Heidelberg Educational Numerics Library

Generated by Doxygen 1.9.3

Chapter 1

Hierarchical Index

1.1 Class Hierarchy

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

Chapter 2

Class Index

2.1 Class List

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

Chapter 3

File Index

3.1 File List

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

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 valIter, 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 valIter, std::vector< size_type >::const_iterator colIndicesIter)
- self_type & **operator++** ()
- self_type **operator++** (int junk)
- value_type **operator*** ()
- value_type::first_type **value** ()
- value_type::second_type **index** ()
- bool **operator==** (const self_type &other)
- bool **operator!=** (const self_type &other)

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

- src/sparsematrix.hh

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

Public Types

- using **self_type** = const_row_iterator
- using **difference_type** = std::ptrdiff_t
- using **value_type** = self_type
- using **pointer** = self_type *
- using **reference** = self_type &
- using **iterator_category** = std::bidirectional_iterator_tag

Public Member Functions

- **const_row_iterator** (std::vector< size_type >::const_iterator rowPtrIter, std::vector< size_type >::const_iterator colIndicesIter, typename std::vector< REAL >::const_iterator valIter)
- const_column_iterator **begin** () const
- const_column_iterator **end** () const
- const_column_index_iterator **ibegin** () const
- const_column_index_iterator **iend** () const
- const_column_iterator **cbegin** () const
- const_column_iterator **cend** () const
- self_type & **operator++** ()
- self_type & **operator++** (int junk)
- self_type & **operator+=** (difference_type offset)
- self_type & **operator-=** (difference_type offset)
- self_type **operator-** (difference_type offset)
- self_type **operator+** (difference_type offset)
- reference **operator[ ]** (difference_type offset)
- bool **operator**< (const self_type &other)
- bool **operator**> (const self_type &other)
- self_type & **operator**∗ ()
- bool **operator==** (const self_type &rhs)
- bool **operator!=** (const self_type &rhs)

Friends

- self_type **operator+** (const difference_type &offset, const self_type &sec)

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

- src/sparsematrix.hh

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

Struct storing the number of operations.

`#include <opcounter.hh>`

Public Member Functions

- void **reset** ()
- template<typename Stream >
  void **reportOperations** (Stream &os, bool doReset=false)

  Report operations to stream object.
- size_type **totalOperationCount** (bool doReset=false)

  Get total number of operations.
- Counters & **operator+=** (const Counters &rhs)
- Counters **operator-** (const Counters &rhs)

Public Attributes

- size_type **addition_count**
- size_type **multiplication_count**
- size_type **division_count**
- size_type **exp_count**
- size_type **pow_count**
- size_type **sin_count**
- size_type **sqrt_count**
- size_type **comparison_count**

4.6.1 Detailed Description

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

Struct storing the number of operations.

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

- src/opcounter.hh

4.7 hdnum::DenseMatrix< REAL > Class Template Reference

Class with mathematical matrix operations.

`#include <densematrix.hh>`

Public Types

- typedef std::size_t **size_type**
  Type used for array indices.
- typedef std::vector< REAL > **VType**
- typedef VType::const_iterator **ConstVectorIterator**
- typedef VType::iterator **VectorIterator**

Public Member Functions

- **DenseMatrix** ()

  default constructor (empty Matrix)
- **DenseMatrix** (const std::size_t _rows, const std::size_t _cols, const REAL def_val=0)

  constructor
- **DenseMatrix** (const std::initializer_list< std::initializer_list< REAL > > &v)

  constructor from initializer list
- **DenseMatrix** (const hdnum::SparseMatrix< REAL > &other)

  constructor from hdnum::SparseMatrix
- void **addNewRow** (const hdnum::Vector< REAL > &rowvector)
- size_t rowsize () const

  get number of rows of the matrix
- size_t colsize () const

  get number of columns of the matrix
- bool **scientific** () const
- void scientific (bool b) const

  Switch between floating point (default=true) and fixed point (false) display.
- std::size_t **iwidth** () const

  get index field width for pretty-printing
- std::size_t **width** () const

  get data field width for pretty-printing
- std::size_t **precision** () const

  get data precision for pretty-printing
- void **iwidth** (std::size_t i) const

  set index field width for pretty-printing
- void **width** (std::size_t i) const

  set data field width for pretty-printing
- void **precision** (std::size_t i) const

  set data precision for pretty-printing
- REAL & operator() (const std::size_t row, const std::size_t col)

  (i,j)-operator for accessing entries of a (m x n)-matrix directly
- const REAL & **operator()** (const std::size_t row, const std::size_t col) const

  read-access on matrix element A_ij using A(i,j)
- const ConstVectorIterator **operator[ ]** (const std::size_t row) const

  read-access on matrix element A_ij using A[i][j]
- VectorIterator **operator[ ]** (const std::size_t row)

  write-access on matrix element A_ij using A[i][j]
- DenseMatrix & operator= (const DenseMatrix &A)

  assignment operator
- DenseMatrix & operator= (const REAL value)

  assignment from a scalar value
- DenseMatrix sub (size_type i, size_type j, size_type rows, size_type cols)

  Submatrix extraction.
- DenseMatrix transpose () const

  Transposition.
- DenseMatrix & operator+= (const DenseMatrix &B)

  Addition assignment.
- DenseMatrix & operator-= (const DenseMatrix &B)

  Subtraction assignment.
- DenseMatrix & operator∗= (const REAL s)

Scalar multiplication assignment.

- DenseMatrix & operator/= (const REAL s)

  Scalar division assignment.
- void update (const REAL s, const DenseMatrix &B)

  Scaled update of a Matrix.
- template<class V >
  void mv (Vector< V > &y, const Vector< V > &x) const

  matrix vector product y = A∗x
- template<class V >
  void umv (Vector< V > &y, const Vector< V > &x) const

  update matrix vector product y += A∗x
- template<class V >
  void umv (Vector< V > &y, const V &s, const Vector< V > &x) const

  update matrix vector product y += sA∗x
- void mm (const DenseMatrix< REAL > &A, const DenseMatrix< REAL > &B)

  assign to matrix product C = A∗B to matrix C
- void umm (const DenseMatrix< REAL > &A, const DenseMatrix< REAL > &B)

  add matrix product A∗B to matrix C
- void sc (const Vector< REAL > &x, std::size_t k)

  set column: make x the k'th column of A
- void sr (const Vector< REAL > &x, std::size_t k)

  set row: make x the k'th row of A
- REAL **norm_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 Functions

(Note that these are not member functions.)

- template<class T >
  void identity (DenseMatrix< T > &A)
- template<typename REAL >
  void spd (DenseMatrix< REAL > &A)
- template<typename REAL >
  void vandermonde (DenseMatrix< REAL > &A, const Vector< REAL > x)
- template<typename REAL >
  void readMatrixFromFileDat (const std::string &filename, DenseMatrix< REAL > &A)

  Read matrix from a text file.
- template<typename REAL >
  void readMatrixFromFileMatrixMarket (const std::string &filename, DenseMatrix< REAL > &A)

  Read matrix from a matrix market file.

4.7.1 Detailed Description

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

Class with mathematical matrix operations.

4.7.2 Member Function Documentation

4.7.2.1 colsize()

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

get number of columns of the matrix

Example:

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

Output:

```
Matrix A has 5 columns.
```

4.7.2.2 mm()

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

assign to matrix product C = A∗B to matrix C

Implements C = A∗B where A and B are matrices

Parameters

in	*A*	constant reference to a DenseMatrix
in	*B*	constant reference to a DenseMatrix

Example:

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

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

Output:

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

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

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

4.7.2.3 mv()

```
template<typename REAL >
template<class V >
void hdnum::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:

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

Output:

```
A =
0           1           2
```

```
0       1.000       1.000       1.000
1       1.000       1.000       1.000

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

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

4.7.2.4 operator()()

```
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	*row*	row index (0...m-1)
in	*col*	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 > hdnum::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 & hdnum::DenseMatrix< REAL >::operator+= (
            const DenseMatrix< REAL > & B )  [inline]
```

Addition assignment.

Implements A += B matrix addition

Parameters

in	*B*	another Matrix

4.7.2.10 operator-()

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

matrix = matrix - matrix

Parameters

in	*x*	constant reference to a 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	*B*	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*	first row index of the new matrix
in	*j*	first column index of the new matrix
in	*rows*	row size of the new matrix, i.e. it has components [i,i+rows-1]
in	*cols*	column size of the new matrix, i.e. it has components [j,j+cols-1]

4.7.2.20 transpose()

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

Transposition.

Return the transposed as a new matrix.

4.7.2.21 umm()

```
template<typename REAL >
void hdnum::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	*A*	constant reference to a DenseMatrix
in	*B*	constant reference to a DenseMatrix

Example:

```
hdnum::DenseMatrix<double> A(2,6,1.0);
hdnum::DenseMatrix<double> B(6,3,-1.0);
hdnum::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 = s*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 = s*A*x =
[ 0]      20.0000000
[ 1]      20.0000000
```

4.7.2.23 umv() [2/2]

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

update matrix vector product y += A∗x

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

Parameters

in	*y*	reference to the resulting Vector
in	*x*	constant reference to a Vector

Example:

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

Output:

```
y =
[ 0]       5.0000000
[ 1]       5.0000000

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

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

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

4.7.2.24 update()

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

Scaled update of a Matrix.

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

Parameters

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

Example:

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

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

Output:

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

4.7.3 Friends And Related Function Documentation

4.7.3.1 identity()

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

Function: make identity matrix

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

Parameters

in	*A*	reference to a 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	*filename*	name of the text file
in,out	*A*	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	*filename*	name of the text file
in,out	*A*	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	*A*	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:

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

Output:

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

A=
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 hdnum::DIRK< M, S > Class Template Reference

Implementation of a general Diagonal Implicit Runge-Kutta method.

`#include <ode.hh>`

Public Types

- typedef M::size_type **size_type**

  export size_type
- typedef M::time_type **time_type**

  export time_type
- typedef M::number_type **number_type**

  export number_type
- typedef DenseMatrix< number_type > **ButcherTableau**

  the type of a Butcher tableau

Public Member Functions

- DIRK (const M &model_, const S &newton_, const ButcherTableau &butcher_, const int order_)
- DIRK (const M &model_, const S &newton_, const std::string method)
- void **set_dt** (time_type dt_)

  set time step for subsequent steps
- void **set_verbosity** (size_type verbosity_)

  set verbosity level
- void **step** ()

  do one step
- bool **get_error** () const

  get current state
- void **set_state** (time_type t_, const Vector< number_type > &u_)

  set current state
- const Vector< number_type > & **get_state** () const

  get current state
- time_type **get_time** () const

  get current time
- time_type **get_dt** () const

  get dt used in last step (i.e. to compute current state)
- size_type **get_order** () const

  return consistency order of the method
- void **get_info** () const

  print some information

4.8.1 Detailed Description

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

Implementation of a general Diagonal Implicit Runge-Kutta method.

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

Template Parameters

M	the model type
S	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

M	the model type

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

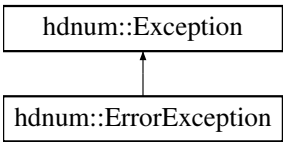
- src/ode.hh

4.10 hdnum::ErrorException Class Reference

General Error.

`#include <exceptions.hh>`

Inheritance diagram for hdnum::ErrorException:



Additional Inherited Members

4.10.1 Detailed Description

General Error.

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

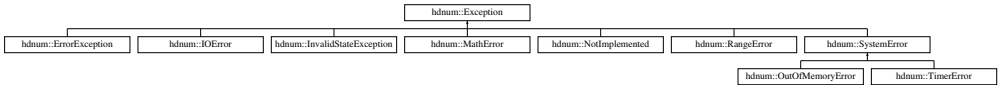
- src/exceptions.hh

4.11 hdnum::Exception Class Reference

Base class for Exceptions.

`#include <exceptions.hh>`

Inheritance diagram for hdnum::Exception:



Public Member Functions

- void **message** (const std::string &message)

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

  output internal message buffer

4.11.1 Detailed Description

Base class for Exceptions.

all HDNUM exceptions are derived from this class via trivial subclassing:
```
class MyException : public Dune::Exception {};
```

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

See also

HDNUM_THROW, IOError, MathError

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

- src/exceptions.hh

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

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

`#include <newton.hh>`

Public Types

- typedef std::size_t **size_type**

  export size_type
- typedef Vec::value_type **number_type**

  export number_type

Public Member Functions

- **GenericNonlinearProblem** (const Lambda &l_, const Vec &x_, number_type eps_=1e-7)

  constructor stores parameter lambda
- std::size_t **size** () const

  return number of componentes for the model
- void **F** (const Vec &x, Vec &result) const

  model evaluation
- void **F_x** (const Vec &x, DenseMatrix< number_type > &result) const

  jacobian evaluation needed for implicit solvers

4.12.1 Detailed Description

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

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

Template Parameters

Lambda	mapping a Vector to a Vector
Vec	the type for the Vector

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

- src/newton.hh

4.13 hdnum::Heun2< M > Class Template Reference

Heun method (order 2 with 2 stages)

`#include <ode.hh>`

Public Types

- typedef M::size_type **size_type**

  export size_type
- typedef M::time_type **time_type**

  export time_type
- typedef M::number_type **number_type**

  export number_type

Public Member Functions

- **Heun2** (const M &model_)
  constructor stores reference to the model
- void **set_dt** (time_type dt_)
  set time step for subsequent steps
- void **step** ()
  do one step
- void **set_state** (time_type t_, const Vector< number_type > &u_)
  set current state
- const Vector< number_type > & **get_state** () const
  get current state
- time_type **get_time** () const
  get current time
- time_type **get_dt** () const
  get dt used in last step (i.e. to compute current state)
- size_type **get_order** () const
  return consistency order of the method

4.13.1 Detailed Description

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

Heun method (order 2 with 2 stages)

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

Template Parameters

M	the model type

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

- src/ode.hh

4.14 hdnum::Heun3< M > Class Template Reference

Heun method (order 3 with 3 stages)

`#include <ode.hh>`

Public Types

- typedef M::size_type **size_type**
  export size_type
- typedef M::time_type **time_type**
  export time_type
- typedef M::number_type **number_type**
  export number_type

Public Member Functions

- **Heun3** (const M &model_)
  constructor stores reference to the model
- void **set_dt** (time_type dt_)
  set time step for subsequent steps
- void **step** ()
  do one step
- void **set_state** (time_type t_, const Vector< number_type > &u_)
  set current state
- const Vector< number_type > & **get_state** () const
  get current state
- time_type **get_time** () const
  get current time
- time_type **get_dt** () const
  get dt used in last step (i.e. to compute current state)
- size_type **get_order** () const
  return consistency order of the method

4.14.1 Detailed Description

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

Heun method (order 3 with 3 stages)

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

Template Parameters

M	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

M	the model type
S	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 implicite solvers

4.16.1 Detailed Description

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

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

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

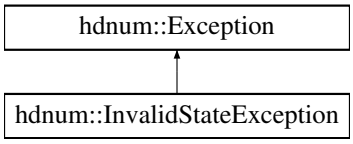
- src/rungekutta.hh

4.17 hdnum::InvalidStateException Class Reference

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

`#include <exceptions.hh>`

Inheritance diagram for hdnum::InvalidStateException:



Additional Inherited Members

4.17.1 Detailed Description

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

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

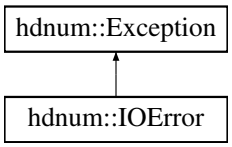
- src/exceptions.hh

4.18 hdnum::IOError Class Reference

Default exception class for I/O errors.

`#include <exceptions.hh>`

Inheritance diagram for hdnum::IOError:



Additional Inherited Members

4.18.1 Detailed Description

Default exception class for I/O errors.

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

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

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

- src/exceptions.hh

4.19 hdnum::Kutta3< M > Class Template Reference

Kutta method (order 3 with 3 stages)

`#include <ode.hh>`

Public Types

- typedef M::size_type **size_type**

  export size_type
- typedef M::time_type **time_type**

  export time_type
- typedef M::number_type **number_type**

  export number_type

Public Member Functions

- **Kutta3** (const M &model_)

  constructor stores reference to the model
- void **set_dt** (time_type dt_)

  set time step for subsequent steps
- void **step** ()

  do one step
- void **set_state** (time_type t_, const Vector< number_type > &u_)

  set current state
- const Vector< number_type > & **get_state** () const

  get current state
- time_type **get_time** () const

  get current time
- time_type **get_dt** () const

  get dt used in last step (i.e. to compute current state)
- size_type **get_order** () const

  return consistency order of the method

4.19.1 Detailed Description

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

Kutta method (order 3 with 3 stages)

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

Template Parameters

M	the model type

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

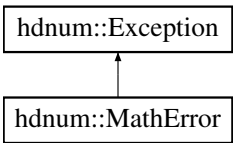
- src/ode.hh

4.20 hdnum::MathError Class Reference

Default exception class for mathematical errors.

`#include <exceptions.hh>`

Inheritance diagram for hdnum::MathError:



Additional Inherited Members

4.20.1 Detailed Description

Default exception class for mathematical errors.

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

- matrix not invertible
- not convergent

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

- src/exceptions.hh

4.21 hdnum::ModifiedEuler< M > Class Template Reference

Modified Euler method (order 2 with 2 stages)

`#include <ode.hh>`

Public Types

- typedef M::size_type **size_type**
  export size_type
- typedef M::time_type **time_type**
  export time_type
- typedef M::number_type **number_type**
  export number_type

Public Member Functions

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

4.21.1 Detailed Description

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

Modified Euler method (order 2 with 2 stages)

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

Template Parameters

M	the model type

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

- src/ode.hh

4.22 hdnum::Newton Class Reference

Solve nonlinear problem using a damped Newton method.

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

Public Member Functions

- **Newton** ()

  constructor stores reference to the model
- void **set_maxit** (size_type n)

  maximum number of iterations before giving up
- void **set_sigma** (double sigma_)
- void **set_linesearchsteps** (size_type n)

  maximum number of steps in linesearch before giving up
- void **set_verbosity** (size_type n)

  control output given 0=nothing, 1=summary, 2=every step, 3=include line search
- void **set_abslimit** (double l)

  basolute limit for defect
- void **set_reduction** (double l)

  reduction factor
- template<class M >
  void **solve** (const M &model, Vector< typename M::number_type > &x) const

  do one step
- bool **has_converged** () const
- size_type **iterations** () const

4.22.1 Detailed Description

Solve nonlinear problem using a damped Newton method.

The Newton solver is parametrized by a model. The model also exports all relevant types for types.

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

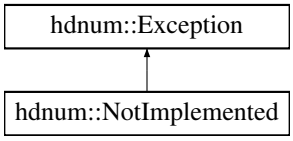
- src/newton.hh

4.23 hdnum::NotImplemented Class Reference

Default exception for dummy implementations.

`#include <exceptions.hh>`

Inheritance diagram for hdnum::NotImplemented:



Additional Inherited Members

4.23.1 Detailed Description

Default exception for dummy implementations.

This exception can be used for functions/methods

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

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

- src/exceptions.hh

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

`#include <opcounter.hh>`

Classes

- struct Counters

  Struct storing the number of operations.

Public Types

- using **size_type** = std::size_t
- using **value_type** = F

Public Member Functions

- template<typename T >
  OpCounter (const T &t, typename std::enable_if< std::is_same< T, int >::value and !std::is_same< F, int >::value >::type *=nullptr)
- **OpCounter** (const F &f)
- **OpCounter** (F &&f)
- **OpCounter** (const char *s)
- OpCounter & **operator=** (const char *s)
- **operator F** () const
- OpCounter & **operator=** (const F &f)
- OpCounter & **operator=** (F &&f)
- F * **data** ()
- const F * **data** () const

Static Public Member Functions

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

Public Attributes

- F **_v**

Static Public Attributes

- static Counters **counters**

Friends

- std::ostream & **operator**<< (std::ostream &os, const OpCounter &f)
- std::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:

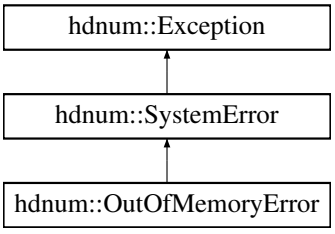
- src/opcounter.hh

4.25 hdnum::OutOfMemoryError Class Reference

Default exception if memory allocation fails.

`#include <exceptions.hh>`

Inheritance diagram for hdnum::OutOfMemoryError:



Additional Inherited Members

4.25.1 Detailed Description

Default exception if memory allocation fails.

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

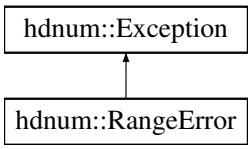
- src/exceptions.hh

4.26 hdnum::RangeError Class Reference

Default exception class for range errors.

`#include <exceptions.hh>`

Inheritance diagram for hdnum::RangeError:



Additional Inherited Members

4.26.1 Detailed Description

Default exception class for range errors.

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

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

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

- src/exceptions.hh

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

Adaptive one-step method using Richardson extrapolation.

`#include <ode.hh>`

Public Types

- typedef M::size_type **size_type**
  export size_type
- typedef M::time_type **time_type**
  export time_type
- typedef M::number_type **number_type**
  export number_type

Public Member Functions

- **RE** (const M &model_, S &solver_)
  constructor stores reference to the model
- void **set_dt** (time_type dt_)
  set time step for subsequent steps
- void **set_TOL** (time_type TOL_)
  set tolerance for adaptive computation
- void **step** ()
  do one step
- const Vector< number_type > & **get_state** () const
  get current state
- time_type **get_time** () const
  get current time
- time_type **get_dt** () const
  get dt used in last step (i.e. to compute current state)
- size_type **get_order** () const
  return consistency order of the method
- void **get_info** () const
  print some information

4.27.1 Detailed Description

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

Adaptive one-step method using Richardson extrapolation.

Template Parameters

M	a model
S	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

M	the model type

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

- src/ode.hh

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

Public Types

- using **self_type** = row_iterator
- using **difference_type** = std::ptrdiff_t
- using **value_type** = self_type
- using **pointer** = self_type *
- using **reference** = self_type &
- using **iterator_category** = std::random_access_iterator_tag

Public Member Functions

- **row_iterator** (std::vector< size_type >::iterator rowPtrIter, std::vector< size_type >::iterator colIndicesIter, typename std::vector< REAL >::iterator valIter)
- column_iterator **begin** ()
- column_iterator **end** ()
- column_index_iterator **ibegin** ()
- column_index_iterator **iend** ()
- self_type & **operator++** ()
- self_type **operator++** (int junk)
- self_type & **operator+=** (difference_type offset)
- self_type & **operator-=** (difference_type offset)
- self_type **operator-** (difference_type offset)
- self_type **operator+** (difference_type offset)
- reference **operator[ ]** (difference_type offset)
- bool **operator<** (const self_type &other)
- bool **operator>** (const self_type &other)
- self_type & **operator*** ()
- bool **operator==** (const self_type &rhs)
- bool **operator!=** (const self_type &rhs)

Friends

- self_type **operator+** (const difference_type &offset, const self_type &sec)

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

- src/sparsematrix.hh

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

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

`#include <rungekutta.hh>`

Public Types

- typedef M::size_type **size_type**

  export size_type
- typedef M::time_type **time_type**

  export time_type
- typedef M::number_type **number_type**

  export number_type

Public Member Functions

- **RungeKutta** (const M &model_, DenseMatrix< number_type > A_, Vector< number_type > b_, Vector< number_type > c_)

  constructor stores reference to the model
- **RungeKutta** (const M &model_, DenseMatrix< number_type > A_, Vector< number_type > b_, Vector< number_type > c_, number_type sigma_)

  constructor stores reference to the model
- void **set_dt** (time_type dt_)

  set time step for subsequent steps
- bool **check_explicit** ()

  test if method is explicit
- void **step** ()

  do one step
- void **set_state** (time_type t_, const Vector< number_type > &u_)

  set current state
- const Vector< number_type > & **get_state** () const

  get current state
- time_type **get_time** () const

  get current time
- time_type **get_dt** () const

  get dt used in last step (i.e. to compute current state)
- void **set_verbosity** (int verbosity_)

  how much should the ODE solver talk

4.30.1 Detailed Description

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

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

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

Template Parameters

M	The model type
S	(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

M	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 In, 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 In.
- bool **isBoundaryNode** (const size_type In) const

  Returns true if the node is on the boundary of the discrete compuational domain.
- size_type **getNumberOfNodes** () const

  Returns the number of nodes which are in the compuational 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 In) 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

N	A continuous type representing coordinate values.
DF	A boolean function which defines the domain.
dimension	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	*extent_*	The extent of the grid domain. The actual computational domain may be smaller and is defined by the domain function df_.
in	*size_*	The number of nodes in each grid dimension.
in	*df_*	The domain function. It has to provide a boolean function evaluate(Vector<number_type> x) which returns true if the node which is positioned at the coordinates of x is within the computational domain.

4.32.3 Member Function Documentation

4.32.3.1 getNeighborIndex()

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

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

Parameters

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

Returns

size_type The index of the neighbor node.

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

- src/sgrid.hh

4.33 hdnum::SparseMatrix< REAL > Class Template Reference

Sparse matrix Class with mathematical matrix operations.

`#include <sparsematrix.hh>`

Classes

- class builder
- class column_index_iterator
- class const_column_index_iterator
- class const_row_iterator
- class row_iterator

Public Types

- using **size_type** = std::size_t

  Types used for array indices.
- using **column_iterator** = typename std::vector< REAL >::iterator

  type of a regular column iterator (no access to indices)
- using **const_column_iterator** = typename std::vector< REAL >::const_iterator

  type of a const regular column iterator (no access to indices)

Public Member Functions

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

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

Static Public Member Functions

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

Related Functions

(Note that these are not member functions.)

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

4.33.1 Detailed Description

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

Sparse matrix Class with mathematical matrix operations.

4.33.2 Constructor & Destructor Documentation

4.33.2.1 SparseMatrix()

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

default constructor (empty SparseMatrix)

Example:

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

Output:

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

4.33.3 Member Function Documentation

4.33.3.1 begin() [1/2]

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

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

The iterator points to the first row in the matrix.

Example:

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

4.33.3.2 begin() [2/2]

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

See also

cbegin() const

4.33.3.3 cbegin()

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

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

The iterator points to the first row in the matrix.

4.33.3.4 cend()

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

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

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

4.33.3.5 colsize()

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

get number of columns of the matrix

Example:

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

Output:

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

4.33.3.6 end() [1/2]

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

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

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

Example:

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

4.33.3.7 end() [2/2]

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

See also

cend() const

4.33.3.8 identity()

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

identity for the matrix

Example:

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

Output:

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

4.33.3.9 matchingIdentity()

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

creates a matching identity

Example:

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

Output:

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

4.33.3.10 mv()

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

matrix vector product y = A∗x

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

Parameters

in	*result*	reference to the resulting Vector
in	*x*	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	*scalar*	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	*scalar*	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 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	*result*	reference to the resulting Vector
in	*x*	constant reference to a Vector

4.33.4 Friends And Related Function Documentation

4.33.4.1 identity()

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

Function: make identity matrix

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

Parameters

in	*A*	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 hdnum::SquareRootProblem< N >

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

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

Template Parameters

N	a type representing x and F components

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

- src/newton.hh

4.35 hdnum::StationarySolver< M > Class Template Reference

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

`#include <pde.hh>`

Public Types

- typedef M::size_type **size_type**
  export size_type
- typedef M::time_type **time_type**
  export time_type
- typedef M::number_type **number_type**
  export number_type

Public Member Functions

- **StationarySolver** (const M &model_)
  constructor stores reference to the model
- void **solve** ()
  do one step
- const Vector< number_type > & **get_state** () const
  get current state
- size_type **get_order** () const
  return consistency order of the method

4.35.1 Detailed Description

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

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

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

M	the model type

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

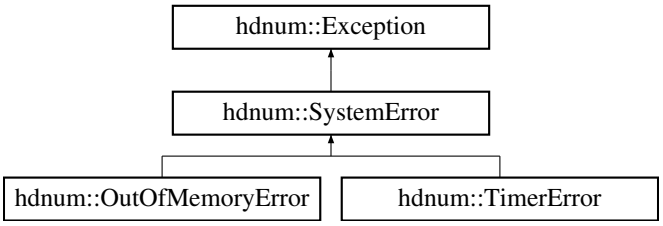
- src/pde.hh

4.36 hdnum::SystemError Class Reference

Default exception class for OS errors.

`#include <exceptions.hh>`

Inheritance diagram for hdnum::SystemError:



Additional Inherited Members

4.36.1 Detailed Description

Default exception class for OS errors.

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

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

- src/exceptions.hh

4.37 hdnum::Timer Class Reference

A simple stop watch.

`#include <timer.hh>`

Public Member Functions

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

4.37.1 Detailed Description

A simple stop watch.

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

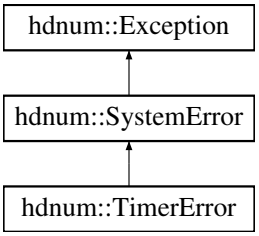
- src/timer.hh

4.38 hdnum::TimerError Class Reference

Exception thrown by the Timer class

`#include <timer.hh>`

Inheritance diagram for hdnum::TimerError:



Additional Inherited Members

4.38.1 Detailed Description

Exception thrown by the Timer class

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

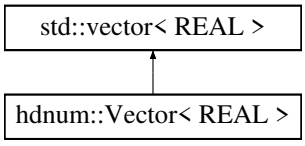
- src/timer.hh

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 = value.
- Vector sub (size_type i, size_type m)

  Subvector extraction.
- Vector & **operator*=** (const REAL value)

  *Multiplication by a scalar value (x *= value)*
- Vector & **operator/=** (const REAL value)

  Division by a scalar value (x /= value)
- Vector & **operator+=** (const Vector &y)

  Add another vector (x += y)
- Vector & **operator-=** (const Vector &y)

  Subtract another vector (x -= y)
- Vector & **update** (const REAL alpha, const Vector &y)

  Update vector by addition of a scaled vector (x += a y)
- REAL operator* (Vector &x) const

  Inner product with another vector.
- Vector operator+ (Vector &x) const

  Adding two vectors x+y.
- Vector operator- (Vector &x) const

  vector subtraction x-y
- REAL **two_norm_2** () const

  Square of the Euclidean norm.
- REAL two_norm () const

  Euclidean norm of a vector.
- bool **scientific** () const

  pretty-print output property: true = scientific, false = fixed point representation
- void scientific (bool b) const

  scientific(true) is the default, scientific(false) switches to the fixed point representation
- std::size_t **iwidth** () const

  get index field width for pretty-printing
- std::size_t **width** () const

  get data field width for pretty-printing
- std::size_t **precision** () const

  get data precision for pretty-printing
- void **iwidth** (std::size_t i) const

  set index field width for pretty-printing
- void **width** (std::size_t i) const

  set data field width for pretty-printing
- void **precision** (std::size_t i) const

  set data precision for pretty-printing

Related Functions

(Note that these are not member functions.)

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

4.39.1 Detailed Description

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

Class with mathematical vector operations.

4.39.2 Member Function Documentation

4.39.2.1 operator*()

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

Inner product with another vector.

Example:

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

Output:

```
x=
[ 0]      12.0000000
[ 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	*value*	constant value which should be assigned

Example:

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

Output:

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

4.39.2.5 scientific()

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

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

Example:

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

Output:

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

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

4.39.2.6 sub()

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

Subvector extraction.

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

Parameters

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

4.39.2.7 two_norm()

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

Euclidean norm of a vector.

Example:

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

Output:

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

euclidean norm of x = 3.0000000
```

4.39.3 Friends And Related Function Documentation

4.39.3.1 fill()

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

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

Example:

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

Output:

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

4.39.3.2 gnuplot()

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

Output contents of a Vector x to a text file named fname.

Example:

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

Output:

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

4.39.3.3 operator<<()

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

Output operator for Vector.

Example:

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

Output:

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

4.39.3.4 readVectorFromFile()

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

Read vector from a text file.

Parameters

in	*filename*	name of the text file
in,out	*vector*	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

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

```
65      inline const REAL& at(const std::size_t row, const std::size_t col) const {
66          return m_data[row * m_cols + col];
67      }
68
69  public:
71      DenseMatrix() : m_data(0, 0), m_rows(0), m_cols(0) {}
72
74      DenseMatrix(const std::size_t _rows, const std::size_t _cols,
75                  const REAL def_val = 0)
76          : m_data(_rows * _cols, def_val), m_rows(_rows), m_cols(_cols) {}
77
79      DenseMatrix(const std::initializer_list<std::initializer_list<REAL>>& v) {
80          m_rows = v.size();
81          m_cols = v.begin()->size();
82          for (auto row : v) {
83              if (row.size() != m_cols) {
84                  std::cout << "Zeilen der Matrix nicht gleich lang" << std::endl;
85                  exit(1);
86              }
87              for (auto elem : row) m_data.push_back(elem);
88          }
89      }
90
92      DenseMatrix(const hdnum::SparseMatrix<REAL>& other)
93          : m_data(other.rowsize() * other.colsize()), m_rows(other.rowsize()),
94            m_cols(other.colsize()) {
95          using counter_type = typename hdnum::SparseMatrix<REAL>::size_type;
96          counter_type row_index {};
97          for (auto& row : other) {
98              for (auto it = row.ibegin(); it != row.iend(); it++) {
99                  this->operator[](row_index)[it.index()] = it.value();
100              }
101              row_index++;
102          }
103      }
104
105      void addNewRow(const hdnum::Vector<REAL>& rowvector) {
106          m_rows++;
107          m_cols = rowvector.size();
108          for (std::size_t i = 0; i < m_cols; i++) m_data.push_back(rowvector[i]);
109      }
110
111      /*
112      // copy constructor (not needed, since it inherits from the STL vector)
113      DenseMatrix( const DenseMatrix& A )
114      {
115      this->m_data = A.m_data;
116      m_rows = A.m_rows;
117      m_cols = A.m_cols;
118      }
119      */
120
136      size_t rowsize() const { return m_rows; }
137
153      size_t colsize() const { return m_cols; }
154
155      // pretty-print output properties
156      bool scientific() const { return bScientific; }
157
179      void scientific(bool b) const { bScientific = b; }
180
182      std::size_t iwidth() const { return nIndexWidth; }
183
185      std::size_t width() const { return nValueWidth; }
186
188      std::size_t precision() const { return nValuePrecision; }
189
191      void iwidth(std::size_t i) const { nIndexWidth = i; }
192
194      void width(std::size_t i) const { nValueWidth = i; }
195
197      void precision(std::size_t i) const { nValuePrecision = i; }
198
242      // overloaded element access operators
243      // write access on matrix element A_ij using A(i,j)
244      inline REAL& operator()(const std::size_t row, const std::size_t col) {
245          assert(row < m_rows || col < m_cols);
246          return at(row, col);
247      }
248
250      inline const REAL& operator()(const std::size_t row,
251                                    const std::size_t col) const {
252          assert(row < m_rows || col < m_cols);
253          return at(row, col);
254      }
255
257      const ConstVectorIterator operator[](const std::size_t row) const {
```

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

```
545             for (std::size_t j = 0; j < colsize(); ++j)
546                 y[i] += (*this)(i, j) * x[j];
547         }
548     }
549
597     template <class V>
598     void umv(Vector<V>& y, const Vector<V>& x) const {
599         if (this->rowsize() != y.size())
600             HDNUM_ERROR("mv: size of A and y do not match");
601         if (this->colsize() != x.size())
602             HDNUM_ERROR("mv: size of A and x do not match");
603         for (std::size_t i = 0; i < rowsize(); ++i) {
604             for (std::size_t j = 0; j < colsize(); ++j)
605                 y[i] += (*this)(i, j) * x[j];
606         }
607     }
608
659     template <class V>
660     void umv(Vector<V>& y, const V& s, const Vector<V>& x) const {
661         if (this->rowsize() != y.size())
662             HDNUM_ERROR("mv: size of A and y do not match");
663         if (this->colsize() != x.size())
664             HDNUM_ERROR("mv: size of A and x do not match");
665         for (std::size_t i = 0; i < rowsize(); ++i) {
666             for (std::size_t j = 0; j < colsize(); ++j)
667                 y[i] += s * (*this)(i, j) * x[j];
668         }
669     }
670
719     void mm(const DenseMatrix<REAL>& A, const DenseMatrix<REAL>& B) {
720         if (this->rowsize() != A.rowsize())
721             HDNUM_ERROR("mm: size incompatible");
722         if (this->colsize() != B.colsize())
723             HDNUM_ERROR("mm: size incompatible");
724         if (A.colsize() != B.rowsize()) HDNUM_ERROR("mm: size incompatible");
725
726         for (std::size_t i = 0; i < rowsize(); i++)
727             for (std::size_t j = 0; j < colsize(); j++) {
728                 (*this)(i, j) = 0;
729                 for (std::size_t k = 0; k < A.colsize(); k++)
730                     (*this)(i, j) += A(i, k) * B(k, j);
731             }
732     }
733
787     void umm(const DenseMatrix<REAL>& A, const DenseMatrix<REAL>& B) {
788         if (this->rowsize() != A.rowsize())
789             HDNUM_ERROR("mm: size incompatible");
790         if (this->colsize() != B.colsize())
791             HDNUM_ERROR("mm: size incompatible");
792         if (A.colsize() != B.rowsize()) HDNUM_ERROR("mm: size incompatible");
793
794         for (std::size_t i = 0; i < rowsize(); i++)
795             for (std::size_t j = 0; j < colsize(); j++)
796                 for (std::size_t k = 0; k < A.colsize(); k++)
797                     (*this)(i, j) += A(i, k) * B(k, j);
798     }
799
833     void sc(const Vector<REAL>& x, std::size_t k) {
834         if (this->rowsize() != x.size()) HDNUM_ERROR("cc: size incompatible");
835
836         for (std::size_t i = 0; i < rowsize(); i++) (*this)(i, k) = x[i];
837     }
838
874     void sr(const Vector<REAL>& x, std::size_t k) {
875         if (this->colsize() != x.size()) HDNUM_ERROR("cc: size incompatible");
876
877         for (std::size_t i = 0; i < colsize(); i++) (*this)(k, i) = x[i];
878     }
879
881     REAL norm_infty() const {
882         REAL norm(0.0);
883         for (std::size_t i = 0; i < rowsize(); i++) {
884             REAL sum(0.0);
885             for (std::size_t j = 0; j < colsize(); j++)
886                 sum += myabs((*this)(i, j));
887             if (sum > norm) norm = sum;
888         }
889         return norm;
890     }
891
893     REAL norm_1() const {
894         REAL norm(0.0);
895         for (std::size_t j = 0; j < colsize(); j++) {
896             REAL sum(0.0);
897             for (std::size_t i = 0; i < rowsize(); i++)
898                 sum += myabs((*this)(i, j));
899             if (sum > norm) norm = sum;
```

```
900         }
901         return norm;
902     }
903
949     Vector<REAL> operator*(const Vector<REAL>& x) const {
950         assert(x.size() == colsize());
951
952         Vector<REAL> y(rowsize());
953         for (std::size_t r = 0; r < rowsize(); ++r) {
954             for (std::size_t c = 0; c < colsize(); ++c) {
955                 y[r] += at(r, c) * x[c];
956             }
957         }
958         return y;
959     }
960
1003     DenseMatrix operator*(const DenseMatrix& x) const {
1004         assert(colsize() == x.rowsize());
1005
1006         const std::size_t out_rows = rowsize();
1007         const std::size_t out_cols = x.colsize();
1008         DenseMatrix y(out_rows, out_cols, 0.0);
1009         for (std::size_t r = 0; r < out_rows; ++r)
1010             for (std::size_t c = 0; c < out_cols; ++c)
1011                 for (std::size_t i = 0; i < colsize(); ++i)
1012                     y(r, c) += at(r, i) * x(i, c);
1013
1014         return y;
1015     }
1016
1059     DenseMatrix operator+(const DenseMatrix& x) const {
1060         assert(colsize() == x.colsize());
1061         assert(rowsize() == x.rowsize());
1062
1063         const std::size_t out_rows = rowsize();
1064         const std::size_t out_cols = x.colsize();
1065         DenseMatrix y(out_rows, out_cols, 0.0);
1066         y = *this;
1067         y += x;
1068         return y;
1069     }
1070
1113     DenseMatrix operator-(const DenseMatrix& x) const {
1114         assert(colsize() == x.colsize());
1115         assert(rowsize() == x.rowsize());
1116
1117         const std::size_t out_rows = rowsize();
1118         const std::size_t out_cols = x.colsize();
1119         DenseMatrix y(out_rows, out_cols, 0.0);
1120         y = *this;
1121         y -= x;
1122         return y;
1123     }
1124 };
1125
1126 template <typename REAL>
1127 bool DenseMatrix<REAL>::bScientific = true;
1128 template <typename REAL>
1129 std::size_t DenseMatrix<REAL>::nIndexWidth = 10;
1130 template <typename REAL>
1131 std::size_t DenseMatrix<REAL>::nValueWidth = 10;
1132 template <typename REAL>
1133 std::size_t DenseMatrix<REAL>::nValuePrecision = 3;
1134
1158 template <typename REAL>
1159 inline std::ostream& operator«(std::ostream& s, const DenseMatrix<REAL>& A) {
1160     s « std::endl;
1161     s « " " « std::setw(A.iwidth()) « " "
1162       « "  ";
1163     for (typename DenseMatrix<REAL>::size_type j = 0; j < A.colsize(); ++j)
1164         s « std::setw(A.width()) « j « " ";
1165     s « std::endl;
1166
1167     for (typename DenseMatrix<REAL>::size_type i = 0; i < A.rowsize(); ++i) {
1168         s « " " « std::setw(A.iwidth()) « i « "  ";
1169         for (typename DenseMatrix<REAL>::size_type j = 0; j < A.colsize();
1170              ++j) {
1171             if (A.scientific()) {
1172                 s « std::setw(A.width()) « std::scientific « std::showpoint
1173                   « std::setprecision(A.precision()) « A[i][j] « " ";
1174             } else {
1175                 s « std::setw(A.width()) « std::fixed « std::showpoint
1176                   « std::setprecision(A.precision()) « A[i][j] « " ";
1177             }
1178         }
1179         s « std::endl;
1180     }
```

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

```
1422                 if (sub.length() > 0) {
1423                     REAL a = atof(sub.c_str());
1424                     // std::cout « std::fixed « std::setw(10) «
1425                     // std::setprecision(5) « a;
1426                     rowvector.push_back(a);
1427                 }
1428                 j++;
1429             }
1430             if (rowvector.size() > 0) {
1431                 A.addNewRow(rowvector);
1432                 i++;
1433                 // std::cout « std::endl;
1434             }
1435         }
1436         fin.close();
1437     } else {
1438         HDNUM_ERROR("Could not open file!");
1439     }
1440 }
1441
1475 template <typename REAL>
1476 inline void readMatrixFromFileMatrixMarket(const std::string& filename,
1477                                            DenseMatrix<REAL>& A) {
1478     std::string buffer;
1479     std::ifstream fin(filename.c_str());
1480     std::size_t i = 0;
1481     std::size_t j = 0;
1482     if (fin.is_open()) {
1483         // ignore all comments from the file (starting with %)
1484         while (fin.peek() == '%') fin.ignore(2048, '\n');
1485
1486         std::getline(fin, buffer);
1487         std::istringstream first_line(buffer);
1488         first_line » i » j;
1489         DenseMatrix<REAL> A_temp(i, j);
1490
1491         while (std::getline(fin, buffer)) {
1492             std::istringstream iss(buffer);
1493
1494             REAL value {};
1495             iss » i » j » value;
1496             // i-1, j-1, because matrix market does not use zero based indexing
1497             A_temp(i - 1, j - 1) = value;
1498         }
1499         A = A_temp;
1500         fin.close();
1501     } else {
1502         HDNUM_ERROR("Could not open file! \"" + filename + "\"");
1503     }
1504 }
1505
1506 }  // namespace hdnum
1507
1508 #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

E	exception class derived from Dune::Exception
m	reason for this exception in ostream-notation

Example:

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

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

5.3 exceptions.hh

Go to the documentation of this file.

```
1 #ifndef HDNUM_EXCEPTIONS_HH
2 #define HDNUM_EXCEPTIONS_HH
3
4 #include <string>
5 #include <sstream>
6
7 namespace hdnum {
8
35   class Exception {
36   public:
37         void message(const std::string &message);
38         const std::string& what() const;
39   private:
40         std::string _message;
41   };
42
43   inline void Exception::message(const std::string &message)
44   {
45         _message = message;
46   }
47
48   inline const std::string& Exception::what() const
49   {
50         return _message;
51   }
52
53   inline std::ostream& operator«(std::ostream &stream, const Exception &e)
54   {
55         return stream « e.what();
56   }
57
58   // the "format" the exception-type gets printed.  __FILE__ and
59   // __LINE__ are standard C-defines, the GNU cpp-infofile claims that
60   // C99 defines __func__ as well. __FUNCTION__ is a GNU-extension
61 #ifdef HDNUM_DEVEL_MODE
62 # define THROWSPEC(E) #E « " [" « __func__ « ":" « __FILE__ « ":" « __LINE__ « "]: "
63 #else
64 # define THROWSPEC(E) #E « ": "
65 #endif
66
84   // this is the magic: use the usual do { ... } while (0) trick, create
85   // the full message via a string stream and throw the created object
86 #define HDNUM_THROW(E, m) do { E th__ex; std::ostringstream th__out;            \
87         th__out « THROWSPEC(E) « m; th__ex.message(th__out.str()); throw th__ex; \
88   } while (0)
89
99   class IOError : public Exception {};
100
109   class MathError : public Exception {};
110
122   class RangeError : public Exception {};
123
131   class NotImplemented : public Exception {};
132
139   class SystemError : public Exception {};
140
144   class OutOfMemoryError : public SystemError {};
145
```

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

5.4 src/lr.hh File Reference

This file implements LU decomposition.

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

Functions

- template<class T >
  void **hdnum::lr** (DenseMatrix< T > &A, Vector< std::size_t > &p)
  compute lr decomposition of A with first nonzero pivoting
- template<class T >
  T **hdnum::abs** (const T &t)
  our own abs class that works also for multiprecision types
- template<class T >
  void **hdnum::lr_partialpivot** (DenseMatrix< T > &A, Vector< std::size_t > &p)
  lr decomposition of A with column pivoting
- template<class T >
  void **hdnum::lr_fullpivot** (DenseMatrix< T > &A, Vector< std::size_t > &p, Vector< std::size_t > &q)
  lr decomposition of A with full pivoting
- template<class T >
  void **hdnum::permute_forward** (const Vector< std::size_t > &p, Vector< T > &b)
  apply permutations to a right hand side vector
- template<class T >
  void **hdnum::permute_backward** (const Vector< std::size_t > &q, Vector< T > &z)
  apply permutations to a solution vector
- template<class T >
  void **hdnum::row_equilibrate** (DenseMatrix< T > &A, Vector< T > &s)
  perform a row equilibration of a matrix; return scaling for later use
- template<class T >
  void **hdnum::apply_equilibrate** (Vector< T > &s, Vector< T > &b)
  apply row equilibration to right hand side vector
- template<class T >
  void **hdnum::solveL** (const DenseMatrix< T > &A, Vector< T > &x, const Vector< T > &b)
  Assume L = lower triangle of A with l_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.

```
1 // -*- tab-width: 4; indent-tabs-mode: nil -*-
2 #ifndef HDNUM_LR_HH
3 #define HDNUM_LR_HH
4
5 #include "vector.hh"
6 #include "densematrix.hh"
7
12 namespace hdnum {
13
15   template<class T>
16   void lr (DenseMatrix<T>& A, Vector<std::size_t>& p)
17   {
18     if (A.rowsize()!=A.colsize() || A.rowsize()==0)
19       HDNUM_ERROR("need square and nonempty matrix");
20     if (A.rowsize()!=p.size())
21       HDNUM_ERROR("permutation vector incompatible with matrix");
22
23     // transformation to upper triangular
24     for (std::size_t k=0; k<A.rowsize()-1; ++k)
25       {
26         // find pivot element and exchange rows
27         for (std::size_t r=k; r<A.rowsize(); ++r)
28           if (A[r][k]!=0)
29             {
30               p[k] = r; // store permutation in step k
31               if (r>k) // exchange complete row if r!=k
32                 for (std::size_t j=0; j<A.colsize(); ++j)
33                   {
34                     T temp(A[k][j]);
35                     A[k][j] = A[r][j];
36                     A[r][j] = temp;
37                   }
38               break;
39             }
40         if (A[k][k]==0) HDNUM_ERROR("matrix is singular");
41
42         // modification
43         for (std::size_t i=k+1; i<A.rowsize(); ++i)
44           {
45             T qik(A[i][k]/A[k][k]);
46             A[i][k] = qik;
47             for (std::size_t j=k+1; j<A.colsize(); ++j)
48               A[i][j] -= qik * A[k][j];
49           }
50       }
51   }
52
54   template<class T>
55   T abs (const T& t)
56   {
57     if (t<0.0)
58       return -t;
59     else
60       return t;
61   }
62
64   template<class T>
65   void lr_partialpivot (DenseMatrix<T>& A, Vector<std::size_t>& p)
66   {
67     if (A.rowsize()!=A.colsize() || A.rowsize()==0)
68       HDNUM_ERROR("need square and nonempty matrix");
69     if (A.rowsize()!=p.size())
70       HDNUM_ERROR("permutation vector incompatible with matrix");
71
72     // initialize permutation
73     for (std::size_t k=0; k<A.rowsize(); ++k)
74       p[k] = k;
75
76     // transformation to upper triangular
77     for (std::size_t k=0; k<A.rowsize()-1; ++k)
78       {
79         // find pivot element
```

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

```
170  template<class T>
171  void permute_backward (const Vector<std::size_t>& q, Vector<T>& z)
172  {
173    if (z.size()!=q.size())
174      HDNUM_ERROR("permutation vector incompatible with z");
175
176    for (int k=z.size()-2; k>=0; --k)
177      if (q[k]!=std::size_t(k)) std::swap(z[k],z[q[k]]);
178  }
179
180  template<class T>
181  void row_equilibrate (DenseMatrix<T>& A, Vector<T>& s)
182  {
183    if (A.rowsize()*A.colsize()==0)
184      HDNUM_ERROR("need nonempty matrix");
185    if (A.rowsize()!=s.size())
186      HDNUM_ERROR("scaling vector incompatible with matrix");
187
188    // equilibrate row sums
189    for (std::size_t k=0; k<A.rowsize(); ++k)
190      {
191        s[k] = T(0.0);
192        for (std::size_t j=0; j<A.colsize(); ++j)
193          s[k] += abs(A[k][j]);
194        if (std::abs(s[k])==0) HDNUM_ERROR("row sum is zero");
195        for (std::size_t j=0; j<A.colsize(); ++j)
196          A[k][j] /= s[k];
197      }
198  }
199
200  template<class T>
202  void apply_equilibrate (Vector<T>& s, Vector<T>& b)
203  {
204    if (s.size()!=b.size())
205      HDNUM_ERROR("s and b incompatible");
206
207    // equilibrate row sums
208    for (std::size_t k=0; k<b.size(); ++k)
209      b[k] /= s[k];
210  }
211
212  template<class T>
214  void solveL (const DenseMatrix<T>& A, Vector<T>& x, const Vector<T>& b)
215  {
216    if (A.rowsize()!=A.colsize() || A.rowsize()==0)
217      HDNUM_ERROR("need square and nonempty matrix");
218    if (A.rowsize()!=b.size())
219      HDNUM_ERROR("right hand side incompatible with matrix");
220
221    for (std::size_t i=0; i<A.rowsize(); ++i)
222      {
223        T rhs(b[i]);
224        for (std::size_t j=0; j<i; j++)
225          rhs -= A[i][j] * x[j];
226        x[i] = rhs;
227      }
228  }
229
230  template<class T>
232  void solveR (const DenseMatrix<T>& A, Vector<T>& x, const Vector<T>& b)
233  {
234    if (A.rowsize()!=A.colsize() || A.rowsize()==0)
235      HDNUM_ERROR("need square and nonempty matrix");
236    if (A.rowsize()!=b.size())
237      HDNUM_ERROR("right hand side incompatible with matrix");
238
239    for (int i=A.rowsize()-1; i>=0; --i)
240      {
241        T rhs(b[i]);
242        for (std::size_t j=i+1; j<A.colsize(); j++)
243          rhs -= A[i][j] * x[j];
244        x[i] = rhs/A[i][i];
245      }
246  }
247
248  template<class T>
250  void linsolve (DenseMatrix<T>& A, Vector<T>& x, Vector<T>& b)
251  {
252    if (A.rowsize()!=A.colsize() || A.rowsize()==0)
253      HDNUM_ERROR("need square and nonempty matrix");
254    if (A.rowsize()!=b.size())
255      HDNUM_ERROR("right hand side incompatible with matrix");
256
257    Vector<T> s(x.size());
258    Vector<std::size_t> p(x.size());
259    Vector<std::size_t> q(x.size());
260    row_equilibrate(A,s);
261
```

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

5.6 src/newton.hh File Reference

Newton's method with line search.

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

Classes

- class hdnum::SquareRootProblem< N >

  Example class for a nonlinear model F(x) = 0;.
- class hdnum::GenericNonlinearProblem< Lambda, Vec >

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

  Solve nonlinear problem using a damped Newton method.
- class hdnum::Banach

  Solve nonlinear problem using a fixed point iteration.

Functions

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

  A function returning a problem class.

5.6.1 Detailed Description

Newton's method with line search.

5.6.2 Function Documentation

5.6.2.1 getNonlinearProblem()

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

A function returning a problem class.

Automatic template parameter extraction makes fiddling with types unnecessary.

Template Parameters

F	a lambda mapping a Vector to a Vector
X	the type for the Vector

5.7 newton.hh

Go to the documentation of this file.

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

```
97        Vec Fx(x.size());
98        F(x,Fx);
99        Vec z(x);
100        Vec Fz(x.size());
101
102        // numerische Jacobimatrix
103        for (int j=0; j<result.colsize(); ++j)
104          {
105            auto zj = z[j];
106            auto dz = (1.0+abs(zj))*eps;
107            z[j] += dz;
108            F(z,Fz);
109            for (int i=0; i<result.rowsize(); i++)
110              result[i][j] = (Fz[i]-Fx[i])/dz;
111            z[j] = zj;
112          }
113      }
114    };
115
123    template<typename F, typename X>
124    GenericNonlinearProblem<F,X> getNonlinearProblem (const F& f, const X& x, typename X::value_type eps =
       1e-7)
125    {
126      return GenericNonlinearProblem<F,X>(f,x,eps);
127    }
128
135    class Newton
136    {
137      typedef std::size_t size_type;
138
139    public:
141      Newton ()
142        : maxit(25), linesearchsteps(10), verbosity(0),
143          reduction(1e-14), abslimit(1e-30), converged(false)
144      {}
145
147      void set_maxit (size_type n)
148      {
149        maxit = n;
150      }
151
152      void set_sigma (double sigma_)
153      {
154
155      }
156
158      void set_linesearchsteps (size_type n)
159      {
160        linesearchsteps = n;
161      }
162
164      void set_verbosity (size_type n)
165      {
166        verbosity = n;
167      }
168
170      void set_abslimit (double l)
171      {
172        abslimit = l;
173      }
174
176      void set_reduction (double l)
177      {
178        reduction = l;
179      }
180
182      template<class M>
183      void solve (const M& model, Vector<typename M::number_type> & x) const
184      {
185        typedef typename M::number_type N;
186        // In complex case, we still need to use real valued numbers for residual norms etc.
187        using Real = typename std::conditional<std::is_same<std::complex<double>, N>::value, double,
       N>::type;
188        Vector<N> r(model.size());              // residual
189        DenseMatrix<N> A(model.size(),model.size()); // Jacobian matrix
190        Vector<N> y(model.size());              // temporary solution in line search
191        Vector<N> z(model.size());              // solution of linear system
192        Vector<N> s(model.size());              // scaling factors
193        Vector<size_type> p(model.size());                // row permutations
194        Vector<size_type> q(model.size());                // column permutations
195
196        model.F(x,r);                                    // compute nonlinear residual
197        Real R0(std::abs(norm(r)));                         // norm of initial residual
198        Real R(R0);                                  // current residual norm
199        if (verbosity>=1)
200          {
201            std::cout << "Newton "
```

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

```
288               if (verbosity>=1)
289                 std::cout « "Newton not converged within " « maxit « " iterations" « std::endl;
290             }
291         }
292     }
293
294     bool has_converged () const
295     {
296       return converged;
297     }
298     size_type iterations() const {
299       return iterations_taken;
300     }
301
302
303   private:
304     size_type maxit;
305     mutable size_type iterations_taken = -1;
306     size_type linesearchsteps;
307     size_type verbosity;
308     double reduction;
309     double abslimit;
310     mutable bool converged;
311   };
312
313
314
315
323   class Banach
324   {
325     typedef std::size_t size_type;
326
327   public:
329     Banach ()
330       : maxit(25), linesearchsteps(10), verbosity(0),
331         reduction(1e-14), abslimit(1e-30),  sigma(1.0), converged(false)
332     {}
333
335     void set_maxit (size_type n)
336     {
337       maxit = n;
338     }
339
341     void set_sigma (double sigma_)
342     {
343       sigma = sigma_;
344     }
345
347     void set_linesearchsteps (size_type n)
348     {
349       linesearchsteps = n;
350     }
351
353     void set_verbosity (size_type n)
354     {
355       verbosity = n;
356     }
357
359     void set_abslimit (double l)
360     {
361       abslimit = l;
362     }
363
365     void set_reduction (double l)
366     {
367       reduction = l;
368     }
369
371     template<class M>
372     void solve (const M& model, Vector<typename M::number_type>& x) const
373     {
374       typedef typename M::number_type N;
375       Vector<N> r(model.size());              // residual
376       Vector<N> y(model.size());              // temporary solution in line search
377
378       model.F(x,r);                           // compute nonlinear residual
379       N R0(norm(r));                          // norm of initial residual
380       N R(R0);                                // current residual norm
381       if (verbosity>=1)
382         {
383           std::cout « "Banach "
384                     « " norm=" « std::scientific « std::showpoint
385                     « std::setprecision(4) « R0
386                     « std::endl;
387         }
388
389       converged = false;
```

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

5.8 src/ode.hh File Reference

solvers for ordinary differential equations

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

Classes

- class hdnum::EE< M >

  Explicit Euler method as an example for an ODE solver.
- class hdnum::ModifiedEuler< M >

Modified Euler method (order 2 with 2 stages)

- class hdnum::Heun2< M >

  Heun method (order 2 with 2 stages)
- class hdnum::Heun3< M >

  Heun method (order 3 with 3 stages)
- class hdnum::Kutta3< M >

  Kutta method (order 3 with 3 stages)
- class hdnum::RungeKutta4< M >

  classical Runge-Kutta method (order 4 with 4 stages)
- class hdnum::RKF45< M >

  Adaptive Runge-Kutta-Fehlberg method.
- class hdnum::RE< M, S >

  Adaptive one-step method using Richardson extrapolation.
- class hdnum::IE< M, S >

  Implicit Euler using Newton's method to solve nonlinear system.
- class hdnum::DIRK< M, S >

  Implementation of a general Diagonal Implicit Runge-Kutta method.

Functions

- template<class T , class N >
  void **hdnum::gnuplot** (const std::string &fname, const std::vector< T > t, const std::vector< Vector< N > > u)

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

  gnuplot output for time and state sequence

5.8.1 Detailed Description

solvers for ordinary differential equations

5.9 ode.hh

Go to the documentation of this file.

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

```
37        : model(model_), u(model.size()), f(model.size())
38      {
39        model.initialize(t,u);
40        dt = 0.1;
41      }
42
44      void set_dt (time_type dt_)
45      {
46        dt = dt_;
47      }
48
50      void step ()
51      {
52        model.f(t,u,f);    // evaluate model
53        u.update(dt,f);    // advance state
54        t += dt;           // advance time
55      }
56
58      void set_state (time_type t_, const Vector<number_type>& u_)
59      {
60        t = t_;
61        u = u_;
62      }
63
65      const Vector<number_type>& get_state () const
66      {
67        return u;
68      }
69
71      time_type get_time () const
72      {
73        return t;
74      }
75
77      time_type get_dt () const
78      {
79        return dt;
80      }
81
83      size_type get_order () const
84      {
85        return 1;
86      }
87
88    private:
89      const M& model;
90      time_type t, dt;
91      Vector<number_type> u;
92      Vector<number_type> f;
93    };
94
103    template<class M>
104    class ModifiedEuler
105    {
106    public:
108      typedef typename M::size_type size_type;
109
111      typedef typename M::time_type time_type;
112
114      typedef typename M::number_type number_type;
115
117      ModifiedEuler (const M& model_)
118        : model(model_), u(model.size()), w(model.size()), k1(model.size()), k2(model.size())
119      {
120        c2 = 0.5;
121        a21 = 0.5;
122        b2 = 1.0;
123        model.initialize(t,u);
124        dt = 0.1;
125      }
126
128      void set_dt (time_type dt_)
129      {
130        dt = dt_;
131      }
132
134      void step ()
135      {
136        // stage 1
137        model.f(t,u,k1);
138
139        // stage 2
140        w = u;
141        w.update(dt*a21,k1);
142        model.f(t+c2*dt,w,k2);
143
144        // final
```

```
145       u.update(dt*b2,k2);
146       t += dt;
147     }
148
150     void set_state (time_type t_, const Vector<number_type>& u_)
151     {
152       t = t_;
153       u = u_;
154     }
155
157     const Vector<number_type>& get_state () const
158     {
159       return u;
160     }
161
163     time_type get_time () const
164     {
165       return t;
166     }
167
169     time_type get_dt () const
170     {
171       return dt;
172     }
173
175     size_type get_order () const
176     {
177       return 2;
178     }
179
180   private:
181     const M& model;
182     time_type t, dt;
183     time_type c2,a21,b2;
184     Vector<number_type> u,w;
185     Vector<number_type> k1,k2;
186   };
187
188
197   template<class M>
198   class Heun2
199   {
200   public:
202     typedef typename M::size_type size_type;
203
205     typedef typename M::time_type time_type;
206
208     typedef typename M::number_type number_type;
209
211     Heun2 (const M& model_)
212       : model(model_), u(model.size()), w(model.size()), k1(model.size()), k2(model.size())
213     {
214       c2 = 1.0;
215       a21 = 1.0;
216       b1 = 0.5;
217       b2 = 0.5;
218       model.initialize(t,u);
219       dt = 0.1;
220     }
221
223     void set_dt (time_type dt_)
224     {
225       dt = dt_;
226     }
227
229     void step ()
230     {
231       // stage 1
232       model.f(t,u,k1);
233
234       // stage 2
235       w = u;
236       w.update(dt*a21,k1);
237       model.f(t+c2*dt,w,k2);
238
239       // final
240       u.update(dt*b1,k1);
241       u.update(dt*b2,k2);
242       t += dt;
243     }
244
246     void set_state (time_type t_, const Vector<number_type>& u_)
247     {
248       t = t_;
249       u = u_;
250     }
251
```

```
253     const Vector<number_type>& get_state () const
254     {
255       return u;
256     }
257
259     time_type get_time () const
260     {
261       return t;
262     }
263
265     time_type get_dt () const
266     {
267       return dt;
268     }
269
271     size_type get_order () const
272     {
273       return 2;
274     }
275
276   private:
277     const M& model;
278     time_type t, dt;
279     time_type c2,a21,b1,b2;
280     Vector<number_type> u,w;
281     Vector<number_type> k1,k2;
282   };
283
284
293   template<class M>
294   class Heun3
295   {
296   public:
298     typedef typename M::size_type size_type;
299
301     typedef typename M::time_type time_type;
302
304     typedef typename M::number_type number_type;
305
307     Heun3 (const M& model_)
308       : model(model_), u(model.size()), w(model.size()), k1(model.size()),
309         k2(model.size()), k3(model.size())
310     {
311       c2 = time_type(1.0)/time_type(3.0);
312       c3 = time_type(2.0)/time_type(3.0);
313       a21 = time_type(1.0)/time_type(3.0);
314       a32 = time_type(2.0)/time_type(3.0);
315       b1 = 0.25;
316       b2 = 0.0;
317       b3 = 0.75;
318       model.initialize(t,u);
319       dt = 0.1;
320     }
321
323     void set_dt (time_type dt_)
324     {
325       dt = dt_;
326     }
327
329     void step ()
330     {
331       // stage 1
332       model.f(t,u,k1);
333
334       // stage 2
335       w = u;
336       w.update(dt*a21,k1);
337       model.f(t+c2*dt,w,k2);
338
339       // stage 3
340       w = u;
341       w.update(dt*a32,k2);
342       model.f(t+c3*dt,w,k3);
343
344       // final
345       u.update(dt*b1,k1);
346       u.update(dt*b3,k3);
347       t += dt;
348     }
349
351     void set_state (time_type t_, const Vector<number_type>& u_)
352     {
353       t = t_;
354       u = u_;
355     }
356
358     const Vector<number_type>& get_state () const
```

```
359     {
360       return u;
361     }
362
364     time_type get_time () const
365     {
366       return t;
367     }
368
370     time_type get_dt () const
371     {
372       return dt;
373     }
374
376     size_type get_order () const
377     {
378       return 3;
379     }
380
381   private:
382     const M& model;
383     time_type t, dt;
384     time_type c2,c3,a21,a31,a32,b1,b2,b3;
385     Vector<number_type> u,w;
386     Vector<number_type> k1,k2,k3;
387   };
388
397   template<class M>
398   class Kutta3
399   {
400   public:
402     typedef typename M::size_type size_type;
403
405     typedef typename M::time_type time_type;
406
408     typedef typename M::number_type number_type;
409
411     Kutta3 (const M& model_)
412       : model(model_), u(model.size()), w(model.size()), k1(model.size()),
413         k2(model.size()), k3(model.size())
414     {
415       c2 = 0.5;
416       c3 = 1.0;
417       a21 = 0.5;
418       a31 = -1.0;
419       a32 = 2.0;
420       b1 = time_type(1.0)/time_type(6.0);
421       b2 = time_type(4.0)/time_type(6.0);
422       b3 = time_type(1.0)/time_type(6.0);
423       model.initialize(t,u);
424       dt = 0.1;
425     }
426
428     void set_dt (time_type dt_)
429     {
430       dt = dt_;
431     }
432
434     void step ()
435     {
436       // stage 1
437       model.f(t,u,k1);
438
439       // stage 2
440       w = u;
441       w.update(dt*a21,k1);
442       model.f(t+c2*dt,w,k2);
443
444       // stage 3
445       w = u;
446       w.update(dt*a31,k1);
447       w.update(dt*a32,k2);
448       model.f(t+c3*dt,w,k3);
449
450       // final
451       u.update(dt*b1,k1);
452       u.update(dt*b2,k2);
453       u.update(dt*b3,k3);
454       t += dt;
455     }
456
458     void set_state (time_type t_, const Vector<number_type>& u_)
459     {
460       t = t_;
461       u = u_;
462     }
463
```

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

```
569     }
570
572     void set_state (time_type t_, const Vector<number_type>& u_)
573     {
574       t = t_;
575       u = u_;
576     }
577
579     const Vector<number_type>& get_state () const
580     {
581       return u;
582     }
583
585     time_type get_time () const
586     {
587       return t;
588     }
589
591     time_type get_dt () const
592     {
593       return dt;
594     }
595
597     size_type get_order () const
598     {
599       return 4;
600     }
601
602   private:
603     const M& model;
604     time_type t, dt;
605     time_type c2,c3,c4,a21,a32,a43,b1,b2,b3,b4;
606     Vector<number_type> u,w;
607     Vector<number_type> k1,k2,k3,k4;
608   };
609
614   template<class M>
615   class RKF45
616   {
617   public:
619     typedef typename M::size_type size_type;
620
622     typedef typename M::time_type time_type;
623
625     typedef typename M::number_type number_type;
626
628     RKF45 (const M& model_)
629       : model(model_), u(model.size()), w(model.size()), ww(model.size()), k1(model.size()),
630         k2(model.size()), k3(model.size()), k4(model.size()), k5(model.size()), k6(model.size()),
631         steps(0), rejected(0)
632     {
633       TOL = time_type(0.0001);
634       rho = time_type(0.8);
635       alpha = time_type(0.25);
636       beta = time_type(4.0);
637       dt_min = 1E-12;
638
639       c2 = time_type(1.0)/time_type(4.0);
640       c3 = time_type(3.0)/time_type(8.0);
641       c4 = time_type(12.0)/time_type(13.0);
642       c5 = time_type(1.0);
643       c6 = time_type(1.0)/time_type(2.0);
644
645       a21 = time_type(1.0)/time_type(4.0);
646
647       a31 = time_type(3.0)/time_type(32.0);
648       a32 = time_type(9.0)/time_type(32.0);
649
650       a41 = time_type(1932.0)/time_type(2197.0);
651       a42 = time_type(-7200.0)/time_type(2197.0);
652       a43 = time_type(7296.0)/time_type(2197.0);
653
654       a51 = time_type(439.0)/time_type(216.0);
655       a52 = time_type(-8.0);
656       a53 = time_type(3680.0)/time_type(513.0);
657       a54 = time_type(-845.0)/time_type(4104.0);
658
659       a61 = time_type(-8.0)/time_type(27.0);
660       a62 = time_type(2.0);
661       a63 = time_type(-3544.0)/time_type(2565.0);
662       a64 = time_type(1859.0)/time_type(4104.0);
663       a65 = time_type(-11.0)/time_type(40.0);
664
665       b1 = time_type(25.0)/time_type(216.0);
666       b2 = time_type(0.0);
667       b3 = time_type(1408.0)/time_type(2565.0);
668       b4 = time_type(2197.0)/time_type(4104.0);
```

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

```
759       //std::cout « "est. error=" « error « " dt_opt=" « dt_opt « std::endl;
760
761       if (error<=TOL)
762         {
763           t += dt;
764           u = ww;
765           dt = dt_opt;
766         }
767       else
768         {
769           rejected++;
770           dt = dt_opt;
771           if (dt>dt_min) step();
772         }
773     }
774
776     const Vector<number_type>& get_state () const
777     {
778       return u;
779     }
780
782     time_type get_time () const
783     {
784       return t;
785     }
786
788     time_type get_dt () const
789     {
790       return dt;
791     }
792
794     size_type get_order () const
795     {
796       return 5;
797     }
798
800     void get_info () const
801     {
802       std::cout « "RE: steps=" « steps « " rejected=" « rejected « std::endl;
803     }
804
805   private:
806     const M& model;
807     time_type t, dt;
808     time_type TOL,rho,alpha,beta,dt_min;
809     time_type c2,c3,c4,c5,c6;
810     time_type a21,a31,a32,a41,a42,a43,a51,a52,a53,a54,a61,a62,a63,a64,a65;
811     time_type b1,b2,b3,b4,b5; // 4th order
812     time_type bb1,bb2,bb3,bb4,bb5,bb6; // 5th order
813     Vector<number_type> u,w,ww;
814     Vector<number_type> k1,k2,k3,k4,k5,k6;
815     mutable size_type steps, rejected;
816   };
817
818
824   template<class M, class S>
825   class RE
826   {
827   public:
829     typedef typename M::size_type size_type;
830
832     typedef typename M::time_type time_type;
833
835     typedef typename M::number_type number_type;
836
838     RE (const M& model_, S& solver_)
839       : model(model_), solver(solver_), u(model.size()),
840         wlow(model.size()), whigh(model.size()), ww(model.size()),
841         steps(0), rejected(0)
842     {
843       model.initialize(t,u); // initialize state
844       dt = 0.1;              // set initial time step
845       two_power_m = 1.0;
846       for (size_type i=0; i<solver.get_order(); i++)
847         two_power_m *= 2.0;
848       TOL = time_type(0.0001);
849       rho = time_type(0.8);
850       alpha = time_type(0.25);
851       beta = time_type(4.0);
852       dt_min = 1E-12;
853     }
854
856     void set_dt (time_type dt_)
857     {
858       dt = dt_;
859     }
860
```

```
862     void set_TOL (time_type TOL_)
863     {
864       TOL = TOL_;
865     }
866
868     void step ()
869     {
870       // count steps done
871       steps++;
872
873       // do 1 step with 2*dt
874       time_type H(2.0*dt);
875       solver.set_state(t,u);
876       solver.set_dt(H);
877       solver.step();
878       wlow = solver.get_state();
879
880       // do 2 steps with dt
881       solver.set_state(t,u);
882       solver.set_dt(dt);
883       solver.step();
884       solver.step();
885       whigh = solver.get_state();
886
887       // estimate local error
888       ww = wlow;
889       ww -= whigh;
890       time_type error(norm(ww)/(pow(H,1.0+solver.get_order())*(1.0-1.0/two_power_m)));
891       time_type dt_opt(pow(rho*TOL/error,1.0/((time_type)solver.get_order())));
892       dt_opt = std::min(beta*dt,std::max(alpha*dt,dt_opt));
893       //std::cout << "est. error=" << error << " dt_opt=" << dt_opt << std::endl;
894
895       if (dt<=dt_opt)
896         {
897           t += H;
898           u = whigh;
899           u *= two_power_m;
900           u -= wlow;
901           u /= two_power_m-1.0;
902           dt = dt_opt;
903         }
904       else
905         {
906           rejected++;
907           dt = dt_opt;
908           if (dt>dt_min) step();
909         }
910     }
911
913     const Vector<number_type>& get_state () const
914     {
915       return u;
916     }
917
919     time_type get_time () const
920     {
921       return t;
922     }
923
925     time_type get_dt () const
926     {
927       return dt;
928     }
929
931     size_type get_order () const
932     {
933       return solver.get_order()+1;
934     }
935
937     void get_info () const
938     {
939       std::cout << "RE: steps=" << steps << " rejected=" << rejected << std::endl;
940     }
941
942   private:
943     const M& model;
944     S& solver;
945     time_type t, dt;
946     time_type two_power_m;
947     Vector<number_type> u,wlow,whigh,ww;
948     time_type TOL,rho,alpha,beta,dt_min;
949     mutable size_type steps, rejected;
950   };
951
952
962   template<class M, class S>
963   class IE
```

```
964   {
966     // h_n f(t_n, y_n) - y_n + y_{n-1} = 0
967     class NonlinearProblem
968     {
969     public:
971       typedef typename M::size_type size_type;
972
974       typedef typename M::number_type number_type;
975
977       NonlinearProblem (const M& model_, const Vector<number_type>& yold_,
978                         typename M::time_type tnew_, typename M::time_type dt_)
979         : model(model_), yold(yold_), tnew(tnew_), dt(dt_)
980       {}
981
983       std::size_t size () const
984       {
985         return model.size();
986       }
987
989       void F (const Vector<number_type>& x, Vector<number_type>& result) const
990       {
991         model.f(tnew,x,result);
992         result *= dt;
993         result -= x;
994         result += yold;
995       }
996
998       void F_x (const Vector<number_type>& x, DenseMatrix<number_type>& result) const
999       {
1000         model.f_x(tnew,x,result);
1001         result *= dt;
1002         for (size_type i=0; i<model.size(); i++) result[i][i] -= number_type(1.0);
1003       }
1004
1005       void set_tnew_dt (typename M::time_type tnew_, typename M::time_type dt_)
1006       {
1007         tnew = tnew_;
1008         dt = dt_;
1009       }
1010
1011     private:
1012       const M& model;
1013       const Vector<number_type>& yold;
1014       typename M::time_type tnew;
1015       typename M::time_type dt;
1016     };
1017
1018   public:
1020     typedef typename M::size_type size_type;
1021
1023     typedef typename M::time_type time_type;
1024
1026     typedef typename M::number_type number_type;
1027
1029     IE (const M& model_, const S& newton_)
1030       : verbosity(0), model(model_), newton(newton_), u(model.size()), unew(model.size())
1031     {
1032       model.initialize(t,u);
1033       dt = dtmax = 0.1;
1034     }
1035
1037     void set_dt (time_type dt_)
1038     {
1039       dt = dtmax = dt_;
1040     }
1041
1043     void set_verbosity (size_type verbosity_)
1044     {
1045       verbosity = verbosity_;
1046     }
1047
1049     void step ()
1050     {
1051       if (verbosity>=2)
1052         std::cout << "IE: step" << " t=" << t << " dt=" << dt << std::endl;
1053       NonlinearProblem nlp(model,u,t+dt,dt);
1054       bool reduced = false;
1055       error = false;
1056       while (1)
1057         {
1058           unew = u;
1059           newton.solve(nlp,unew);
1060           if (newton.has_converged())
1061             {
1062               u = unew;
1063               t += dt;
1064               if (!reduced && dt<dtmax-1e-13)
```

```
1065                {
1066                  dt = std::min(2.0*dt,dtmax);
1067                  if (verbosity>0)
1068                    std::cout << "IE: increasing time step to " << dt << std::endl;
1069                }
1070              return;
1071            }
1072          else
1073            {
1074              if (dt<1e-12)
1075                {
1076                  HDNUM_ERROR("time step too small in implicit Euler");
1077                  error = true;
1078                  break;
1079                }
1080              dt *= 0.5;
1081              reduced = true;
1082              nlp.set_tnew_dt(t+dt,dt);
1083              if (verbosity>0) std::cout << "IE: reducing time step to " << dt << std::endl;
1084            }
1085        }
1086    }
1087
1089    bool get_error () const
1090    {
1091      return error;
1092    }
1093
1095    void set_state (time_type t_, const Vector<number_type>& u_)
1096    {
1097      t = t_;
1098      u = u_;
1099    }
1100
1102    const Vector<number_type>& get_state () const
1103    {
1104      return u;
1105    }
1106
1108    time_type get_time () const
1109    {
1110      return t;
1111    }
1112
1114    time_type get_dt () const
1115    {
1116      return dt;
1117    }
1118
1120    size_type get_order () const
1121    {
1122      return 1;
1123    }
1124
1126    void get_info () const
1127    {
1128    }
1129
1130  private:
1131    size_type verbosity;
1132    const M& model;
1133    const S& newton;
1134    time_type t, dt, dtmax;
1135    number_type reduction;
1136    size_type linesearchsteps;
1137    Vector<number_type> u;
1138    Vector<number_type> unew;
1139    mutable bool error;
1140  };
1141
1152  template<class M, class S>
1153  class DIRK
1154  {
1155  public:
1156
1158    typedef typename M::size_type size_type;
1159
1161    typedef typename M::time_type time_type;
1162
1164    typedef typename M::number_type number_type;
1165
1167    typedef DenseMatrix<number_type> ButcherTableau;
1168
1169  private:
1170
1173    static ButcherTableau initTableau(const std::string method)
1174    {
```

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

```
1262        return 3;
1263      }
1264      else if(method.find("Midpoint Rule") != std::string::npos){
1265        return 2;
1266      }
1267      else if(method.find("Fractional Step Theta") != std::string::npos){
1268        return 2;
1269      }
1270      else{
1271        HDNUM_ERROR("Order not available for Runge Kutta solver.");
1272      }
1273    }
1274
1275
1277    // h_n f(t_n, y_n) - y_n + y_{n-1} = 0
1278    class NonlinearProblem
1279    {
1280    public:
1282      typedef typename M::size_type size_type;
1283
1285      typedef typename M::number_type number_type;
1286
1288      NonlinearProblem (const M& model_, const Vector<number_type>& yold_,
1289                        typename M::time_type told_, typename M::time_type dt_,
1290                        const ButcherTableau & butcher_, const int rk_step_,
1291                        const std::vector< Vector<number_type> > & k_)
1292        : model(model_), yold(yold_), told(told_),
1293          dt(dt_), butcher(butcher_), rk_step(rk_step_), k_old(model.size(),0)
1294      {
1295        for(int i=0; i<rk_step; ++i)
1296          k_old.update(butcher[rk_step][1+i] * dt, k_[i]);
1297      }
1298
1300      std::size_t size () const
1301      {
1302        return model.size();
1303      }
1304
1306      void F (const Vector<number_type>& x, Vector<number_type>& result) const
1307      {
1308        result = k_old;
1309
1310        Vector<number_type> current_z(x);
1311        current_z.update(1.,yold);
1312
1313        const number_type tnew = told + butcher[rk_step][0] * dt;
1314
1315        Vector<number_type> current_k(model.size(),0.);
1316        model.f(tnew,current_z,current_k);
1317        result.update(butcher[rk_step][rk_step+1] * dt, current_k);
1318
1319        result.update(-1.,x);
1320      }
1321
1323      void F_x (const Vector<number_type>& x, DenseMatrix<number_type>& result) const
1324      {
1325        const number_type tnew = told + butcher[rk_step][0] * dt;
1326
1327        Vector<number_type> current_z(x);
1328        current_z.update(1.,yold);
1329
1330        model.f_x(tnew,current_z,result);
1331
1332        result *= dt * butcher[rk_step][rk_step+1];
1333
1334        for (size_type i=0; i<model.size(); i++) result[i][i] -= number_type(1.0);
1335      }
1336
1337      void set_told_dt (typename M::time_type told_, typename M::time_type dt_)
1338      {
1339        told = told_;
1340        dt = dt_;
1341      }
1342
1343    private:
1344      const M& model;
1345      const Vector<number_type>& yold;
1346      typename M::time_type told;
1347      typename M::time_type dt;
1348      const ButcherTableau & butcher;
1349      const int rk_step;
1350      Vector<number_type> k_old;
1351    };
1352
1353  public:
1354
1357    DIRK (const M& model_, const S& newton_, const ButcherTableau & butcher_, const int order_)
```

```
1358        : verbosity(0), butcher(butcher_), model(model_), newton(newton_),
1359          u(model.size()), order(order_)
1360      {
1361        model.initialize(t,u);
1362        dt = dtmax = 0.1;
1363      }
1364
1367      DIRK (const M& model_, const S& newton_, const std::string method)
1368        : verbosity(0), butcher(initTableau(method)), model(model_), newton(newton_), u(model.size()),
1369          order(initOrder(method))
1370      {
1371        model.initialize(t,u);
1372        dt = dtmax = 0.1;
1373      }
1374
1375
1377      void set_dt (time_type dt_)
1378      {
1379        dt = dtmax = dt_;
1380      }
1381
1383      void set_verbosity (size_type verbosity_)
1384      {
1385        verbosity = verbosity_;
1386      }
1387
1389      void step ()
1390      {
1391
1392        const size_type R = butcher.colsize()-1;
1393
1394        bool reduced = false;
1395        error = false;
1396        if(verbosity>=2)
1397          std::cout « "DIRK: step to" « " t+dt=" « t+dt « " dt=" « dt « std::endl;
1398
1399        while (1)
1400          {
1401            bool converged = true;
1402
1403            // Perform R Runge-Kutta steps
1404            std::vector< Vector<number_type> > k;
1405            for(size_type i=0; i<R; ++i) {
1406              if (verbosity>=2)
1407                std::cout « "DIRK: step nr "« i « std::endl;
1408
1409              Vector<number_type> current_z(model.size(),0.0);
1410
1411              // Set starting value of k_i
1412              // model.f(t,u,current_k);
1413
1414              // Solve nonlinear problem
1415              NonlinearProblem nlp(model,u,t,dt,butcher,i,k);
1416
1417              newton.solve(nlp,current_z);
1418
1419              converged = converged && newton.has_converged();
1420              if(!converged)
1421                break;
1422
1423              current_z.update(1., u);
1424              const number_type t_i = t + butcher[i][0] * dt;
1425              Vector<number_type>current_k(model.size(),0.);
1426              model.f(t_i,current_z,current_k);
1427
1428              k.push_back( current_k );
1429            }
1430
1431            if (converged)
1432              {
1433                if(verbosity >= 2)
1434                  std::cout « "DIRK finished"« std::endl;
1435
1436                // Update to new solution
1437                for(size_type i=0; i<R; ++i)
1438                  u.update(dt*butcher[R][1+i],k[i]);
1439
1440                t += dt;
1441                if (!reduced && dt<dtmax-1e-13)
1442                  {
1443                    dt = std::min(2.0*dt,dtmax);
1444                    if (verbosity>0)
1445                      std::cout « "DIRK: increasing time step to " « dt « std::endl;
1446                  }
1447                return;
1448              }
1449            else
```

```
1450            {
1451              if (dt<1e-12)
1452                {
1453                  HDNUM_ERROR("time step too small in implicit Euler");
1454                  error = true;
1455                  break;
1456                }
1457              dt *= 0.5;
1458              reduced = true;
1459              if (verbosity>0) std::cout << "DIRK: reducing time step to " << dt << std::endl;
1460            }
1461        }
1462    }
1463
1465    bool get_error () const
1466    {
1467      return error;
1468    }
1469
1471    void set_state (time_type t_, const Vector<number_type>& u_)
1472    {
1473      t = t_;
1474      u = u_;
1475    }
1476
1478    const Vector<number_type>& get_state () const
1479    {
1480      return u;
1481    }
1482
1484    time_type get_time () const
1485    {
1486      return t;
1487    }
1488
1490    time_type get_dt () const
1491    {
1492      return dt;
1493    }
1494
1496    size_type get_order () const
1497    {
1498      return order;
1499    }
1500
1502    void get_info () const
1503    {
1504    }
1505
1506  private:
1507    size_type verbosity;
1508    const DenseMatrix<number_type> butcher;
1509    const M& model;
1510    const S& newton;
1511    time_type t, dt, dtmax;
1512    number_type reduction;
1513    size_type linesearchsteps;
1514    Vector<number_type> u;
1515    int order;
1516    mutable bool error;
1517  };
1518
1520  template<class T, class N>
1521  inline void gnuplot (const std::string& fname, const std::vector<T> t, const std::vector<Vector<N> >
     u)
1522  {
1523    std::fstream f(fname.c_str(),std::ios::out);
1524    for (typename std::vector<T>::size_type n=0; n<t.size(); n++)
1525      {
1526        f << std::scientific << std::showpoint
1527          << std::setprecision(16) << t[n];
1528        for (typename Vector<N>::size_type i=0; i<u[n].size(); i++)
1529          f << " " << std::scientific << std::showpoint
1530            << std::setprecision(u[n].precision()) << u[n][i];
1531        f << std::endl;
1532      }
1533    f.close();
1534  }
1535
1537  template<class T, class N>
1538  inline void gnuplot (const std::string& fname, const std::vector<T> t, const std::vector<Vector<N> >
     u, const std::vector<T> dt)
1539  {
1540    std::fstream f(fname.c_str(),std::ios::out);
1541    for (typename std::vector<T>::size_type n=0; n<t.size(); n++)
1542      {
1543        f << std::scientific << std::showpoint
```

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

5.10 src/opcounter.hh File Reference

This file implements an operator counting class.

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

Classes

- class hdnum::oc::OpCounter< F >
- struct hdnum::oc::OpCounter< F >::Counters

  Struct storing the number of operations.

Functions

- template<typename F >
  OpCounter< F > **hdnum::oc::operator-** (const OpCounter< F > &a)
- template<typename F >
  OpCounter< F > **hdnum::oc::operator+** (const OpCounter< F > &a, const OpCounter< F > &b)
- template<typename F >
  OpCounter< F > **hdnum::oc::operator+** (const OpCounter< F > &a, const F &b)
- template<typename F >
  OpCounter< F > **hdnum::oc::operator+** (const F &a, const OpCounter< F > &b)
- template<typename F , typename T >
  std::enable_if< std::is_arithmetic< T >::value, OpCounter< F > >::type **hdnum::oc::operator+** (const OpCounter< F > &a, const T &b)
- template<typename F , typename T >
  std::enable_if< std::is_arithmetic< T >::value, OpCounter< F > >::type **hdnum::oc::operator+** (const T &a, const OpCounter< F > &b)
- template<typename F >
  OpCounter< F > & **hdnum::oc::operator+=** (OpCounter< F > &a, const OpCounter< F > &b)
- template<typename F >
  OpCounter< F > & **hdnum::oc::operator+=** (OpCounter< F > &a, const F &b)
- template<typename F , typename T >
  std::enable_if< std::is_arithmetic< T >::value, OpCounter< F > & >::type **hdnum::oc::operator+=** (OpCounter< F > &a, const T &b)
- template<typename F >
  OpCounter< F > **hdnum::oc::operator-** (const OpCounter< F > &a, const OpCounter< F > &b)

- 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 >
  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 >
  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 >
  bool **hdnum::oc::operator<** (const OpCounter< F > &a, const OpCounter< F > &b)

- template<typename F >
  bool **hdnum::oc::operator<** (const OpCounter< F > &a, const F &b)
- template<typename F >
  bool **hdnum::oc::operator<** (const F &a, const OpCounter< F > &b)
- template<typename F , typename T >
  bool **hdnum::oc::operator<** (const OpCounter< F > &a, const T &b)
- template<typename F , typename T >
  bool **hdnum::oc::operator<** (const T &a, const OpCounter< F > &b)
- template<typename F >
  bool **hdnum::oc::operator<=** (const OpCounter< F > &a, const OpCounter< F > &b)
- template<typename F >
  bool **hdnum::oc::operator<=** (const OpCounter< F > &a, const F &b)
- template<typename F >
  bool **hdnum::oc::operator<=** (const F &a, const OpCounter< F > &b)
- template<typename F , typename T >
  bool **hdnum::oc::operator<=** (const OpCounter< F > &a, const T &b)
- template<typename F , typename T >
  bool **hdnum::oc::operator<=** (const T &a, const OpCounter< F > &b)
- template<typename F >
  bool **hdnum::oc::operator>** (const OpCounter< F > &a, const OpCounter< F > &b)
- template<typename F >
  bool **hdnum::oc::operator>** (const OpCounter< F > &a, const F &b)
- template<typename F >
  bool **hdnum::oc::operator>** (const F &a, const OpCounter< F > &b)
- template<typename F , typename T >
  bool **hdnum::oc::operator>** (const OpCounter< F > &a, const T &b)
- template<typename F , typename T >
  bool **hdnum::oc::operator>** (const T &a, const OpCounter< F > &b)
- template<typename F >
  bool **hdnum::oc::operator>=** (const OpCounter< F > &a, const OpCounter< F > &b)
- template<typename F >
  bool **hdnum::oc::operator>=** (const OpCounter< F > &a, const F &b)
- template<typename F >
  bool **hdnum::oc::operator>=** (const F &a, const OpCounter< F > &b)
- template<typename F , typename T >
  bool **hdnum::oc::operator>=** (const OpCounter< F > &a, const T &b)
- template<typename F , typename T >
  bool **hdnum::oc::operator>=** (const T &a, const OpCounter< F > &b)
- template<typename F >
  bool **hdnum::oc::operator!=** (const OpCounter< F > &a, const OpCounter< F > &b)
- template<typename F >
  bool **hdnum::oc::operator!=** (const OpCounter< F > &a, const F &b)
- template<typename F >
  bool **hdnum::oc::operator!=** (const F &a, const OpCounter< F > &b)
- template<typename F , typename T >
  bool **hdnum::oc::operator!=** (const OpCounter< F > &a, const T &b)
- template<typename F , typename T >
  bool **hdnum::oc::operator!=** (const T &a, const OpCounter< F > &b)
- template<typename F >
  bool **hdnum::oc::operator==** (const OpCounter< F > &a, const OpCounter< F > &b)
- template<typename F >
  bool **hdnum::oc::operator==** (const OpCounter< F > &a, const F &b)
- template<typename F >
  bool **hdnum::oc::operator==** (const F &a, const OpCounter< F > &b)
- template<typename F , typename T >
  bool **hdnum::oc::operator==** (const OpCounter< F > &a, const T &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.

```
1 // -*- tab-width: 4; indent-tabs-mode: nil -*-
2 #ifndef __OPCOUNTER__
3 #define __OPCOUNTER__
4
5 #include <type_traits>
6 #include <iostream>
7 #include <cmath>
8 #include <cstdlib>
9
14 namespace hdnum {
15   namespace oc {
16     template<typename F>
17     class OpCounter;
18   }
19 }
20
21 namespace hdnum {
22   namespace oc {
28     template<typename F>
29     class OpCounter
30     {
31
32     public:
33
34       using size_type = std::size_t;
35
36       using value_type = F;
37
38       OpCounter()
39         : _v()
40       {}
41
42       template<typename T>
43       OpCounter(const T& t, typename std::enable_if<std::is_same<T,int>::value and
      !std::is_same<F,int>::value>::type* = nullptr)
```

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

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

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

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

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

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

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

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

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

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

5.12 src/pde.hh File Reference

solvers for partial differential equations

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

Classes

- class hdnum::StationarySolver< M >

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

Functions

- template<class N , class G >
  void **hdnum::pde_gnuplot2d** (const std::string &fname, const Vector< N > solution, const G &grid)

  gnuplot output for stationary state

5.12.1 Detailed Description

solvers for partial differential equations

5.13 pde.hh

Go to the documentation of this file.

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

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

5.14 src/precision.hh File Reference

find machine precision for given float type

Functions

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

5.14.1 Detailed Description

find machine precision for given float type

5.15 precision.hh

Go to the documentation of this file.

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

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

5.16 src/qr.hh File Reference

This file implements QR decomposition using Gram-Schmidt method.

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

Functions

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

  computes orthonormal basis of Im(A) using classical Gram-Schmidt
- template<class T >
  DenseMatrix< T > hdnum::modified_gram_schmidt (const DenseMatrix< T > &A)

  computes orthonormal basis of Im(A) using modified Gram-Schmidt
- template<class T >
  DenseMatrix< T > hdnum::qr_gram_schmidt_simple (DenseMatrix< T > &Q)

  computes qr decomposition using modified Gram-Schmidt - works only with small (m>n) and square matrices
- template<class T >
  DenseMatrix< T > hdnum::qr_gram_schmidt (DenseMatrix< T > &Q)

  computes qr decomposition using modified Gram-Schmidt - works only with small (m>n) and square matrices
- template<class T >
  DenseMatrix< T > hdnum::qr_gram_schmidt_pivoting (DenseMatrix< T > &Q, Vector< int > &p, int &rank, T threshold=0.00000000001)

  computes qr decomposition using modified Gram-Schmidt and pivoting - works with all types of matrices
- template<typename T >
  void hdnum::permute_forward (DenseMatrix< T > &A, Vector< int > &p)

  applies a permutation vector to a matrix

5.16.1 Detailed Description

This file implements QR decomposition using Gram-Schmidt method.

5.16.2 Function Documentation

5.16.2.1 gram_schmidt()

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

computes orthonormal basis of Im(A) using classical Gram-Schmidt

Template Parameters

hdnum::DenseMatrix<T>	A

Example:

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

Output:

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

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

5.16.2.2 modified_gram_schmidt()

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

computes orthonormal basis of Im(A) using modified Gram-Schmidt

Template Parameters

hdnum::DenseMatrix<T>	A

Example:

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

Output:

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

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

5.16.2.3 permute_forward()

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

applies a permutation vector to a matrix

Template Parameters

hdnum::DenseMatrix<T>	A

Parameters

hdnum::Vector<int>	p

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

hdnum::DenseMatrix<T>	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

hdnum::DenseMatrix<T>	Q
T	threshold (optional)

Parameters

hdnum::Vector<int>	p
int	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

hdnum::DenseMatrix<T>	Q

Example:

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

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

Output:

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

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

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

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

5.17 qr.hh

Go to the documentation of this file.

```
1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 /*
3  * File:   qr.hh
4  * Author: Raphael Vogt <cx238@stud.uni-heidelberg.de>
5  *
6  * Created on August 30, 2020
7  */
8
9 #ifndef HDNUM_QR_HH
10 #define HDNUM_QR_HH
11
12 #include <cmath>
13 #include <utility>
14
15 #include "densematrix.hh"
16 #include "vector.hh"
17
22 namespace hdnum {
23
52 template <class T>
53 DenseMatrix<T> gram_schmidt(const DenseMatrix<T>& A) {
54     DenseMatrix<T> Q(A);
55
56     // for all columns except the first
57     for (int k = 1; k < Q.colsize(); k++) {
58         // orthogonalize column k against all previous
59         for (int j = 0; j < k; j++) {
60             // compute factor
61             T sum_nom(0.0);
62             T sum_denom(0.0);
63             for (int i = 0; i < Q.rowsize(); i++) {
64                 sum_nom += A[i][k] * Q[i][j];
65                 sum_denom += Q[i][j] * Q[i][j];
66             }
67             // modify
68             T alpha = sum_nom / sum_denom;
69             for (int i = 0; i < Q.rowsize(); i++) Q[i][k] -= alpha * Q[i][j];
70         }
71     }
72     for (int j = 0; j < Q.colsize(); j++) {
73         // compute norm of column j
74         T sum(0.0);
75         for (int i = 0; i < Q.rowsize(); i++) sum += Q[i][j] * Q[i][j];
76         sum = sqrt(sum);
77         // scale
78         for (int i = 0; i < Q.rowsize(); i++) Q[i][j] = Q[i][j] / sum;
```

```
79      }
80      return Q;
81  }
82
111 template <class T>
112 DenseMatrix<T> modified_gram_schmidt(const DenseMatrix<T>& A) {
113     DenseMatrix<T> Q(A);
114
115     for (int k = 0; k < Q.colsize(); k++) {
116         // modify all later columns with column k
117         for (int j = k + 1; j < Q.colsize(); j++) {
118             // compute factor
119             T sum_nom(0.0);
120             T sum_denom(0.0);
121             for (int i = 0; i < Q.rowsize(); i++) {
122                 sum_nom += Q[i][j] * Q[i][k];
123                 sum_denom += Q[i][k] * Q[i][k];
124             }
125             // modify
126             T alpha = sum_nom / sum_denom;
127             for (int i = 0; i < Q.rowsize(); i++) Q[i][j] -= alpha * Q[i][k];
128         }
129     }
130     for (int j = 0; j < Q.colsize(); j++) {
131         // compute norm of column j
132         T sum(0.0);
133         for (int i = 0; i < Q.rowsize(); i++) sum += Q[i][j] * Q[i][j];
134         sum = sqrt(sum);
135         // scale
136         for (int i = 0; i < Q.rowsize(); i++) Q[i][j] = Q[i][j] / sum;
137     }
138     return Q;
139 }
140
182 template <class T>
183 DenseMatrix<T> qr_gram_schmidt_simple(DenseMatrix<T>& Q) {
184     // save matrix A, before it's replaced with Q
185     DenseMatrix<T> A(Q);
186
187     // create matrix R
188     DenseMatrix<T> R(Q.colsize(), Q.colsize());
189
190     // start orthogonalizing
191     for (int k = 0; k < Q.colsize(); k++) {
192         // modify all later columns with column k
193         for (int j = k + 1; j < Q.colsize(); j++) {
194             // compute factor
195             T sum_nom(0.0);
196             T sum_denom(0.0);
197             for (int i = 0; i < Q.rowsize(); i++) {
198                 sum_nom += Q(i, j) * Q(i, k);
199                 sum_denom += Q(i, k) * Q(i, k);
200             }
201
202             T alpha = sum_nom / sum_denom;
203             for (int i = 0; i < Q.rowsize(); i++) Q(i, j) -= alpha * Q(i, k);
204         }
205     }
206
207     // add values to R, except main diagonal
208     for (int i = 1; i < R.colsize(); i++) {
209         for (int j = 0; j < i; j++) {
210             T sum_nom(0.0);
211             T sum_l2nom(0.0);
212             for (int k = 0; k < Q.rowsize(); k++) {
213                 sum_nom += A(k, i) * Q(k, j);
214                 sum_l2nom += Q(k, j) * Q(k, j);
215             }
216             sum_l2nom = sqrt(sum_l2nom);
217             // add element
218             R(j, i) = sum_nom / sum_l2nom;
219         }
220     }
221
222     // add missing values and scale
223     for (int j = 0; j < Q.colsize(); j++) {
224         // compute norm of column j
225         T sum(0.0);
226         for (int i = 0; i < Q.rowsize(); i++) sum += Q(i, j) * Q(i, j);
227         sum = sqrt(sum);
228         // add main diagonal to R
229         R(j, j) = sum;
230         // scale Q
231         for (int i = 0; i < Q.rowsize(); i++) Q(i, j) = Q(i, j) / sum;
232     }
233     return R;
234 }
```

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

```
425         // swap columns if necessary
426         if (p[k] > k) {
427             for (int r = 0; r < Q.rowsize(); r++) {
428                 T temp_Q = Q(r, k);
429                 Q(r, k) = Q(r, p[k]);
430                 Q(r, p[k]) = temp_Q;
431             }
432             p[p[k]] = k;
433 
434             // compute norm of the new column k
435             norm_k = 0;
436             for (int i = 0; i < Q.rowsize(); i++) {
437                 norm_k += Q(i, k) * Q(i, k);
438             }
439             norm_k = sqrt(norm_k);
440         }
441 
442         // if norm of column k > threshold -> column k is linear independent
443         if (norm_k > threshold) {
444             rank++;
445         } else {
446             break;
447         }
448 
449         // modify all later columns with column k
450         for (int j = k + 1; j < Q.colsize(); j++) {
451             // compute factor
452             T sum_nom(0.0);
453             T sum_denom(0.0);
454             for (int i = 0; i < Q.rowsize(); i++) {
455                 sum_nom += Q(i, j) * Q(i, k);
456                 sum_denom += Q(i, k) * Q(i, k);
457             }
458 
459             T alpha = sum_nom / sum_denom;
460             for (int i = 0; i < Q.rowsize(); i++) Q(i, j) -= alpha * Q(i, k);
461         }
462     }
463 
464     // add values to R, except main diagonal
465     for (int i = 1; i < R.colsize(); i++) {
466         for (int j = 0; j < i; j++) {
467             T sum_nom(0.0);
468             T sum_l2nom(0.0);
469             for (int k = 0; k < Q.rowsize(); k++) {
470                 sum_nom += A(k, p[i]) * Q(k, j);
471                 sum_l2nom += Q(k, j) * Q(k, j);
472             }
473             sum_l2nom = sqrt(sum_l2nom);
474             // add element
475             R(j, i) = sum_nom / sum_l2nom;
476         }
477     }
478 
479     // add missing values and scale
480     for (int j = 0; j < Q.colsize(); j++) {
481         // compute norm of column j
482         T sum(0.0);
483         for (int i = 0; i < Q.rowsize(); i++) sum += Q(i, j) * Q(i, j);
484         sum = sqrt(sum);
485         // add main diagonal to R
486         R(j, j) = sum;
487         // scale Q
488         for (int i = 0; i < Q.rowsize(); i++) Q(i, j) = Q(i, j) / sum;
489     }
490 
491     return R;
492 }
493 
524 template<typename T>
525 void permute_forward(DenseMatrix<T>& A, Vector<int>& p) {
526     // check if permutation vector has the right size
527     if (p.size() != A.colsize()) {
528         HDNUM_ERROR("Permutation Vector incompatible with Matrix!");
529     }
530 
531     // permutate the columns
532     for (int k = 0; k < p.size(); k++) {
533         if (p[k] != k) {
534             // swap column
535             for (int j=0; j < A.rowsize(); j++) {
536                 T temp_A = A(j, k);
537                 A(j, k) = A(j, p[k]);
538                 A(j, p[k]) = temp_A;
539             }
540             // swap inside permutation vector
541             int temp_p = p[k];
```

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

5.18 src/qrhousholder.hh File Reference

This file implements QR decomposition using housholder transformation.

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

Functions

- template<class REAL >
  DenseMatrix< REAL > **hdnum::creat_I_matrix** (size_t n)
- template<typename REAL >
  size_t **hdnum::sgn** (REAL val)

  Function that return the sign of a number.
- template<class REAL >
  void hdnum::qrhousholder (DenseMatrix< REAL > &A, hdnum::Vector< REAL > &v)

  Funktion that calculate the QR decoposition in place 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<class REAL >
  DenseMatrix< REAL > hdnum::qrhousholderexplizitQ (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.

5.18.1 Detailed Description

This file implements QR decomposition using housholder transformation.

5.18.2 Function Documentation

5.18.2.1 qrhousholder()

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

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

Template Parameters

A	the Matrix
v	oa vector of hdnum::Vector

5.18.2.2 qrhousholderexplizitQ()

```
template<class REAL >
DenseMatrix< REAL > hdnum::qrhousholderexplizitQ (
             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

A	the Matrix
v	oa vector of hdnum::Vector

Returns

Q matrix

5.19 qrhousholder.hh

Go to the documentation of this file.

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

```
36 template <typename REAL>
37 size_t sgn(REAL val) {
38     return (REAL(0) < val) - (val < REAL(0));
39 }
49 template <class REAL>
50 void qrhousholder(DenseMatrix<REAL>& A, hdnum::Vector<REAL>& v) {
51     auto m = A.rowsize();
52     auto n = A.colsize();
53     for (size_t j = 0; j < n; j++) {
54         REAL s = 0;
55         for (size_t i = j; i < m; i++) {
56             s = s + pow(A(i, j), 2);
57         }
58         s = sqrt(s);
59         v[j] = (-1.0) * sgn(A(j, j)) * s;
60         REAL fak = sqrt(s * (s + std::abs(A(j, j))));
61         A(j, j) = A(j, j) - v[j];
62         for (size_t k = j; k < m; k++) {
63             A(k, j) = A(k, j) / fak;
64         }
65         for (size_t i = j + 1; i < n; i++) {
66             s = 0;
67             for (size_t k = j; k < m; k++) {
68                 s = s + (A(k, j) * A(k, i));
69             }
70             for (size_t k = j; k < m; k++) {
71                 A(k, i) = A(k, i) - (A(k, j) * s);
72             }
73         }
74         // normalize the vi vectors again
75         for (size_t i = m; i >= 0; i--) {
76             A(i, j) = A(i, j) * fak;
77             if (i == j) {
78                 break;
79             }
80         }
81     }
82 }
93 template <class REAL>
94 DenseMatrix<REAL> qrhousholderexplizitQ(DenseMatrix<REAL>& A,
95                                         hdnum::Vector<REAL>& v,
96                                         bool show_Hi = false) {
97     auto m = A.rowsize();
98     auto n = A.colsize();
99     auto I = creat_I_matrix<REAL>(std::max(m, n));
100
101     DenseMatrix<REAL> Q(m, m, 0);
102     for (size_t j = 0; j < n; j++) {
103         REAL s = 0;
104         for (size_t i = j; i < m; i++) {
105             s = s + pow(A(i, j), 2);
106         }
107         s = sqrt(s);
108         v[j] = (-1.0) * sgn(A(j, j)) * s;
109         REAL fak = sqrt(s * (s + std::abs(A(j, j))));
110         A(j, j) = A(j, j) - v[j];
111         for (size_t k = j; k < m; k++) {
112             A(k, j) = A(k, j) / fak;
113         }
114         for (size_t i = j + 1; i < n; i++) {
115             s = 0;
116             for (size_t k = j; k < m; k++) {
117                 s = s + (A(k, j) * A(k, i));
118             }
119             for (size_t k = j; k < m; k++) {
120                 A(k, i) = A(k, i) - (A(k, j) * s);
121             }
122         }
123         // normalize the vi vectors again
124         for (size_t i = m; i >= 0; i--) {
125             A(i, j) = A(i, j) * fak;
126             if (i == j) {
127                 break;
128             }
129         }
130     }
131     // create qi and multiply them
132     if (m >= n) {
133         for (size_t j = 0; j < n; j++) {
134             DenseMatrix<REAL> TempQ(m, m, 0.0);
135             DenseMatrix<REAL> v1(m, 1, 0.0);
136             DenseMatrix<REAL> v1t(1, m, 0.0);
137             hdnum::Vector<double> v__i(m, 0);
138             for (size_t i = 0; i < m; i++) {
139                 if (i < j) {
140                     v1(i, 0) = 0;
141
```

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

5.20 src/rungekutta.hh File Reference

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

Classes

- class hdnum::ImplicitRungeKuttaStepProblem< M >

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

- class hdnum::RungeKutta< M, S >

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

Functions

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

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

5.20.1 Detailed Description

@general Runge-Kutta solver

5.20.2 Function Documentation

5.20.2.1 ordertest()

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

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

Template Parameters

M	Type of model
S	Type of ODE solver

Parameters

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

5.21 rungekutta.hh

Go to the documentation of this file.

```
1 // -*- tab-width: 4; indent-tabs-mode: nil -*-
2 #ifndef HDNUM_RUNGEKUTTA_HH
3 #define HDNUM_RUNGEKUTTA_HH
4
5 #include "vector.hh"
6 #include "newton.hh"
7
12 namespace hdnum {
15   template<class M>
16   class ImplicitRungeKuttaStepProblem
17   {
18   public:
20     typedef typename M::size_type size_type;
21
23     typedef typename M::time_type time_type;
24
26     typedef typename M::number_type number_type;
27
29     ImplicitRungeKuttaStepProblem (const M& model_,
30                                    DenseMatrix<number_type> A_,
31                                    Vector<number_type> b_,
32                                    Vector<number_type> c_,
33                                    time_type t_,
34                                    Vector<number_type> u_,
35                                    time_type dt_)
36         : model(model_) , u(model.size())
37       {
38         A = A_;
39         b = b_;
40         c = c_;
41         s = A_.rowsize ();
42         dt = dt_;
43         n = model.size();
44         t = t_;
45         u = u_;
46       }
47
49     std::size_t size () const
50     {
51       return n*s;
52     }
53
55     void F (const Vector<number_type>& x, Vector<number_type>& result) const
56     {
57       Vector<Vector<number_type> > xx (s);
58       for (int i = 0; i < s; i++)
59       {
60         xx[i].resize(n,number_type(0));
61         for(int k = 0; k < n; k++)
62         {
63           xx[i][k] = x[i*n + k];
64         }
65       }
66       Vector<Vector<number_type> > f (s);
67       for (int i = 0; i < s; i++)
68       {
69         f[i].resize(n, number_type(0));
70         model.f(t + c[i] * dt, u + xx[i], f[i]);
71       }
72       Vector<Vector<number_type> > hr (s);
73       for (int i = 0; i < s; i++)
74       {
75         hr[i].resize(n, number_type(0));
76       }
77       for (int i = 0; i < s; i++)
78       {
79         Vector<number_type> sum (n, number_type(0));
80         for (int j = 0; j < s; j++)
81         {
82           sum.update(dt*A[i][j], f[j]);
83         }
84         hr[i]  = xx[i] - sum;
85       }
86       //translating hr into result
87       for (int i = 0; i < s; i++)
88       {
89         for (int j = 0; j < n; j++)
90         {
91           result[i*n + j] = hr[i][j];
92         }
93       }
94     }
```

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

```
196   }
197
199   RungeKutta (const M& model_,
200               DenseMatrix<number_type> A_,
201               Vector<number_type> b_,
202               Vector<number_type> c_,
203               number_type sigma_)
204     : model(model_), u(model.size()), w(model.size()), K(A_.rowsize ())
205   {
206     A = A_;
207     b = b_;
208     c = c_;
209     s = A_.rowsize ();
210     n = model.size();
211     model.initialize(t,u);
212     dt = 0.1;
213     for (int i = 0; i < s; i++)
214     {
215       K[i].resize(n, number_type(0));
216     }
217     sigma = sigma_;
218     verbosity = 0;
219     if (A_.rowsize()!=A_.colsize())
220     HDNUM_ERROR("need square and nonempty matrix");
221     if (A_.rowsize()!=b_.size())
222     HDNUM_ERROR("vector incompatible with matrix");
223     if (A_.colsize()!=c_.size())
224     HDNUM_ERROR("vector incompatible with matrix");
225   }
226
228   void set_dt (time_type dt_)
229   {
230     dt = dt_;
231   }
232
234   bool check_explicit ()
235   {
236     bool is_explicit = true;
237     for (int i = 0; i < s; i++)
238     {
239       for (int j = i; j < s; j++)
240       {
241         if (A[i][j] != 0.0)
242         {
243           is_explicit = false;
244         }
245       }
246     }
247     return is_explicit;
248   }
249
251   void step ()
252   {
253     if (check_explicit())
254     {
255       // compute k_1
256       w = u;
257       model.f(t, w, K[0]);
258       for (int i = 0; i < s; i++)
259       {
260         Vector<number_type> sum (K[0].size(), 0.0);
261         sum.update(b[0], K[0]);
262         //compute k_i
263         for (int j = 0; j < i+1; j++)
264         {
265           sum.update(A[i][j],K[j]);
266         }
267         Vector<number_type> wert = w.update(dt,sum);
268         model.f(t + c[i]*dt, wert, K[i]);
269         u.update(dt *b[i], K[i]);
270       }
271     }
272     if (not check_explicit())
273     {
274       // In the implicit case we need to solve a nonlinear problem
275       // to do a time step.
276       ImplicitRungeKuttaStepProblem<M> problem(model, A, b, c, t, u, dt);
277       bool last_row_eq_b = true;
278       for (int i = 0; i<s; i++)
279       {
280         if (A[s-1][i] != b[i])
281         {
282           last_row_eq_b = false;
283         }
284       }
285
286       // Solve nonlinear problem and determine coefficients
```

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

```
376
378   time_type get_time () const
379   {
380     return t;
381   }
382
384   time_type get_dt () const
385   {
386     return dt;
387   }
388
390   void set_verbosity(int verbosity_)
391   {
392     verbosity = verbosity_;
393   }
394
395  private:
396    const M& model;
397    time_type t, dt;
398    Vector<number_type> u;
399    Vector<number_type> w;
400    Vector<Vector<number_type> > K;                    // save ki
401    int n;                                                                    //
     dimension of matrix A
402    int s;
403    DenseMatrix<number_type> A;                                  // A, b, c as in the butcher tableau
404        Vector<number_type> b;
405        Vector<number_type> c;
406    number_type sigma;
407    int verbosity;
408  };
409
410
422  template<class M, class S>
423  void ordertest(const M& model,
424                 S solver,
425                 typename M::number_type T,
426                 typename M::number_type h_0,
427                 int l)
428  {
429    // Get types
430    typedef typename M::time_type time_type;
431    typedef typename M::number_type number_type;
432
433    // error_array[i] = ||u(T)-u_i(T)||
434    number_type error_array[l];
435
436    Vector<number_type> exact_solution;
437    model.exact_solution(T, exact_solution);
438
439    for (int i=0; i<l; i++)
440    {
441      // Set initial time and value
442      time_type t_start;
443      Vector<number_type> initial_solution(1);
444      model.initialize(t_start, initial_solution);
445      solver.set_state(t_start, initial_solution);
446
447      // Initial time step
448      time_type dt = h_0/pow(2,i) ;
449      solver.set_dt(dt);
450
451      // Time loop
452      while (solver.get_time()<T-2*solver.get_dt())
453      {
454        solver.step();
455      }
456
457      // Last steps
458      if (solver.get_time()<T-solver.get_dt())
459      {
460        solver.set_dt((T-solver.get_time())/2.0);
461        for(int i=0; i<2; i++)
462        {
463          solver.step();
464        }
465      }
466      else
467      {
468        solver.set_dt(T-solver.get_time());
469        solver.step();
470      }
471
472      // Error
473      Vector<number_type> state = solver.get_state();
474      error_array[i] = norm(exact_solution-state);
475
```

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

5.22 sgrid.hh

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

```
63        return w;
64      }
65
66      inline Vector<number_type> index2world(size_type index) const
67      {
68        Vector<number_type> w(dim);
69        Vector<size_type> c = index2grid(index);
70        return grid2world(c);
71      }
72
73
74    public:
75
77      const size_type invalid_node;
78
93      SGrid(const Vector<number_type> extent_,
94            const Vector<size_type> size_,
95            const DomainFunction & df_)
96        : extent(extent_), size(size_), df(df_),
97          h(dim), offsets(dim),
98          invalid_node(std::numeric_limits<size_type>::max())
99      {
100       // Determine total number of nodes, increment offsets, and cell
101       // widths.
102       n_nodes = 1;
103       offsets.resize(dim);
104       h.resize(dim);
105       for(int d=0; d<dim; ++d){
106         n_nodes *= size[d];
107         offsets[d] = d==0 ? 1 : size[d-1] * offsets[d-1];
108         h[d] = extent[d] / number_type(size[d]-1);
109       }
110
111       // Initialize maps.
112       node_map.resize(0);
113       inside_map.resize(n_nodes);
114       grid_map.resize(n_nodes);
115       boundary_map.resize(0);
116       boundary_map.resize(n_nodes,false);
117
118       for(size_type n=0; n<n_nodes; ++n){
119         Vector<size_type> c = index2grid(n);
120         Vector<number_type> x = grid2world(c);
121
122         inside_map[n] = df.evaluate(x);
123         if(inside_map[n]){
124           node_map.push_back(n);
125           grid_map[n] = node_map.size()-1;
126         }
127         else
128           grid_map[n] = invalid_node;
129       }
130
131       // Find boundary nodes
132       for(size_type n=0; n<node_map.size(); ++n){
133         for(int d=0; d<dim; ++d){
134           for(int s=0; s<2; ++s){
135             const int side = s*2-1;
136             const size_type neighbor = getNeighborIndex(n,d,side,1);
137             if(neighbor == invalid_node)
138               boundary_map[node_map[n]] = true;
139           }
140         }
141       }
142
143     }
144
164     size_type getNeighborIndex(const size_type ln, const size_type n_dim, const int n_side, const int k
      = 1) const
165     {
166       const size_type n = node_map[ln];
167       const Vector<size_type> c = index2grid(n);
168       size_type neighbors[2];
169       neighbors[0] = c[n_dim];
170       neighbors[1] = size[n_dim]-c[n_dim]-1;
171
172       assert(n_side == 1 || n_side == -1);
173       if(size_type(k) > neighbors[(n_side+1)/2])
174         return invalid_node;
175
176       const size_type neighbor = n + offsets[n_dim] * n_side * k;
177
178       if(!inside_map[neighbor])
179         return invalid_node;
180
181       return grid_map[neighbor];
182     }
```

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

5.23 sparsematrix.hh

```
1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 /*
3  * File:   sparsematrix.hh
4  * Author: Christian Heusel <christian@heusel.eu>
5  *
6  * Created on August 25, 2020
7  */
8
9 #ifndef SPARSEMATRIX_HH
10 #define SPARSEMATRIX_HH
11
12 #include <algorithm>
13 #include <complex>
14 #include <functional>
15 #include <iomanip>
16 #include <iostream>
17 #include <map>
18 #include <numeric>
19 #include <string>
20 #include <type_traits>
21 #include <vector>
22
23 #include "densematrix.hh"
24 #include "vector.hh"
25
26 namespace hdnum {
27
30 template <typename REAL>
31 class SparseMatrix {
32 public:
34     using size_type = std::size_t;
35
37     using column_iterator = typename std::vector<REAL>::iterator;
39     using const_column_iterator = typename std::vector<REAL>::const_iterator;
40
41 private:
42     // Matrix data is stored in an STL vector!
43     std::vector<REAL> _data;
44
```

```
45      // The non-null indices are stored in STL vectors with the size_type!
46      // Explanation on how the mapping works can be found here:
47      // https://de.wikipedia.org/wiki/Compressed_Row_Storage
48      std::vector<size_type> _colIndices;
49      std::vector<size_type> _rowPtr;
50
51      size_type m_rows = 0;  // Number of Matrix rows
52      size_type m_cols = 0;  // Number of Matrix columns
53
54      static bool bScientific;
55      static size_type nIndexWidth;
56      static size_type nValueWidth;
57      static size_type nValuePrecision;
58      static const REAL _zero;
59
60      // !function that converts container contents into
61      // { 1, 2, 3, 4 }
62      template <typename T>
63      [[nodiscard]] std::string comma_fold(T container) const {
64          return "{ " +
65                 std::accumulate(
66                     std::next(container.cbegin()), container.cend(),
67                     std::to_string(container[0]),  // start with first element
68                     [](const std::string &a, REAL b) {
69                         return a + ", " + std::to_string(b);
70                     }) +
71                 " }";
72      }
73
74      // This code was copied from StackOverflow to gerneralize a check whether a
75      // template is a specialization i.e. for std::complex
76      // https://stackoverflow.com/questions/31762958/check-if-class-is-a-template-specialization
77      template <class T, template <class...> class Template>
78      struct is_specialization : std::false_type {};
79
80      template <template <class...> class Template, class... Args>
81      struct is_specialization<Template<Args...>, Template> : std::true_type {};
82
83      bool checkIfAccessIsInBounds(const size_type row_index,
84                                   const size_type col_index) const {
85          if (not (row_index < m_rows)) {
86              HDNUM_ERROR("Out of bounds access: row too big! -> " +
87                          std::to_string(row_index) + " is not < " +
88                          std::to_string(m_rows));
89              return false;
90          }
91          if (not (col_index < m_cols)) {
92              HDNUM_ERROR("Out of bounds access: column too big! -> " +
93                          std::to_string(col_index) + " is not < " +
94                          std::to_string(m_cols));
95              return false;
96          }
97          return true;
98      }
99
100 public:
116     SparseMatrix() = default;
117
119     SparseMatrix(const size_type _rows, const size_type _cols)
120         : _rowPtr(_rows + 1), m_rows(_rows), m_cols(_cols) {}
121
138     [[nodiscard]] size_type rowsize() const { return m_rows; }
139
155     [[nodiscard]] size_type colsize() const { return m_cols; }
156
158     [[nodiscard]] bool scientific() const { return bScientific; }
159
160     class column_index_iterator {
161     public:
162         using self_type = column_index_iterator;
163
164         // conform to the iterator traits
165         // https://en.cppreference.com/w/cpp/iterator/iterator_traits
166         using difference_type = std::ptrdiff_t;
167         using value_type = std::pair<REAL &, size_type const &>;
168         using pointer = value_type *;
169         using reference = value_type &;
170         using iterator_category = std::bidirectional_iterator_tag;
171
172         column_index_iterator(typename std::vector<REAL>::iterator valIter,
173                               std::vector<size_type>::iterator colIndicesIter)
174             : _valIter(valIter), _colIndicesIter(colIndicesIter) {}
175
176         // prefix
177         self_type &operator++() {
178             _valIter++;
179             _colIndicesIter++;
```

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

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

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

```
441             self_type cached = *this;
442             _rowPtrIter++;
443             return cached;
444         }
445
446         self_type &operator+=(difference_type offset) {
447             _rowPtrIter += offset;
448             return *this;
449         }
450
451         self_type &operator-=(difference_type offset) {
452             _rowPtrIter -= offset;
453             return *this;
454         }
455
456         // iter - n
457         self_type operator-(difference_type offset) {
458             self_type cache(*this);
459             cache -= offset;
460             return cache;
461         }
462
463         // iter + n
464         self_type operator+(difference_type offset) {
465             self_type cache(*this);
466             cache += offset;
467             return cache;
468         }
469         // n + iter
470         friend self_type operator+(const difference_type &offset,
471                                    const self_type &sec) {
472             self_type cache(sec);
473             cache += offset;
474             return cache;
475         }
476
477         reference operator[](difference_type offset) {
478             return *(*this + offset);
479         }
480
481         bool operator<(const self_type &other) {
482             return other - (*this) > 0;  //
483         }
484
485         bool operator>(const self_type &other) {
486             return other < (*this);  //
487         }
488
489         [[nodiscard]] self_type &operator*() { return *this; }
490         // [[nodiscard]] self_type &operator->() { return this; }
491
492         [[nodiscard]] bool operator==(const self_type &rhs) {
493             return _rowPtrIter == rhs._rowPtrIter;
494         }
495         [[nodiscard]] bool operator!=(const self_type &rhs) {
496             return _rowPtrIter != rhs._rowPtrIter;
497         }
498
499     private:
500         std::vector<size_type>::const_iterator _rowPtrIter;
501         std::vector<size_type>::const_iterator _colIndicesIter;
502         typename std::vector<REAL>::const_iterator _valIter;
503     };
504
523     [[nodiscard]] row_iterator begin() {
524         return row_iterator(_rowPtr.begin(), _colIndices.begin(),
525                             _data.begin());
526     }
527
546     [[nodiscard]] row_iterator end() {
547         return row_iterator(_rowPtr.end() - 1, _colIndices.begin(),
548                             _data.begin());
549     }
550
556     [[nodiscard]] const_row_iterator cbegin() const {
557         return const_row_iterator(_rowPtr.cbegin(), _colIndices.cbegin(),
558                                   _data.cbegin());
559     }
560
566     [[nodiscard]] const_row_iterator cend() const {
567         return const_row_iterator(_rowPtr.cend() - 1, _colIndices.cbegin(),
568                                   _data.cbegin());
569     }
570
572     [[nodiscard]] const_row_iterator begin() const { return this->cbegin(); }
574     [[nodiscard]] const_row_iterator end() const { return this->cend(); }
575
```

```
600     void scientific(bool b) const { bScientific = b; }
601
603     size_type iwidth() const { return nIndexWidth; }
604
606     size_type width() const { return nValueWidth; }
607
609     size_type precision() const { return nValuePrecision; }
610
612     void iwidth(size_type i) const { nIndexWidth = i; }
613
615     void width(size_type i) const { nValueWidth = i; }
616
618     void precision(size_type i) const { nValuePrecision = i; }
619
620     column_iterator find(const size_type row_index,
621                          const size_type col_index) const {
622         checkIfAccessIsInBounds(row_index, col_index);
623
624         using value_pair = typename const_column_index_iterator::value_type;
625         auto row = const_row_iterator(_rowPtr.begin() + row_index,
626                                       _colIndices.begin(), _data.begin());
627         return std::find_if(row.ibegin(), row.iend(),
628                             [col_index](value_pair el) {
629                                 // only care for the index here since the value
630                                 // is unknown
631                                 return el.second == col_index;
632                             });
633     }
634
635     bool exists(const size_type row_index, const size_type col_index) const {
636         auto row = const_row_iterator(_rowPtr.begin() + row_index,
637                                       _colIndices.begin(), _data.begin());
638         return find(row_index, col_index) != row.iend();
639     }
640
642     REAL &get(const size_type row_index, const size_type col_index) {
643         checkIfAccessIsInBounds(row_index, col_index);
644         // look for the entry
645         using value_pair = typename const_column_index_iterator::value_type;
646         auto row = row_iterator(_rowPtr.begin() + row_index,
647                                 _colIndices.begin(), _data.begin());
648         auto result =
649             std::find_if(row.ibegin(), row.iend(), [col_index](value_pair el) {
650                 // only care for the index here
651                 // since the value is unknown
652                 // anyways
653                 return el.second == col_index;
654             });
655         // we found something within the right row
656         if (result != row.iend()) {
657             return result.value();
658         }
659         throw std::out_of_range(
660             "There is no non-zero element for these given indicies!");
661     }
662
664     const REAL &operator()(const size_type row_index,
665                            const size_type col_index) const {
666         checkIfAccessIsInBounds(row_index, col_index);
667
668         using value_pair = typename const_column_index_iterator::value_type;
669         auto row = const_row_iterator(_rowPtr.begin() + row_index,
670                                       _colIndices.begin(), _data.begin());
671         auto result =
672             std::find_if(row.ibegin(), row.iend(), [col_index](value_pair el) {
673                 // only care for the index here since the value is unknown
674                 return el.second == col_index;
675             });
676         // we found something within the right row
677         if (result != row.iend()) {
678             return result.value();
679         }
680         return _zero;
681     }
682
684     [[nodiscard]] bool operator==(const SparseMatrix &other) const {
685         return (_data == other._data) and              //
686                (_rowPtr == other._rowPtr) and          //
687                (_colIndices == other._colIndices) and  //
688                (m_cols == other.m_cols) and            //
689                (m_rows == other.m_rows);
690     }
691
693     [[nodiscard]] bool operator!=(const SparseMatrix &other) const {
694         return not (*this == other);
695     }
696
```

```
697     // delete all the invalid comparisons
698     bool operator<(const SparseMatrix &other) = delete;
699     bool operator>(const SparseMatrix &other) = delete;
700     bool operator<=(const SparseMatrix &other) = delete;
701     bool operator>=(const SparseMatrix &other) = delete;
702
703     SparseMatrix transpose() const {
704         // TODO: remove / find bug here!
705         SparseMatrix::builder builder(m_cols, m_rows);
706         SparseMatrix::size_type curr_row = 0;
707         for (auto &row : (*this)) {
708             for (auto it = row.ibegin(); it != row.iend(); it++) {
709                 builder.addEntry(it.index(), curr_row, it.value());
710             }
711             curr_row++;
712         }
713
714         return builder.build();
715     }
716
719     [[nodiscard]] SparseMatrix operator*=(const REAL scalar) {
720         // This could also be done out of order
721         std::transform(_data.cbegin(), _data.cend(), _data.begin(),
722                        [&](REAL value) { return value * scalar; });
723     }
724
727     [[nodiscard]] SparseMatrix operator/=(const REAL scalar) {
728         // This could also be done out of order
729         std::transform(_data.cbegin(), _data.cend(), _data.begin(),
730                        [&](REAL value) { return value / scalar; });
731     }
732
741     template <class V>
742     void mv(Vector<V> &result, const Vector<V> &x) const {
743         static_assert(std::is_convertible<V, REAL>::value,
744                       "The types in the Matrix vector multiplication cant be "
745                       "converted properly!");
746
747         if (result.size() != this->colsize()) {
748             HDNUM_ERROR(
749                 (std::string("The result vector has the wrong dimension! ") +
750                  "Vector dimension " + std::to_string(result.size()) +
751                  " != " + std::to_string(this->colsize()) + " colsize"));
752         }
753
754         if (x.size() != this->colsize()) {
755             HDNUM_ERROR(
756                 (std::string("The input vector has the wrong dimension! ") +
757                  "Vector dimension " + std::to_string(x.size()) +
758                  " != " + std::to_string(this->colsize()) + " colsize"));
759         }
760
761         size_type curr_row = 0;
762         for (auto row : (*this)) {
763             result[curr_row] = std::accumulate(
764                 row.ibegin(), row.iend(), V {}, [&](V result, auto el) -> V {
765                     return result + (x[el.second] * el.first);
766                 });
767             curr_row++;
768         }
769     }
770
778     [[nodiscard]] Vector<REAL> operator*(const Vector<REAL> &x) const {
779         hdnum::Vector<REAL> result(this->colsize(), 0);
780         this->mv(result, x);
781         return result;
782     }
783
792     template <class V>
793     void umv(Vector<V> &result, const Vector<V> &x) const {
794         static_assert(std::is_convertible<V, REAL>::value,
795                       "The types in the Matrix vector multiplication cant be "
796                       "converted properly!");
797
798         if (result.size() != this->colsize()) {
799             HDNUM_ERROR(
800                 (std::string("The result vector has the wrong dimension! ") +
801                  "Vector dimension " + std::to_string(result.size()) +
802                  " != " + std::to_string(this->colsize()) + " colsize"));
803         }
804
805         if (x.size() != this->colsize()) {
806             HDNUM_ERROR(
807                 (std::string("The input vector has the wrong dimension! ") +
808                  "Vector dimension " + std::to_string(result.size()) +
809                  " != " + std::to_string(this->colsize()) + " colsize"));
810         }
```

```
811
812         size_type curr_row {};
813         for (auto row : (*this)) {
814             result[curr_row] += std::accumulate(
815                 row.ibegin(), row.iend(), V {}, [&](V result, auto el) -> V {
816                     return result + (x[el.second] * el.first);
817                 });
818             curr_row++;
819         }
820     }
821
822 private:
823     template <typename norm_type>
824     norm_type norm_infty_impl() const {
825         norm_type norm {};
826         for (auto row : *this) {
827             norm_type rowsum =
828                 std::accumulate(row.begin(), row.end(), norm_type {},
829                                 [](norm_type res, REAL value) -> norm_type {
830                                     return res + std::abs(value);
831                                 });
832             if (norm < rowsum) {
833                 norm = rowsum;
834             }
835         }
836         return norm;
837     }
838
839 public:
847     auto norm_infty() const {
848         if constexpr (is_specialization<REAL, std::complex> {}) {
849             return norm_infty_impl<double>();
850         } else {
851             return norm_infty_impl<REAL>();
852         }
853     }
854
855     [[nodiscard]] std::string to_string() const noexcept {
856         return "values=" + comma_fold(_data) + "\n" +         //
857                "colInd=" + comma_fold(_colIndices) + "\n" +   //
858                "rowPtr=" + comma_fold(_rowPtr) + "\n";        //
859     }
860
861     void print() const noexcept { std::cout « *this; }
862
886     static SparseMatrix identity(const size_type dimN) {
887         auto builder = typename SparseMatrix<REAL>::builder(dimN, dimN);
888         for (typename SparseMatrix<REAL>::size_type i = 0; i < dimN; ++i) {
889             builder.addEntry(i, i, REAL {1});
890         }
891         return builder.build();
892     }
893
918     SparseMatrix<REAL> matchingIdentity() const { return identity(m_cols); }
919
920     class builder {
921         size_type m_rows {};  // Number of Matrix rows, 0 by default
922         size_type m_cols {};  // Number of Matrix columns, 0 by default
923         std::vector<std::map<size_type, REAL» _rows;
924
925     public:
926         builder(size_type new_m_rows, size_type new_m_cols)
927             : m_rows {new_m_rows}, m_cols {new_m_cols}, _rows {m_rows} {}
928
929         builder(const std::initializer_list<std::initializer_list<REAL» &v)
930             : m_rows {v.size()}, m_cols {v.begin()->size()}, _rows(m_rows) {
931             size_type i = 0;
932             for (auto &row : v) {
933                 size_type j = 0;
934                 for (const REAL &element : row) {
935                     addEntry(i, j, element);
936                     j++;
937                 }
938                 i++;
939             }
940         }
941
942         builder() = default;
943
944         std::pair<typename std::map<size_type, REAL>::iterator, bool> addEntry(
945             size_type i, size_type j, REAL value) {
946             return _rows.at(i).emplace(j, value);
947         }
948
949         std::pair<typename std::map<size_type, REAL>::iterator, bool> addEntry(
950             size_type i, size_type j) {
951             return addEntry(i, j, REAL {});
```

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

```
1039             if (A.scientific()) {
1040                 s « std::setw(A.width()) « std::scientific « std::showpoint
1041                   « std::setprecision(A.precision()) « A(i, j) « " ";
1042             } else {
1043                 s « std::setw(A.width()) « std::fixed « std::showpoint
1044                   « std::setprecision(A.precision()) « A(i, j) « " ";
1045             }
1046         }
1047         s « std::endl;
1048     }
1049     return s;
1050 }
1051
1053 template <typename REAL>
1054 inline void zero(SparseMatrix<REAL> &A) {
1055     A = SparseMatrix<REAL>(A.rowsize(), A.colsize());
1056 }
1057
1091 template <class REAL>
1092 inline void identity(SparseMatrix<REAL> &A) {
1093     if (A.rowsize() != A.colsize()) {
1094         HDNUM_ERROR("Will not overwrite A since Dimensions are not equal!");
1095     }
1096     A = SparseMatrix<REAL>::identity(A.colsize());
1097 }
1098
1099 template <typename REAL>
1100 inline void readMatrixFromFile(const std::string &filename,
1101                                SparseMatrix<REAL> &A) {
1102     // Format taken from here:
1103     // https://math.nist.gov/MatrixMarket/formats.html#coord
1104
1105     using size_type = typename SparseMatrix<REAL>::size_type;
1106     std::string buffer;
1107     std::ifstream fin(filename);
1108     size_type i = 0;
1109     size_type j = 0;
1110     size_type non_zeros = 0;
1111
1112     if (fin.is_open()) {
1113         // ignore all comments from the file (starting with %)
1114         while (fin.peek() == '%') fin.ignore(2048, '\n');
1115
1116         std::getline(fin, buffer);
1117         std::istringstream first_line(buffer);
1118         first_line » i » j » non_zeros;
1119
1120         auto builder = typename SparseMatrix<REAL>::builder(i, j);
1121
1122         while (std::getline(fin, buffer)) {
1123             std::istringstream iss(buffer);
1124
1125             REAL value {};
1126             iss » i » j » value;
1127             // i-1, j-1, because matrix market does not use zero based indexing
1128             builder.addEntry(i - 1, j - 1, value);
1129         }
1130         A = builder.build();
1131         fin.close();
1132     } else {
1133         HDNUM_ERROR(("Could not osspen file! \"" + filename + "\""));
1134     }
1135 }
1136
1137 }  // namespace hdnum
1138
1139 #endif  // SPARSEMATRIX_HH
```

5.24 src/timer.hh File Reference

A simple timing class.

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

Classes

- class hdnum::TimerError

  Exception thrown by the Timer class
- class hdnum::Timer

  A simple stop watch.

5.24.1 Detailed Description

A simple timing class.

5.25 timer.hh

Go to the documentation of this file.

```
1 #ifndef DUNE_TIMER_HH
2 #define DUNE_TIMER_HH
3
4 #ifndef TIMER_USE_STD_CLOCK
5 // headers for getrusage(2)
6 #include <sys/resource.h>
7 #endif
8
9 #include <ctime>
10
11 // headers for stderror(3)
12 #include <cstring>
13
14 // access to errno in C++
15 #include <cerrno>
16
17 #include "exceptions.hh"
18
19 namespace hdnum {
20
26   class TimerError : public SystemError {} ;
27
28
41 class Timer
42 {
43 public:
45         Timer ()
46         {
47           reset();
48         }
49
51         void reset()
52         {
53 #ifdef TIMER_USE_STD_CLOCK
54           cstart = std::clock();
55 #else
56           rusage ru;
57           if (getrusage(RUSAGE_SELF, &ru))
58             HDNUM_THROW(TimerError, strerror(errno));
59           cstart = ru.ru_utime;
60 #endif
61         }
62
64         double elapsed () const
65         {
66 #ifdef TIMER_USE_STD_CLOCK
67           return (std::clock()-cstart) / static_cast<double>(CLOCKS_PER_SEC);
68 #else
69           rusage ru;
70           if (getrusage(RUSAGE_SELF, &ru))
71             HDNUM_THROW(TimerError, strerror(errno));
72           return 1.0 * (ru.ru_utime.tv_sec - cstart.tv_sec) + (ru.ru_utime.tv_usec - cstart.tv_usec) /
       (1000.0 * 1000.0);
73 #endif
74         }
75
76 private:
77 #ifdef TIMER_USE_STD_CLOCK
78   std::clock_t cstart;
```

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

5.26 vector.hh

```
1 // -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
2 /*
3  * File:   vector.hh
4  * Author: ngo
5  *
6  * Created on April 14th, 2011
7  */
8
9 #ifndef _VECTOR_HH
10 #define _VECTOR_HH
11
12 #include <assert.h>
13
14 #include <cmath>
15 #include <cstdlib>
16 #include <fstream>
17 #include <iomanip>
18 #include <iostream>
19 #include <sstream>
20 #include <vector>
21
22 #include "exceptions.hh"
23
24 namespace hdnum {
25
29   template<typename REAL>
30   class Vector : public std::vector<REAL>  // inherit from the STL vector
31   {
32   public:
34     typedef std::size_t size_type;
35
36   private:
37     static bool bScientific;
38     static std::size_t nIndexWidth;
39     static std::size_t nValueWidth;
40     static std::size_t nValuePrecision;
41
42   public:
43
45     Vector() : std::vector<REAL>()
46     {
47     }
48
50     Vector( const size_t size,                 // user must specify the size
51             const REAL defaultvalue_ = 0    // if not specified, the value 0 will take effect
52             )
53       : std::vector<REAL>( size, defaultvalue_ )
54     {
55     }
56
58     Vector (const std::initializer_list<REAL> &v)
59     {
60       for (auto elem : v) this->push_back(elem);
61     }
62
63     // Methods:
64
86     Vector& operator=( const REAL value )
87     {
88       const size_t s = this->size();
89       Vector & self = *this;
90       for(size_t i=0; i<s; ++i)
91         self[i] = value;
92       return *this;
93     }
94
104     Vector sub (size_type i, size_type m)
105     {
106       Vector v(m);
107       Vector &self = *this;
108       size_type k=0;
109       for (size_type j=i; j<i+m; j++){
```

```
110         v[k]=self[j];
111         k++;
112       }
113       return v;
114     }
115
116
117
118 #ifdef DOXYGEN
150     Vector& operator=( const Vector& y )
151     {
152       // It is already implemented in the STL vector class itself!
153     }
154 #endif
155
156
157
159     Vector& operator*=( const REAL value )
160     {
161       Vector &self = *this;
162       for (size_t i = 0; i < this->size(); ++i)
163         self[i] *= value;
164       return *this;
165     }
166
167
169     Vector& operator/=( const REAL value )
170     {
171       Vector &self = *this;
172       for (size_t i = 0; i < this->size(); ++i)
173         self[i] /= value;
174       return *this;
175     }
176
177
179     Vector& operator+=( const Vector & y )
180     {
181       assert( this->size() == y.size());
182       Vector &self = *this;
183       for (size_t i = 0; i < this->size(); ++i)
184         self[i] += y[i];
185       return *this;
186     }
187
188
190     Vector& operator-=( const Vector & y )
191     {
192       assert( this->size() == y.size());
193       Vector &self = *this;
194       for (size_t i = 0; i < this->size(); ++i)
195         self[i] -= y[i];
196       return *this;
197     }
198
199
201     Vector & update(const REAL alpha, const Vector & y)
202     {
203       assert( this->size() == y.size());
204       Vector &self = *this;
205       for (size_t i = 0; i < this->size(); ++i)
206         self[i] += alpha * y[i];
207       return *this;
208     }
209
210
242     REAL operator*(Vector & x) const
243     {
244       assert( x.size() == this->size() );   // checks if the dimensions of the two vectors are equal
245       REAL sum( 0 );
246       const Vector & self = *this;
247       for( size_t i = 0; i < this->size(); ++i )
248         sum += self[i] * x[i];
249       return sum;
250     }
251
252
253
254
287     Vector operator+(Vector & x) const
288     {
289       assert( x.size() == this->size() );   // checks if the dimensions of the two vectors are equal
290       Vector sum( *this );
291       sum += x;
292       return sum;
293     }
294
295
```

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

```
495     for (size_t r = 0; r < x.size(); ++r)
496       {
497         if( x.scientific() )
498           {
499             os « "["
500                « std::setw(x.iwidth())
501                « r
502                « "]"
503                « std::scientific
504                « std::showpoint
505                « std::setw( x.width() )
506                « std::setprecision( x.precision() )
507               « x[r]
508                « std::endl;
509           }
510         else
511           {
512             os « "["
513                « std::setw(x.iwidth())
514                « r
515                « "]"
516                « std::fixed
517                « std::showpoint
518                « std::setw( x.width() )
519                « std::setprecision( x.precision() )
520                « x[r]
521                « std::endl;
522           }
523       }
524     return os;
525   }
526
527
528
551   template<typename REAL>
552   inline void gnuplot(
553                       const std::string& fname,
554                       const Vector<REAL> x
555                       )
556   {
557     std::fstream f(fname.c_str(),std::ios::out);
558     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
559       {
560         if( x.scientific() )
561           {
562             f « std::setw(x.width())
563               « i
564               « std::scientific
565               « std::showpoint
566               « std::setw( x.width() )
567               « std::setprecision( x.precision() )
568               « x[i]
569               « std::endl;
570           }
571         else
572           {
573             f « std::setw(x.width())
574               « i
575               « std::fixed
576               « std::showpoint
577               « std::setw( x.width() )
578               « std::setprecision( x.precision() )
579               « x[i]
580               « std::endl;
581           }
582       }
583     f.close();
584   }
585
586   template<typename REAL>
587   inline void gnuplot(
588                       const std::string& fname,
589                       const std::vector<std::string>& t,
590                       const Vector<REAL> x
591                       )
592   {
593     std::fstream f(fname.c_str(),std::ios::out);
594     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
595       {
596         if( x.scientific() )
597           {
598             f « t[i] « " "
599               « std::scientific
600               « std::showpoint
601               « std::setw( x.width() )
602               « std::setprecision( x.precision() )
603               « x[i]
```

```
604               « std::endl;
605           }
606         else
607           {
608             f « t[i] « " "
609               « std::fixed
610               « std::showpoint
611               « std::setw( x.width() )
612               « std::setprecision( x.precision() )
613               « x[i]
614               « std::endl;
615           }
616       }
617     f.close();
618   }
619
621   template<typename REAL>
622   inline void gnuplot(
623                       const std::string& fname,
624                       const Vector<REAL> x,
625                       const Vector<REAL> y
626                       )
627   {
628     std::fstream f(fname.c_str(),std::ios::out);
629     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
630       {
631         if( x.scientific() )
632           {
633             f « std::setw(x.width())
634               « i
635               « std::scientific
636               « std::showpoint
637               « std::setw( x.width() )
638               « std::setprecision( x.precision() )
639               « x[i]
640               « " "
641               « std::setw( x.width() )
642               « std::setprecision( x.precision() )
643               « y[i]
644               « std::endl;
645           }
646         else
647           {
648             f « std::setw(x.width())
649               « i
650               « std::fixed
651               « std::showpoint
652               « std::setw( x.width() )
653               « std::setprecision( x.precision() )
654               « x[i]
655               « " "
656               « std::setw( x.width() )
657               « std::setprecision( x.precision() )
658               « y[i]
659               « std::endl;
660           }
661       }
662
663     f.close();
664   }
665
666
667
696   template<typename REAL>
697   inline void readVectorFromFile (const std::string& filename, Vector<REAL> &vector)
698   {
699     std::string buffer;
700     std::ifstream fin( filename.c_str() );
701     if( fin.is_open() ){
702       while( fin ){
703         std::string sub;
704         fin » sub;
705         //std::cout « " sub = " « sub.c_str() « ": ";
706         if( sub.length()>0 ){
707           REAL a = atof(sub.c_str());
708           //std::cout « std::fixed « std::setw(10) « std::setprecision(5) « a;
709           vector.push_back(a);
710         }
711       }
712       fin.close();
713     }
714     else{
715       HDNUM_ERROR("Could not open file!");
716     }
717   }
718
719
```

```
721   template<class REAL>
722   inline void zero (Vector<REAL>& x)
723   {
724     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
725       x[i] = REAL(0);
726   }
727
729   template<class REAL>
730   inline REAL norm (Vector<REAL> x)
731   {
732     REAL sum(0.0);
733     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
734       sum += x[i]*x[i];
735     return sqrt(sum);
736   }
737
739   template<class REAL>
740   inline void fill (Vector<REAL>& x, const REAL t)
741   {
742     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
743       x[i] = t;
744   }
745
768   template<class REAL>
769   inline void fill (Vector<REAL>& x, const REAL& t, const REAL& dt)
770   {
771     REAL myt(t);
772     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
773       {
774         x[i] = myt;
775         myt += dt;
776       }
777   }
778
779
802   template<class REAL>
803   inline void unitvector (Vector<REAL> & x, std::size_t j)
804   {
805     for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
806       if (i==j)
807         x[i] = REAL(1);
808       else
809         x[i] = REAL(0);
810   }
811
812
813 } // end of namespace hdnum
814
815 #endif  /* _VECTOR_HH */
```

Index