of_range(
00660         "There is no non-zero element for these given indices!");
00661 }
00662
00663 const REAL &operator()(const size_type row_index,
00664                       const size_type col_index) const {
00665     checkIfAccessIsInBounds(row_index, col_index);
00666
00667     using value_pair = typename const_column_index_iterator::value_type;
00668     auto row = const_row_iterator(_rowPtr.begin() + row_index,
00669                                 _colIndices.begin(), _data.begin());
00670
00671     auto result =
00672         std::find_if(row.ibegin(), row.iend(), [col_index](value_pair el) {
00673             // only care for the index here since the value is unknown
00674             return el.second == col_index;
00675         });
00676     // we found something within the right row
00677     if (result != row.iend()) {
00678         return result.value();
00679     }
00680     return _zero;
00681 }
00682
00683 [[nodiscard]] bool operator==(const SparseMatrix &other) const {
00684     return (_data == other._data) and //
00685            (_rowPtr == other._rowPtr) and //
00686            (_colIndices == other._colIndices) and //
00687            (m_cols == other.m_cols) and //
00688            (m_rows == other.m_rows);
00689 }
00690
00691 [[nodiscard]] bool operator!=(const SparseMatrix &other) const {
00692     return not (*this == other);
00693 }
00694
00695 // delete all the invalid comparisons
00696 bool operator<(const SparseMatrix &other) = delete;
00697 bool operator>(const SparseMatrix &other) = delete;
00698 bool operator<=(const SparseMatrix &other) = delete;
00699 bool operator>=(const SparseMatrix &other) = delete;
00700
00701 SparseMatrix transpose() const {
00702     // TODO: remove / find bug here!
00703     SparseMatrix::builder builder(m_cols, m_rows);
00704     SparseMatrix::size_type curr_row = 0;
00705     for (auto &row : (*this)) {
00706         for (auto it = row.ibegin(); it != row.iend(); it++) {
00707             builder.addEntry(it.index(), curr_row, it.value());
00708         }
00709         curr_row++;
00710     }
00711     return builder.get();
00712 }

```

```

00710     }
00711     curr_row++;
00712 }
00713
00714     return builder.build();
00715 }
00716
00717 [[nodiscard]] SparseMatrix operator*=(const REAL scalar) {
00720     // This could also be done out of order
00721     std::transform(_data.cbegin(), _data.cend(), _data.begin(),
00722         [&](REAL value) { return value * scalar; });
00723 }
00724
00725 [[nodiscard]] SparseMatrix operator/=(const REAL scalar) {
00728     // This could also be done out of order
00729     std::transform(_data.cbegin(), _data.cend(), _data.begin(),
00730         [&](REAL value) { return value / scalar; });
00731 }
00732
00733 template <class V>
00734 void mv(Vector<V> &result, const Vector<V> &x) const {
00737     static_assert(std::is_convertible<V, REAL>::value,
00738         "The types in the Matrix vector multiplication cant be "
00739         "converted properly!");
00740
00741     if (result.size() != this->colsize()) {
00742         HDNUM_ERROR(
00743             (std::string("The result vector has the wrong dimension! ") +
00744             "Vector dimension " + std::to_string(result.size()) +
00745             " != " + std::to_string(this->colsize()) + " colsize"));
00746     }
00747
00748     if (x.size() != this->colsize()) {
00749         HDNUM_ERROR(
00750             (std::string("The input vector has the wrong dimension! ") +
00751             "Vector dimension " + std::to_string(x.size()) +
00752             " != " + std::to_string(this->colsize()) + " colsize"));
00753     }
00754
00755     size_type curr_row = 0;
00756     for (auto row : (*this)) {
00757         result[curr_row] = std::accumulate(
00758             row.ibegin(), row.iend(), V {}, [&](V result, auto el) -> V {
00759                 return result + (x[el.second] * el.first);
00760             });
00761         curr_row++;
00762     }
00763 }
00764
00765 [[nodiscard]] Vector<REAL> operator*(const Vector<REAL> &x) const {
00768     hdnum::Vector<REAL> result(this->colsize(), 0);
00769     this->mv(result, x);
00770     return result;
00771 }
00772
00773 template <class V>
00774 void umv(Vector<V> &result, const Vector<V> &x) const {
00777     static_assert(std::is_convertible<V, REAL>::value,
00778         "The types in the Matrix vector multiplication cant be "
00779         "converted properly!");
00780
00781     if (result.size() != this->colsize()) {
00782         HDNUM_ERROR(
00783             (std::string("The result vector has the wrong dimension! ") +
00784             "Vector dimension " + std::to_string(result.size()) +
00785             " != " + std::to_string(this->colsize()) + " colsize"));
00786     }
00787
00788     if (x.size() != this->colsize()) {
00789         HDNUM_ERROR(
00790             (std::string("The input vector has the wrong dimension! ") +
00791             "Vector dimension " + std::to_string(result.size()) +
00792             " != " + std::to_string(this->colsize()) + " colsize"));
00793     }
00794
00795     size_type curr_row {};
00796     for (auto row : (*this)) {
00797         result[curr_row] += std::accumulate(
00798             row.ibegin(), row.iend(), V {}, [&](V result, auto el) -> V {
00799                 return result + (x[el.second] * el.first);
00800             });
00801         curr_row++;
00802     }
00803 }
00804
00805 private:
00806     template <typename norm_type>

```

```

00824     norm_type norm_infty_impl() const {
00825         norm_type norm {};
00826         for (auto row : *this) {
00827             norm_type rowsum =
00828                 std::accumulate(row.begin(), row.end(), norm_type {},
00829                                 [](norm_type res, REAL value) -> norm_type {
00830                                     return res + std::abs(value);
00831                                 });
00832             if (norm < rowsum) {
00833                 norm = rowsum;
00834             }
00835         }
00836         return norm;
00837     }
00838
00839 public:
00840     auto norm_infty() const {
00841         if constexpr (is_specialization<REAL, std::complex> {}) {
00842             return norm_infty_impl<double>();
00843         } else {
00844             return norm_infty_impl<REAL>();
00845         }
00846     }
00847
00848     [[nodiscard]] std::string to_string() const noexcept {
00849         return "values=" + comma_fold(_data) + "\n" + //
00850             "colInd=" + comma_fold(_colIndices) + "\n" + //
00851             "rowPtr=" + comma_fold(_rowPtr) + "\n"; //
00852     }
00853
00854     void print() const noexcept { std::cout << *this; }
00855
00856     static SparseMatrix identity(const size_type dimN) {
00857         auto builder = typename SparseMatrix<REAL>::builder(dimN, dimN);
00858         for (typename SparseMatrix<REAL>::size_type i = 0; i < dimN; ++i) {
00859             builder.addEntry(i, i, REAL {1});
00860         }
00861         return builder.build();
00862     }
00863
00864     SparseMatrix<REAL> matchingIdentity() const { return identity(m_cols); }
00865
00866     class builder {
00867     public:
00868         size_type m_rows {}; // Number of Matrix rows, 0 by default
00869         size_type m_cols {}; // Number of Matrix columns, 0 by default
00870         std::vector<std::map<size_type, REAL>> _rows;
00871
00872         builder(size_type new_m_rows, size_type new_m_cols)
00873             : m_rows {new_m_rows}, m_cols {new_m_cols}, _rows {m_rows} {}
00874
00875         builder(const std::initializer_list<std::initializer_list<REAL>> &v)
00876             : m_rows {v.size()}, m_cols {v.begin()->size()}, _rows {m_rows} {
00877             size_type i = 0;
00878             for (auto &row : v) {
00879                 size_type j = 0;
00880                 for (const REAL &element : row) {
00881                     addEntry(i, j, element);
00882                     j++;
00883                 }
00884                 i++;
00885             }
00886         }
00887
00888         builder() = default;
00889
00890         std::pair<typename std::map<size_type, REAL>::iterator, bool> addEntry(
00891             size_type i, size_type j, REAL value) {
00892             return _rows.at(i).emplace(j, value);
00893         }
00894
00895         std::pair<typename std::map<size_type, REAL>::iterator, bool> addEntry(
00896             size_type i, size_type j) {
00897             return addEntry(i, j, REAL {});
00898         };
00899
00900         [[nodiscard]] bool operator==(
00901             const SparseMatrix::builder &other) const {
00902             return (m_rows == other.m_rows) and //
00903                 (m_cols == other.m_cols) and //
00904                 (_rows == other._rows);
00905         }
00906
00907         [[nodiscard]] bool operator!=(
00908             const SparseMatrix::builder &other) const {
00909             return not (*this == other);
00910         }
00911     };

```

```

00965
00966     [[nodiscard]] size_type colsize() const noexcept { return m_cols; }
00967     [[nodiscard]] size_type rowsize() const noexcept { return m_rows; }
00968
00969     size_type setNumCols(size_type new_m_cols) noexcept {
00970         m_cols = new_m_cols;
00971         return m_cols;
00972     }
00973     size_type setNumRows(size_type new_m_rows) {
00974         m_rows = new_m_rows;
00975         _rows.resize(m_cols);
00976         return m_rows;
00977     }
00978
00979     void clear() noexcept {
00980         for (auto &row : _rows) {
00981             row.clear();
00982         }
00983     }
00984
00985     [[nodiscard]] std::string to_string() const {
00986         std::string output;
00987         for (std::size_t i = 0; i < _rows.size(); i++) {
00988             for (const auto &indexpair : _rows[i]) {
00989                 output += std::to_string(i) + ", " +
00990                     std::to_string(indexpair.first) + " => " +
00991                     std::to_string(indexpair.second) + "\n";
00992             }
00993         }
00994         return output;
00995     }
00996
00997     [[nodiscard]] SparseMatrix build() {
00998         auto result = SparseMatrix<REAL>(m_rows, m_cols);
00999
01000         for (std::size_t i = 0; i < _rows.size(); i++) {
01001             result._rowPtr[i + 1] = result._rowPtr[i];
01002             for (const auto &indexpair : _rows[i]) {
01003                 result._colIndices.push_back(indexpair.first);
01004                 result._data.push_back(indexpair.second);
01005                 result._rowPtr[i + 1]++;
01006             }
01007         }
01008         return result;
01009     }
01010 };
01011 };
01012
01013 template <typename REAL>
01014 bool SparseMatrix<REAL>::bScientific = true;
01015 template <typename REAL>
01016 std::size_t SparseMatrix<REAL>::nIndexWidth = 10;
01017 template <typename REAL>
01018 std::size_t SparseMatrix<REAL>::nValueWidth = 10;
01019 template <typename REAL>
01020 std::size_t SparseMatrix<REAL>::nValuePrecision = 3;
01021 template <typename REAL>
01022 const REAL SparseMatrix<REAL>::_zero {};
01023
01024 template <typename REAL>
01025 std::ostream &operator<<(std::ostream &s, const SparseMatrix<REAL> &A) {
01026     using size_type = typename SparseMatrix<REAL>::size_type;
01027
01028     s << std::endl;
01029     s << " " << std::setw(A.iwidth()) << " "
01030     << " ";
01031     for (size_type j = 0; j < A.colsize(); ++j) {
01032         s << std::setw(A.width()) << j << " ";
01033     }
01034     s << std::endl;
01035
01036     for (size_type i = 0; i < A.rowsize(); ++i) {
01037         s << " " << std::setw(A.iwidth()) << i << " ";
01038         for (size_type j = 0; j < A.colsize(); ++j) {
01039             if (A.scientific()) {
01040                 s << std::setw(A.width()) << std::scientific << std::showpoint
01041                 << std::setprecision(A.precision()) << A(i, j) << " ";
01042             } else {
01043                 s << std::setw(A.width()) << std::fixed << std::showpoint
01044                 << std::setprecision(A.precision()) << A(i, j) << " ";
01045             }
01046         }
01047         s << std::endl;
01048     }
01049     return s;
01050 }
01051

```

```

01053 template <typename REAL>
01054 inline void zero(SparseMatrix<REAL> &A) {
01055     A = SparseMatrix<REAL>(A.rowsize(), A.colsize());
01056 }
01057
01091 template <class REAL>
01092 inline void identity(SparseMatrix<REAL> &A) {
01093     if (A.rowsize() != A.colsize()) {
01094         HDNUM_ERROR("Will not overwrite A since Dimensions are not equal!");
01095     }
01096     A = SparseMatrix<REAL>::identity(A.colsize());
01097 }
01098
01099 template <typename REAL>
01100 inline void readMatrixFromFile(const std::string &filename,
01101                               SparseMatrix<REAL> &A) {
01102     // Format taken from here:
01103     // https://math.nist.gov/MatrixMarket/formats.html#coord
01104
01105     using size_type = typename SparseMatrix<REAL>::size_type;
01106     std::string buffer;
01107     std::ifstream fin(filename);
01108     size_type i = 0;
01109     size_type j = 0;
01110     size_type non_zeros = 0;
01111
01112     if (fin.is_open()) {
01113         // ignore all comments from the file (starting with %)
01114         while (fin.peek() == '%') fin.ignore(2048, '\n');
01115
01116         std::getline(fin, buffer);
01117         std::istringstream first_line(buffer);
01118         first_line >> i >> j >> non_zeros;
01119
01120         auto builder = typename SparseMatrix<REAL>::builder(i, j);
01121
01122         while (std::getline(fin, buffer)) {
01123             std::istringstream iss(buffer);
01124
01125             REAL value {};
01126             iss >> i >> j >> value;
01127             // i-1, j-1, because matrix market does not use zero based indexing
01128             builder.addEntry(i - 1, j - 1, value);
01129         }
01130         A = builder.build();
01131         fin.close();
01132     } else {
01133         HDNUM_ERROR(("Could not osspen file! \"" + filename + "\""));
01134     }
01135 }
01136
01137 } // namespace hdnum
01138
01139 #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.](#)

```
00001 #ifndef DUNE_TIMER_HH
00002 #define DUNE_TIMER_HH
00003
00004 #ifndef TIMER_USE_STD_CLOCK
00005 // headers for getrusage(2)
00006 #include <sys/resource.h>
00007 #endif
00008
00009 #include <ctime>
00010
00011 // headers for stderr(3)
00012 #include <cstring>
00013
00014 // access to errno in C++
00015 #include <cerrno>
00016
00017 #include "exceptions.hh"
00018
00019 namespace hdnum {
00020
00021     class TimerError : public SystemError {} ;
00022
00023
00024
00025
00026
00027
00028
00029
00030
00031
00032
00033
00034
00035
00036
00037
00038
00039
00040
00041     class Timer
00042     {
00043     public:
00044         Timer ()
00045         {
00046             reset();
00047         }
00048
00049         void reset()
00050         {
00051             #ifdef TIMER_USE_STD_CLOCK
00052                 cstart = std::clock();
00053             #else
00054                 rusage ru;
00055                 if (getrusage(RUSAGE_SELF, &ru))
00056                     HDNUM_THROW(TimerError, strerror(errno));
00057                 cstart = ru.ru_utime;
00058             #endif
00059         }
00060
00061         double elapsed () const
00062         {
00063             #ifdef TIMER_USE_STD_CLOCK
00064                 return (std::clock()-cstart) / static_cast<double>(CLOCKS_PER_SEC);
00065             #else
00066                 rusage ru;
00067                 if (getrusage(RUSAGE_SELF, &ru))
00068                     HDNUM_THROW(TimerError, strerror(errno));
00069                 return 1.0 * (ru.ru_utime.tv_sec - cstart.tv_sec) + (ru.ru_utime.tv_usec - cstart.tv_usec) /
00070                    (1000.0 * 1000.0);
00071             #endif
00072         }
00073     private:
00074         #ifdef TIMER_USE_STD_CLOCK
00075             std::clock_t cstart;
00076         #else
00077             struct timeval cstart;
00078         #endif
00079     }; // end class Timer
00080
00081 } // end namespace
00082
00083 #endif
```

5.26 vector.hh

```

00001 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
00002 /*
00003  * File:    vector.hh
00004  * Author:  ngo
00005  *
00006  * Created on April 14th, 2011
00007  */
00008
00009 #ifndef _VECTOR_HH
00010 #define _VECTOR_HH
00011
00012 #include <assert.h>
00013
00014 #include <cmath>
00015 #include <cstdlib>
00016 #include <fstream>
00017 #include <iomanip>
00018 #include <iostream>
00019 #include <sstream>
00020 #include <vector>
00021
00022 #include "exceptions.hh"
00023
00024 namespace hdnum {
00025
00026     template<typename REAL>
00027     class Vector : public std::vector<REAL> // inherit from the STL vector
00028     {
00029     public:
00030         typedef std::size_t size_type;
00031
00032     private:
00033         static bool bScientific;
00034         static std::size_t nIndexWidth;
00035         static std::size_t nValueWidth;
00036         static std::size_t nValuePrecision;
00037
00038     public:
00039
00040         Vector() : std::vector<REAL>()
00041         {
00042         }
00043
00044         Vector( const size_t size, // user must specify the size
00045                const REAL defaultvalue_ = 0 // if not specified, the value 0 will take effect
00046                )
00047             : std::vector<REAL>( size, defaultvalue_ )
00048         {
00049         }
00050
00051         Vector( const std::initializer_list<REAL> &v )
00052         {
00053             for (auto elem : v) this->push_back(elem);
00054         }
00055
00056         // Methods:
00057
00058         Vector& operator=( const REAL value )
00059         {
00060             const size_t s = this->size();
00061             Vector &self = *this;
00062             for(size_t i=0; i<s; ++i)
00063                 self[i] = value;
00064             return *this;
00065         }
00066
00067         Vector sub (size_type i, size_type m)
00068         {
00069             Vector v(m);
00070             Vector &self = *this;
00071             size_type k=0;
00072             for (size_type j=i; j<i+m; j++){
00073                 v[k]=self[j];
00074                 k++;
00075             }
00076             return v;
00077         }
00078
00079 #ifdef DOXYGEN
00080         Vector& operator=( const Vector& y )
00081         {
00082             // It is already implemented in the STL vector class itself!
00083         }
00084 #endif
00085
00086     };
00087
00088 }
00089

```

```

00154 #endif
00155
00156
00157
00159 Vector& operator*=( const REAL value )
00160 {
00161     Vector &self = *this;
00162     for (size_t i = 0; i < this->size(); ++i)
00163         self[i] *= value;
00164     return *this;
00165 }
00166
00167
00169 Vector& operator/=( const REAL value )
00170 {
00171     Vector &self = *this;
00172     for (size_t i = 0; i < this->size(); ++i)
00173         self[i] /= value;
00174     return *this;
00175 }
00176
00177
00179 Vector& operator+=( const Vector & y )
00180 {
00181     assert( this->size() == y.size() );
00182     Vector &self = *this;
00183     for (size_t i = 0; i < this->size(); ++i)
00184         self[i] += y[i];
00185     return *this;
00186 }
00187
00188
00190 Vector& operator-=( const Vector & y )
00191 {
00192     assert( this->size() == y.size() );
00193     Vector &self = *this;
00194     for (size_t i = 0; i < this->size(); ++i)
00195         self[i] -= y[i];
00196     return *this;
00197 }
00198
00199
00201 Vector & update(const REAL alpha, const Vector & y)
00202 {
00203     assert( this->size() == y.size() );
00204     Vector &self = *this;
00205     for (size_t i = 0; i < this->size(); ++i)
00206         self[i] += alpha * y[i];
00207     return *this;
00208 }
00209
00210
00242 REAL operator*(Vector & x) const
00243 {
00244     assert( x.size() == this->size() ); // checks if the dimensions of the two vectors are equal
00245     REAL sum( 0 );
00246     const Vector & self = *this;
00247     for( size_t i = 0; i < this->size(); ++i )
00248         sum += self[i] * x[i];
00249     return sum;
00250 }
00251
00252
00253
00254
00287 Vector operator+(Vector & x) const
00288 {
00289     assert( x.size() == this->size() ); // checks if the dimensions of the two vectors are equal
00290     Vector sum( *this );
00291     sum += x;
00292     return sum;
00293 }
00294
00295
00296
00329 Vector operator-(Vector & x) const
00330 {
00331     assert( x.size() == this->size() ); // checks if the dimensions of the two vectors are equal
00332     Vector sum( *this );
00333     sum -= x;
00334     return sum;
00335 }
00336
00337
00338
00340 REAL two_norm_2() const
00341 {

```

```

00342     REAL sum( 0 );
00343     const Vector & self = *this;
00344     for (size_t i = 0; i < (size_t) this->size(); ++i)
00345         sum += self[i] * self[i];
00346     return sum;
00347 }
00348
00373 REAL two_norm() const
00374 {
00375     return sqrt(two_norm_2());
00376 }
00377
00379 bool scientific() const
00380 {
00381     return bScientific;
00382 }
00383
00411 void scientific(bool b) const
00412 {
00413     bScientific=b;
00414 }
00415
00417 std::size_t iwidth () const
00418 {
00419     return nIndexWidth;
00420 }
00421
00423 std::size_t width () const
00424 {
00425     return nValueWidth;
00426 }
00427
00429 std::size_t precision () const
00430 {
00431     return nValuePrecision;
00432 }
00433
00435 void iwidth (std::size_t i) const
00436 {
00437     nIndexWidth=i;
00438 }
00439
00441 void width (std::size_t i) const
00442 {
00443     nValueWidth=i;
00444 }
00445
00447 void precision (std::size_t i) const
00448 {
00449     nValuePrecision=i;
00450 }
00451
00452 };
00453
00454
00455
00456 template<typename REAL>
00457 bool Vector<REAL>::bScientific = true;
00458
00459 template<typename REAL>
00460 std::size_t Vector<REAL>::nIndexWidth = 2;
00461
00462 template<typename REAL>
00463 std::size_t Vector<REAL>::nValueWidth = 15;
00464
00465 template<typename REAL>
00466 std::size_t Vector<REAL>::nValuePrecision = 7;
00467
00468
00490 template <typename REAL>
00491 inline std::ostream & operator <<(std::ostream & os, const Vector<REAL> & x)
00492 {
00493     os << std::endl;
00494
00495     for (size_t r = 0; r < x.size(); ++r)
00496     {
00497         if ( x.scientific() )
00498         {
00499             os << "["
00500                << std::setw(x.iwidth())
00501                << r
00502                << "]"
00503                << std::scientific
00504                << std::showpoint
00505                << std::setw( x.width() )
00506                << std::setprecision( x.precision() )
00507                << x[r]

```

```

00508         << std::endl;
00509     }
00510     else
00511     {
00512         os << "["
00513            << std::setw(x.iwidth())
00514            << r
00515            << "]"
00516            << std::fixed
00517            << std::showpoint
00518            << std::setw( x.width() )
00519            << std::setprecision( x.precision() )
00520            << x[r]
00521            << std::endl;
00522     }
00523 }
00524 return os;
00525 }
00526
00527
00528
00551 template<typename REAL>
00552 inline void gnuplot(
00553     const std::string& fname,
00554     const Vector<REAL> x
00555 )
00556 {
00557     std::fstream f(fname.c_str(),std::ios::out);
00558     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
00559     {
00560         if( x.scientific() )
00561         {
00562             f << std::setw(x.width())
00563             << i
00564             << std::scientific
00565             << std::showpoint
00566             << std::setw( x.width() )
00567             << std::setprecision( x.precision() )
00568             << x[i]
00569             << std::endl;
00570         }
00571         else
00572         {
00573             f << std::setw(x.width())
00574             << i
00575             << std::fixed
00576             << std::showpoint
00577             << std::setw( x.width() )
00578             << std::setprecision( x.precision() )
00579             << x[i]
00580             << std::endl;
00581         }
00582     }
00583     f.close();
00584 }
00585
00586 template<typename REAL>
00587 inline void gnuplot(
00588     const std::string& fname,
00589     const std::vector<std::string>& t,
00590     const Vector<REAL> x
00591 )
00592 {
00593     std::fstream f(fname.c_str(),std::ios::out);
00594     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
00595     {
00596         if( x.scientific() )
00597         {
00598             f << t[i] << " "
00599             << std::scientific
00600             << std::showpoint
00601             << std::setw( x.width() )
00602             << std::setprecision( x.precision() )
00603             << x[i]
00604             << std::endl;
00605         }
00606         else
00607         {
00608             f << t[i] << " "
00609             << std::fixed
00610             << std::showpoint
00611             << std::setw( x.width() )
00612             << std::setprecision( x.precision() )
00613             << x[i]
00614             << std::endl;
00615         }
00616     }

```

```

00617     f.close();
00618 }
00619
00621 template<typename REAL>
00622 inline void gnuplot(
00623     const std::string& fname,
00624     const Vector<REAL> x,
00625     const Vector<REAL> y
00626 )
00627 {
00628     std::fstream f(fname.c_str(),std::ios::out);
00629     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
00630     {
00631         if( x.scientific() )
00632         {
00633             f << std::setw(x.width())
00634               << i
00635               << std::scientific
00636               << std::showpoint
00637               << std::setw( x.width() )
00638               << std::setprecision( x.precision() )
00639               << x[i]
00640               << " "
00641               << std::setw( x.width() )
00642               << std::setprecision( x.precision() )
00643               << y[i]
00644               << std::endl;
00645         }
00646         else
00647         {
00648             f << std::setw(x.width())
00649               << i
00650               << std::fixed
00651               << std::showpoint
00652               << std::setw( x.width() )
00653               << std::setprecision( x.precision() )
00654               << x[i]
00655               << " "
00656               << std::setw( x.width() )
00657               << std::setprecision( x.precision() )
00658               << y[i]
00659               << std::endl;
00660         }
00661     }
00662
00663     f.close();
00664 }
00665
00666
00667
00696 template<typename REAL>
00697 inline void readVectorFromFile (const std::string& filename, Vector<REAL> &vector)
00698 {
00699     std::string buffer;
00700     std::ifstream fin( filename.c_str() );
00701     if( fin.is_open() ){
00702         while( fin ){
00703             std::string sub;
00704             fin >> sub;
00705             //std::cout << " sub = " << sub.c_str() << ": ";
00706             if( sub.length()>0 ){
00707                 REAL a = atof(sub.c_str());
00708                 //std::cout << std::fixed << std::setw(10) << std::setprecision(5) << a;
00709                 vector.push_back(a);
00710             }
00711         }
00712         fin.close();
00713     }
00714     else{
00715         HDNUM_ERROR("Could not open file!");
00716     }
00717 }
00718
00719
00721 template<class REAL>
00722 inline void zero (Vector<REAL>& x)
00723 {
00724     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
00725         x[i] = REAL(0);
00726 }
00727
00729 template<class REAL>
00730 inline REAL norm (Vector<REAL> x)
00731 {
00732     REAL sum(0.0);
00733     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
00734         sum += x[i]*x[i];

```

```
00735     return sqrt(sum);
00736 }
00737
00739 template<class REAL>
00740 inline void fill (Vector<REAL>& x, const REAL t)
00741 {
00742     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
00743         x[i] = t;
00744 }
00745
00768 template<class REAL>
00769 inline void fill (Vector<REAL>& x, const REAL& t, const REAL& dt)
00770 {
00771     REAL myt(t);
00772     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
00773     {
00774         x[i] = myt;
00775         myt += dt;
00776     }
00777 }
00778
00779
00802 template<class REAL>
00803 inline void unitvector (Vector<REAL> & x, std::size_t j)
00804 {
00805     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
00806         if (i==j)
00807             x[i] = REAL(1);
00808         else
00809             x[i] = REAL(0);
00810 }
00811
00812
00813 } // end of namespace hdnun
00814
00815 #endif /* _VECTOR_HH */
```


Index

begin
 hdnum::SparseMatrix< REAL >, 60

cbegin
 hdnum::SparseMatrix< REAL >, 60

cend
 hdnum::SparseMatrix< REAL >, 60

colsize
 hdnum::DenseMatrix< REAL >, 14
 hdnum::SparseMatrix< REAL >, 60

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

end
 hdnum::SparseMatrix< REAL >, 61

exceptions.hh
 HDNUM_ERROR, 84
 HDNUM_THROW, 84

fill
 hdnum::Vector< REAL >, 74

getNeighborIndex
 hdnum::SGrid< N, DF, dimension >, 56

getNonlinearProblem
 newton.hh, 90

gnuplot
 hdnum::Vector< REAL >, 74

gram_schmidt
 qr.hh, 129

hdnum::Banach, 7

hdnum::DenseMatrix< REAL >, 11
 colsize, 14
 identity, 28
 mm, 14
 mv, 15
 operator(), 16
 operator+, 18
 operator+=", 19
 operator-, 19
 operator-=, 20
 operator/=", 20
 operator=, 22
 operator*, 16, 17
 operator*=", 18
 readMatrixFromFileDat, 29
 readMatrixFromFileMatrixMarket, 29
 rowsize, 22
 sc, 23
 scientific, 23
 spd, 30
 sr, 24
 sub, 24
 transpose, 25
 umm, 25
 umv, 26, 27
 update, 28
 vandermonde, 30
 hdnum::DIRK< M, S >, 31
 DIRK, 33
 hdnum::EE< M >, 33
 hdnum::ErrorException, 34
 hdnum::Exception, 35
 hdnum::GenericNonlinearProblem< Lambda, Vec >, 36
 hdnum::Heun2< M >, 36
 hdnum::Heun3< M >, 37
 hdnum::IE< M, S >, 39
 hdnum::ImplicitRungeKuttaStepProblem< M >, 40
 hdnum::InvalidStateException, 41
 hdnum::IOError, 41
 hdnum::Kutta3< M >, 42
 hdnum::MathError, 43
 hdnum::ModifiedEuler< M >, 44
 hdnum::Newton, 45
 hdnum::NotImplemented, 46
 hdnum::oc::OpCounter< F >, 47
 hdnum::oc::OpCounter< F >::Counters, 10
 hdnum::OutOfMemoryError, 48
 hdnum::RangeError, 49
 hdnum::RE< M, S >, 50
 hdnum::RK45< M >, 51
 hdnum::RungeKutta< M, S >, 53
 hdnum::RungeKutta4< M >, 54
 hdnum::SGrid< N, DF, dimension >, 55
 getNeighborIndex, 56
 SGrid, 56
 hdnum::SparseMatrix< REAL >, 57
 begin, 60
 cbegin, 60
 cend, 60
 colsize, 60
 end, 61
 identity, 61, 64
 matchingIdentity, 61
 mv, 62
 norm_infty, 62
 operator/=", 63
 operator*, 62

- operator*=[, 63](#)
- rowsize, [63](#)
- scientific, [64](#)
- SparseMatrix, [59](#)
- umv, [64](#)
- 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, [52](#)
- hdnum::SquareRootProblem< N >, [65](#)
- hdnum::StationarySolver< M >, [66](#)
- hdnum::SystemError, [67](#)
- hdnum::Timer, [67](#)
- hdnum::TimerError, [68](#)
- hdnum::Vector< REAL >, [69](#)
 - fill, [74](#)
 - gnuplot, [74](#)
 - operator<<, [75](#)
 - operator+, [71](#)
 - operator-, [72](#)
 - operator=, [72](#)
 - operator*, [71](#)
 - readVectorFromFile, [75](#)
 - scientific, [73](#)
 - sub, [73](#)
 - two_norm, [74](#)
 - unitvector, [76](#)
- HDNUM_ERROR
 - exceptions.hh, [84](#)
- HDNUM_THROW
 - exceptions.hh, [84](#)
- identity
 - hdnum::DenseMatrix< REAL >, [28](#)
 - hdnum::SparseMatrix< REAL >, [61, 64](#)
- matchingIdentity
 - hdnum::SparseMatrix< REAL >, [61](#)
- mm
 - hdnum::DenseMatrix< REAL >, [14](#)
- modified_gram_schmidt
 - qr.hh, [129](#)
- mv
 - hdnum::DenseMatrix< REAL >, [15](#)
 - hdnum::SparseMatrix< REAL >, [62](#)
- newton.hh
 - getNonlinearProblem, [90](#)
- norm_infty
 - hdnum::SparseMatrix< REAL >, [62](#)
- operator<<
 - hdnum::Vector< REAL >, [75](#)
- operator()
 - hdnum::DenseMatrix< REAL >, [16](#)
- operator+
 - hdnum::DenseMatrix< REAL >, [18](#)
 - hdnum::Vector< REAL >, [71](#)
- operator+=
 - hdnum::DenseMatrix< REAL >, [19](#)
- operator-
 - hdnum::DenseMatrix< REAL >, [19](#)
 - hdnum::Vector< REAL >, [72](#)
- operator-=
 - hdnum::DenseMatrix< REAL >, [20](#)
- operator/=
 - hdnum::DenseMatrix< REAL >, [20](#)
 - hdnum::SparseMatrix< REAL >, [63](#)
- operator=
 - hdnum::DenseMatrix< REAL >, [22](#)
 - hdnum::Vector< REAL >, [72](#)
- operator*
 - hdnum::DenseMatrix< REAL >, [16, 17](#)
 - hdnum::SparseMatrix< REAL >, [62](#)
 - hdnum::Vector< REAL >, [71](#)
- operator*=
 - hdnum::DenseMatrix< REAL >, [18](#)
 - hdnum::SparseMatrix< REAL >, [63](#)
- ordertest
 - rungekutta.hh, [142](#)
- permute_forward
 - qr.hh, [130](#)
- qr.hh
 - gram_schmidt, [129](#)
 - modified_gram_schmidt, [129](#)
 - permute_forward, [130](#)
 - qr_gram_schmidt, [130](#)
 - qr_gram_schmidt_pivoting, [131](#)
 - qr_gram_schmidt_simple, [132](#)
- qr_gram_schmidt
 - qr.hh, [130](#)
- qr_gram_schmidt_pivoting
 - qr.hh, [131](#)
- qr_gram_schmidt_simple
 - qr.hh, [132](#)
- qrhousholder
 - qrhousholder.hh, [138](#)
- qrhousholder.hh
 - qrhousholder, [138](#)
 - qrhousholderexplicitQ, [139](#)
- qrhousholderexplicitQ
 - qrhousholder.hh, [139](#)
- readMatrixFromFileDat
 - hdnum::DenseMatrix< REAL >, [29](#)
- readMatrixFromFileMatrixMarket
 - hdnum::DenseMatrix< REAL >, [29](#)
- readVectorFromFile
 - hdnum::Vector< REAL >, [75](#)
- rowsize
 - hdnum::DenseMatrix< REAL >, [22](#)
 - hdnum::SparseMatrix< REAL >, [63](#)
- rungekutta.hh

- ordertest, [142](#)
- sc
 - hdnum::DenseMatrix< REAL >, [23](#)
- scientific
 - hdnum::DenseMatrix< REAL >, [23](#)
 - hdnum::SparseMatrix< REAL >, [64](#)
 - hdnum::Vector< REAL >, [73](#)
- SGrid
 - hdnum::SGrid< N, DF, dimension >, [56](#)
- SparseMatrix
 - hdnum::SparseMatrix< REAL >, [59](#)
- spd
 - hdnum::DenseMatrix< REAL >, [30](#)
- sr
 - hdnum::DenseMatrix< REAL >, [24](#)
- src/densematrix.hh, [77](#)
- src/exceptions.hh, [83](#), [85](#)
- src/lr.hh, [86](#), [87](#)
- src/newton.hh, [90](#), [91](#)
- src/ode.hh, [95](#), [96](#)
- src/opcounter.hh, [112](#), [115](#)
- src/pde.hh, [125](#), [126](#)
- src/precision.hh, [127](#)
- src/qr.hh, [128](#), [133](#)
- src/qrhousholder.hh, [137](#), [139](#)
- src/rungekutta.hh, [141](#), [142](#)
- src/sgrid.hh, [148](#)
- src/sparsematrix.hh, [150](#)
- src/timer.hh, [160](#), [161](#)
- src/vector.hh, [162](#)
- sub
 - hdnum::DenseMatrix< REAL >, [24](#)
 - hdnum::Vector< REAL >, [73](#)
- transpose
 - hdnum::DenseMatrix< REAL >, [25](#)
- two_norm
 - hdnum::Vector< REAL >, [74](#)
- umm
 - hdnum::DenseMatrix< REAL >, [25](#)
- umv
 - hdnum::DenseMatrix< REAL >, [26](#), [27](#)
 - hdnum::SparseMatrix< REAL >, [64](#)
- unitvector
 - hdnum::Vector< REAL >, [76](#)
- update
 - hdnum::DenseMatrix< REAL >, [28](#)
- vandermonde
 - hdnum::DenseMatrix< REAL >, [30](#)