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:

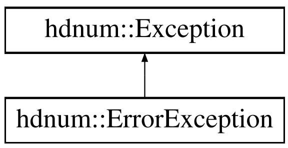
- src/ode.hh

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:

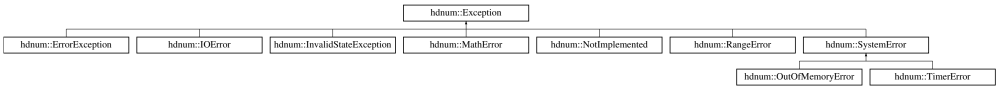
- src/exceptions.hh

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:

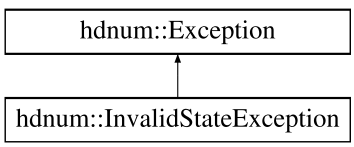
- src/rungekutta.hh

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:

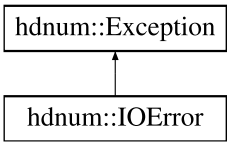
- src/exceptions.hh

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:

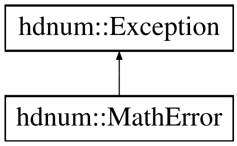
- src/ode.hh

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:

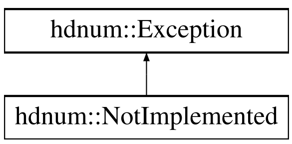
- src/newton.hh

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:

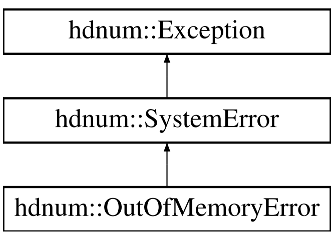
- src/opcounter.hh

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:

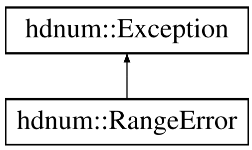
- src/exceptions.hh

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 ln, const size_type n_dim, const int n_side, const int k=1) const

  Provides the index of the k-th neighbor of the node with index ln.
- bool **isBoundaryNode** (const size_type ln) const

  Returns true if the node is on the boundary of the discrete 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 ln) const

  Returns the world coordinates of the node with the given node index.
- std::vector< Vector< number_type > > **getNodeCoordinates** () const

Public Attributes

- const size_type **invalid_node**

  The value which is returned to indicate an invalid node.

Static Public Attributes

- static const int **positive** = 1

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

4.32.1 Detailed Description

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

Structured Grid for Finite Differences.

Template Parameters

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:

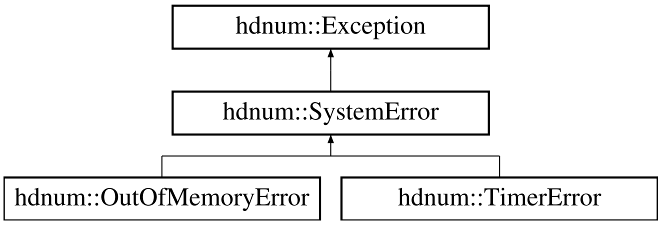
- src/pde.hh

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:

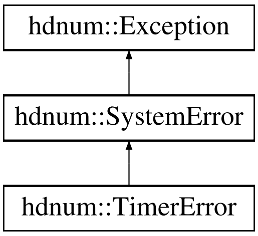
- src/timer.hh

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:

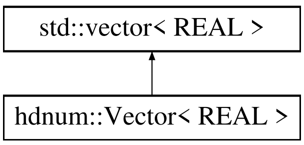
- src/timer.hh

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

```cpp
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
/*
 * File:   densematrix.hh
 * Author: ngo
 *
 * Created on April 15, 2011
 */

#ifndef DENSEMATRIX_HH
#define DENSEMATRIX_HH

#include <cstdlib>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>

#include "exceptions.hh"
#include "sparsematrix.hh"
#include "vector.hh"

namespace hdnum {

// forward-declare the sparse matrix template to make the transforming
// constructor from hdnum::SparseMatrix -> hdnum::DenseMatrix working
template <typename REAL>
class SparseMatrix;

template <typename REAL>
class DenseMatrix {
public:
    typedef std::size_t size_type;
    typedef typename std::vector<REAL> VType;
    typedef typename VType::const_iterator ConstVectorIterator;
    typedef typename VType::iterator VectorIterator;

private:
    VType m_data;         // Matrix data is stored in an STL vector!
    std::size_t m_rows;   // Number of Matrix rows
    std::size_t m_cols;   // Number of Matrix columns

    static bool bScientific;
    static std::size_t nIndexWidth;
    static std::size_t nValueWidth;
    static std::size_t nValuePrecision;

    REAL myabs(REAL x) const {
        if (x >= REAL(0))
            return x;
        else
            return -x;
    }

    inline REAL& at(const std::size_t row, const std::size_t col) {
        return m_data[row * m_cols + col];
    }

```

```
    inline const REAL& at(const std::size_t row, const std::size_t col) const {
        return m_data[row * m_cols + col];
    }

public:
    DenseMatrix() : m_data(0, 0), m_rows(0), m_cols(0) {}

    DenseMatrix(const std::size_t _rows, const std::size_t _cols,
                const REAL def_val = 0)
        : m_data(_rows * _cols, def_val), m_rows(_rows), m_cols(_cols) {}

    DenseMatrix(const std::initializer_list<std::initializer_list<REAL»& v) {
        m_rows = v.size();
        m_cols = v.begin()->size();
        for (auto row : v) {
            if (row.size() != m_cols) {
                std::cout « "Zeilen der Matrix nicht gleich lang" « std::endl;
                exit(1);
            }
            for (auto elem : row) m_data.push_back(elem);
        }
    }

    DenseMatrix(const hdnum::SparseMatrix<REAL>& other)
        : m_data(other.rowsize() * other.colsize()), m_rows(other.rowsize()),
          m_cols(other.colsize()) {
        using counter_type = typename hdnum::SparseMatrix<REAL>::size_type;
        counter_type row_index {};
        for (auto& row : other) {
            for (auto it = row.ibegin(); it != row.iend(); it++) {
                this->operator[](row_index)[it.index()] = it.value();
            }
            row_index++;
        }
    }

    void addNewRow(const hdnum::Vector<REAL>& rowvector) {
        m_rows++;
        m_cols = rowvector.size();
        for (std::size_t i = 0; i < m_cols; i++) m_data.push_back(rowvector[i]);
    }

    /*
    // copy constructor (not needed, since it inherits from the STL vector)
    DenseMatrix( const DenseMatrix& A )
    {
    this->m_data = A.m_data;
    m_rows = A.m_rows;
    m_cols = A.m_cols;
    }
    */

    size_t rowsize() const { return m_rows; }

    size_t colsize() const { return m_cols; }

    // pretty-print output properties
    bool scientific() const { return bScientific; }

    void scientific(bool b) const { bScientific = b; }

    std::size_t iwidth() const { return nIndexWidth; }

    std::size_t width() const { return nValueWidth; }

    std::size_t precision() const { return nValuePrecision; }

    void iwidth(std::size_t i) const { nIndexWidth = i; }

    void width(std::size_t i) const { nValueWidth = i; }

    void precision(std::size_t i) const { nValuePrecision = i; }

    // overloaded element access operators
    // write access on matrix element A_ij using A(i,j)
    inline REAL& operator()(const std::size_t row, const std::size_t col) {
        assert(row < m_rows || col < m_cols);
        return at(row, col);
    }

    inline const REAL& operator()(const std::size_t row,
                                  const std::size_t col) const {
        assert(row < m_rows || col < m_cols);
        return at(row, col);
    }

    const ConstVectorIterator operator[](const std::size_t row) const {
```

```
        assert(row < m_rows);
        return m_data.begin() + row * m_cols;
    }

    VectorIterator operator[](const std::size_t row) {
        assert(row < m_rows);
        return m_data.begin() + row * m_cols;
    }

    DenseMatrix& operator=(const DenseMatrix& A) {
        m_data = A.m_data;
        m_rows = A.m_rows;
        m_cols = A.m_cols;
        return *this;
    }

    DenseMatrix& operator=(const REAL value) {
        for (std::size_t i = 0; i < rowsize(); i++)
            for (std::size_t j = 0; j < colsize(); j++) (*this)(i, j) = value;
        return *this;
    }

    DenseMatrix sub(size_type i, size_type j, size_type rows, size_type cols) {
        DenseMatrix A(rows, cols);
        DenseMatrix& self = *this;
        for (size_type k1 = 0; k1 < rows; k1++) {
            for (size_type k2 = 0; k2 < cols; k2++) {
                A[k1][k2] = self[k1 + i][k2 + j];
            }
        }
        return A;
    }

    DenseMatrix transpose() const {
        DenseMatrix A(m_cols, m_rows);
        for (size_type i = 0; i < m_rows; i++) {
            for (size_type j = 0; j < m_cols; j++) {
                A[j][i] = this->operator[](i)[j];
            }
        }
        return A;
    }

    // Basic Matrix Operations

    DenseMatrix& operator+=(const DenseMatrix& B) {
        for (size_type i = 0; i < rowsize(); ++i) {
            for (size_type j = 0; j < colsize(); ++j) {
                (*this)(i, j) += B(i, j);
            }
        }
        return *this;
    }

    DenseMatrix& operator-=(const DenseMatrix& B) {
        for (std::size_t i = 0; i < rowsize(); ++i)
            for (std::size_t j = 0; j < colsize(); ++j)
                (*this)(i, j) -= B(i, j);
        return *this;
    }

    DenseMatrix& operator*=(const REAL s) {
        for (std::size_t i = 0; i < rowsize(); ++i)
            for (std::size_t j = 0; j < colsize(); ++j) (*this)(i, j) *= s;
        return *this;
    }

    DenseMatrix& operator/=(const REAL s) {
        for (std::size_t i = 0; i < rowsize(); ++i)
            for (std::size_t j = 0; j < colsize(); ++j) (*this)(i, j) /= s;
        return *this;
    }

    void update(const REAL s, const DenseMatrix& B) {
        for (std::size_t i = 0; i < rowsize(); ++i)
            for (std::size_t j = 0; j < colsize(); ++j)
                (*this)(i, j) += s * B(i, j);
    }

    template <class V>
    void mv(Vector<V>& y, const Vector<V>& x) const {
        if (this->rowsize() != y.size())
            HDNUM_ERROR("mv: size of A and y do not match");
        if (this->colsize() != x.size())
            HDNUM_ERROR("mv: size of A and x do not match");
        for (std::size_t i = 0; i < rowsize(); ++i) {
            y[i] = 0;
```

```
            for (std::size_t j = 0; j < colsize(); ++j)
                y[i] += (*this)(i, j) * x[j];
        }
    }

    template <class V>
    void umv(Vector<V>& y, const Vector<V>& x) const {
        if (this->rowsize() != y.size())
            HDNUM_ERROR("mv: size of A and y do not match");
        if (this->colsize() != x.size())
            HDNUM_ERROR("mv: size of A and x do not match");
        for (std::size_t i = 0; i < rowsize(); ++i) {
            for (std::size_t j = 0; j < colsize(); ++j)
                y[i] += (*this)(i, j) * x[j];
        }
    }

    template <class V>
    void umv(Vector<V>& y, const V& s, const Vector<V>& x) const {
        if (this->rowsize() != y.size())
            HDNUM_ERROR("mv: size of A and y do not match");
        if (this->colsize() != x.size())
            HDNUM_ERROR("mv: size of A and x do not match");
        for (std::size_t i = 0; i < rowsize(); ++i) {
            for (std::size_t j = 0; j < colsize(); ++j)
                y[i] += s * (*this)(i, j) * x[j];
        }
    }

    void mm(const DenseMatrix<REAL>& A, const DenseMatrix<REAL>& B) {
        if (this->rowsize() != A.rowsize())
            HDNUM_ERROR("mm: size incompatible");
        if (this->colsize() != B.colsize())
            HDNUM_ERROR("mm: size incompatible");
        if (A.colsize() != B.rowsize()) HDNUM_ERROR("mm: size incompatible");

        for (std::size_t i = 0; i < rowsize(); i++)
            for (std::size_t j = 0; j < colsize(); j++) {
                (*this)(i, j) = 0;
                for (std::size_t k = 0; k < A.colsize(); k++)
                    (*this)(i, j) += A(i, k) * B(k, j);
            }
    }

    void umm(const DenseMatrix<REAL>& A, const DenseMatrix<REAL>& B) {
        if (this->rowsize() != A.rowsize())
            HDNUM_ERROR("mm: size incompatible");
        if (this->colsize() != B.colsize())
            HDNUM_ERROR("mm: size incompatible");
        if (A.colsize() != B.rowsize()) HDNUM_ERROR("mm: size incompatible");

        for (std::size_t i = 0; i < rowsize(); i++)
            for (std::size_t j = 0; j < colsize(); j++)
                for (std::size_t k = 0; k < A.colsize(); k++)
                    (*this)(i, j) += A(i, k) * B(k, j);
    }

    void sc(const Vector<REAL>& x, std::size_t k) {
        if (this->rowsize() != x.size()) HDNUM_ERROR("cc: size incompatible");

        for (std::size_t i = 0; i < rowsize(); i++) (*this)(i, k) = x[i];
    }

    void sr(const Vector<REAL>& x, std::size_t k) {
        if (this->colsize() != x.size()) HDNUM_ERROR("cc: size incompatible");

        for (std::size_t i = 0; i < colsize(); i++) (*this)(k, i) = x[i];
    }

    REAL norm_infty() const {
        REAL norm(0.0);
        for (std::size_t i = 0; i < rowsize(); i++) {
            REAL sum(0.0);
            for (std::size_t j = 0; j < colsize(); j++)
                sum += myabs((*this)(i, j));
            if (sum > norm) norm = sum;
        }
        return norm;
    }

    REAL norm_1() const {
        REAL norm(0.0);
        for (std::size_t j = 0; j < colsize(); j++) {
            REAL sum(0.0);
            for (std::size_t i = 0; i < rowsize(); i++)
                sum += myabs((*this)(i, j));
            if (sum > norm) norm = sum;
```

```
        }
        return norm;
    }

    Vector<REAL> operator*(const Vector<REAL>& x) const {
        assert(x.size() == colsize());

        Vector<REAL> y(rowsize());
        for (std::size_t r = 0; r < rowsize(); ++r) {
            for (std::size_t c = 0; c < colsize(); ++c) {
                y[r] += at(r, c) * x[c];
            }
        }
        return y;
    }

    DenseMatrix operator*(const DenseMatrix& x) const {
        assert(colsize() == x.rowsize());

        const std::size_t out_rows = rowsize();
        const std::size_t out_cols = x.colsize();
        DenseMatrix y(out_rows, out_cols, 0.0);
        for (std::size_t r = 0; r < out_rows; ++r)
            for (std::size_t c = 0; c < out_cols; ++c)
                for (std::size_t i = 0; i < colsize(); ++i)
                    y(r, c) += at(r, i) * x(i, c);

        return y;
    }

    DenseMatrix operator+(const DenseMatrix& x) const {
        assert(colsize() == x.colsize());
        assert(rowsize() == x.rowsize());

        const std::size_t out_rows = rowsize();
        const std::size_t out_cols = x.colsize();
        DenseMatrix y(out_rows, out_cols, 0.0);
        y = *this;
        y += x;
        return y;
    }

    DenseMatrix operator-(const DenseMatrix& x) const {
        assert(colsize() == x.colsize());
        assert(rowsize() == x.rowsize());

        const std::size_t out_rows = rowsize();
        const std::size_t out_cols = x.colsize();
        DenseMatrix y(out_rows, out_cols, 0.0);
        y = *this;
        y -= x;
        return y;
    }
};

template <typename REAL>
bool DenseMatrix<REAL>::bScientific = true;
template <typename REAL>
std::size_t DenseMatrix<REAL>::nIndexWidth = 10;
template <typename REAL>
std::size_t DenseMatrix<REAL>::nValueWidth = 10;
template <typename REAL>
std::size_t DenseMatrix<REAL>::nValuePrecision = 3;

template <typename REAL>
inline std::ostream& operator«(std::ostream& s, const DenseMatrix<REAL>& A) {
    s « std::endl;
    s « " " « std::setw(A.iwidth()) « " "
      « "  ";
    for (typename DenseMatrix<REAL>::size_type j = 0; j < A.colsize(); ++j)
        s « std::setw(A.width()) « j « " ";
    s « std::endl;

    for (typename DenseMatrix<REAL>::size_type i = 0; i < A.rowsize(); ++i) {
        s « " " « std::setw(A.iwidth()) « i « "  ";
        for (typename DenseMatrix<REAL>::size_type j = 0; j < A.colsize();
             ++j) {
            if (A.scientific()) {
                s « std::setw(A.width()) « std::scientific « std::showpoint
                  « std::setprecision(A.precision()) « A[i][j] « " ";
            } else {
                s « std::setw(A.width()) « std::fixed « std::showpoint
                  « std::setprecision(A.precision()) « A[i][j] « " ";
            }
        }
        s « std::endl;
    }
```

```
    return s;
}

template <typename REAL>
inline void fill(DenseMatrix<REAL>& A, const REAL& t) {
    for (typename DenseMatrix<REAL>::size_type i = 0; i < A.rowsize(); ++i)
        for (typename DenseMatrix<REAL>::size_type j = 0; j < A.colsize(); ++j)
            A[i][j] = t;
}

template <typename REAL>
inline void zero(DenseMatrix<REAL>& A) {
    for (std::size_t i = 0; i < A.rowsize(); ++i)
        for (std::size_t j = 0; j < A.colsize(); ++j) A(i, j) = REAL(0);
}

template <class T>
inline void identity(DenseMatrix<T>& A) {
    for (typename DenseMatrix<T>::size_type i = 0; i < A.rowsize(); ++i)
        for (typename DenseMatrix<T>::size_type j = 0; j < A.colsize(); ++j)
            if (i == j)
                A[i][i] = T(1);
            else
                A[i][j] = T(0);
}

template <typename REAL>
inline void spd(DenseMatrix<REAL>& A) {
    if (A.rowsize() != A.colsize() || A.rowsize() == 0)
        HDNUM_ERROR("need square and nonempty matrix");
    for (std::size_t i = 0; i < A.rowsize(); ++i)
        for (std::size_t j = 0; j < A.colsize(); ++j)
            if (i == j)
                A(i, i) = REAL(4.0);
            else
                A(i, j) = -REAL(1.0) / ((i - j) * (i - j));
}

template <typename REAL>
inline void vandermonde(DenseMatrix<REAL>& A, const Vector<REAL> x) {
    if (A.rowsize() != A.colsize() || A.rowsize() == 0)
        HDNUM_ERROR("need square and nonempty matrix");
    if (A.rowsize() != x.size()) HDNUM_ERROR("need A and x of same size");
    for (typename DenseMatrix<REAL>::size_type i = 0; i < A.rowsize(); ++i) {
        REAL p(1.0);
        for (typename DenseMatrix<REAL>::size_type j = 0; j < A.colsize();
             ++j) {
            A[i][j] = p;
            p *= x[i];
        }
    }
}

template <typename REAL>
inline void gnuplot(const std::string& fname, const DenseMatrix<REAL>& A) {
    std::fstream f(fname.c_str(), std::ios::out);
    for (typename DenseMatrix<REAL>::size_type i = 0; i < A.rowsize(); ++i) {
        for (typename DenseMatrix<REAL>::size_type j = 0; j < A.colsize();
             ++j) {
            if (A.scientific()) {
                f « std::setw(A.width()) « std::scientific « std::showpoint
                  « std::setprecision(A.precision()) « A[i][j];
            } else {
                f « std::setw(A.width()) « std::fixed « std::showpoint
                  « std::setprecision(A.precision()) « A[i][j];
            }
        }
        f « std::endl;
    }
    f.close();
}

template <typename REAL>
inline void readMatrixFromFileDat(const std::string& filename,
                                  DenseMatrix<REAL>& A) {
    std::string buffer;
    std::ifstream fin(filename.c_str());
    std::size_t i = 0;
    std::size_t j = 0;
    if (fin.is_open()) {
        while (std::getline(fin, buffer)) {
            std::istringstream iss(buffer);
            hdnum::Vector<REAL> rowvector;
            while (iss) {
                std::string sub;
                iss » sub;
                // std::cout « " sub = " « sub.c_str() « ": ";
```

```
                if (sub.length() > 0) {
                    REAL a = atof(sub.c_str());
                    // std::cout « std::fixed « std::setw(10) «
                    // std::setprecision(5) « a;
                    rowvector.push_back(a);
                }
                j++;
            }
            if (rowvector.size() > 0) {
                A.addNewRow(rowvector);
                i++;
                // std::cout « std::endl;
            }
        }
        fin.close();
    } else {
        HDNUM_ERROR("Could not open file!");
    }
}

template <typename REAL>
inline void readMatrixFromFileMatrixMarket(const std::string& filename,
                                           DenseMatrix<REAL>& A) {
    std::string buffer;
    std::ifstream fin(filename.c_str());
    std::size_t i = 0;
    std::size_t j = 0;
    if (fin.is_open()) {
        // ignore all comments from the file (starting with %)
        while (fin.peek() == '%') fin.ignore(2048, '\n');

        std::getline(fin, buffer);
        std::istringstream first_line(buffer);
        first_line » i » j;
        DenseMatrix<REAL> A_temp(i, j);

        while (std::getline(fin, buffer)) {
            std::istringstream iss(buffer);

            REAL value {};
            iss » i » j » value;
            // i-1, j-1, because matrix market does not use zero based indexing
            A_temp(i - 1, j - 1) = value;
        }
        A = A_temp;
        fin.close();
    } else {
        HDNUM_ERROR("Could not open file! \"" + filename + "\"");
    }
}

}  // namespace hdnum

#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.
```
#ifndef HDNUM_EXCEPTIONS_HH
#define HDNUM_EXCEPTIONS_HH

#include <string>
#include <sstream>

namespace hdnum {

  class Exception {
  public:
        void message(const std::string &message);
        const std::string& what() const;
  private:
        std::string _message;
  };

  inline void Exception::message(const std::string &message)
  {
        _message = message;
  }

  inline const std::string& Exception::what() const
  {
        return _message;
  }

  inline std::ostream& operator«(std::ostream &stream, const Exception &e)
  {
        return stream « e.what();
  }

  // the "format" the exception-type gets printed.  __FILE__ and
  // __LINE__ are standard C-defines, the GNU cpp-infofile claims that
  // C99 defines __func__ as well. __FUNCTION__ is a GNU-extension
#ifdef HDNUM_DEVEL_MODE
# define THROWSPEC(E) #E « " [" « __func__ « ":" « __FILE__ « ":" « __LINE__ « "]: "
#else
# define THROWSPEC(E) #E « ": "
#endif

  // this is the magic: use the usual do { ... } while (0) trick, create
  // the full message via a string stream and throw the created object
#define HDNUM_THROW(E, m) do { E th__ex; std::ostringstream th__out;            \
        th__out « THROWSPEC(E) « m; th__ex.message(th__out.str()); throw th__ex; \
  } while (0)

  class IOError : public Exception {};

   class MathError : public Exception {};

   class RangeError : public Exception {};

   class NotImplemented : public Exception {};

   class SystemError : public Exception {};

   class OutOfMemoryError : public SystemError {};

```

```
  class InvalidStateException : public Exception {};

  class ErrorException : public Exception {};

  // throw ErrorException with message
#define HDNUM_ERROR(m) 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)

} // end namespace

#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.

```
// -*- tab-width: 4; indent-tabs-mode: nil -*-
#ifndef HDNUM_LR_HH
#define HDNUM_LR_HH

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

namespace hdnum {

  template<class T>
  void lr (DenseMatrix<T>& A, Vector<std::size_t>& p)
  {
    if (A.rowsize()!=A.colsize() || A.rowsize()==0)
      HDNUM_ERROR("need square and nonempty matrix");
    if (A.rowsize()!=p.size())
      HDNUM_ERROR("permutation vector incompatible with matrix");

    // transformation to upper triangular
    for (std::size_t k=0; k<A.rowsize()-1; ++k)
      {
        // find pivot element and exchange rows
        for (std::size_t r=k; r<A.rowsize(); ++r)
          if (A[r][k]!=0)
            {
              p[k] = r; // store permutation in step k
              if (r>k) // exchange complete row if r!=k
                for (std::size_t j=0; j<A.colsize(); ++j)
                  {
                    T temp(A[k][j]);
                    A[k][j] = A[r][j];
                    A[r][j] = temp;
                  }
              break;
            }
        if (A[k][k]==0) HDNUM_ERROR("matrix is singular");

        // modification
        for (std::size_t i=k+1; i<A.rowsize(); ++i)
          {
            T qik(A[i][k]/A[k][k]);
            A[i][k] = qik;
            for (std::size_t j=k+1; j<A.colsize(); ++j)
              A[i][j] -= qik * A[k][j];
          }
      }
  }

  template<class T>
  T abs (const T& t)
  {
    if (t<0.0)
      return -t;
    else
      return t;
  }

  template<class T>
  void lr_partialpivot (DenseMatrix<T>& A, Vector<std::size_t>& p)
  {
    if (A.rowsize()!=A.colsize() || A.rowsize()==0)
      HDNUM_ERROR("need square and nonempty matrix");
    if (A.rowsize()!=p.size())
      HDNUM_ERROR("permutation vector incompatible with matrix");

    // initialize permutation
    for (std::size_t k=0; k<A.rowsize(); ++k)
      p[k] = k;

    // transformation to upper triangular
    for (std::size_t k=0; k<A.rowsize()-1; ++k)
      {
        // find pivot element
```

```
     for (std::size_t r=k+1; r<A.rowsize(); ++r)
       if (abs(A[r][k])>abs(A[k][k]))
         p[k] = r; // store permutation in step k

     if (p[k]>k) // exchange complete row if r!=k
       for (std::size_t j=0; j<A.colsize(); ++j)
         {
           T temp(A[k][j]);
           A[k][j] = A[p[k]][j];
           A[p[k]][j] = temp;
         }

     if (A[k][k]==0) HDNUM_ERROR("matrix is singular");

     // modification
     for (std::size_t i=k+1; i<A.rowsize(); ++i)
       {
         T qik(A[i][k]/A[k][k]);
         A[i][k] = qik;
         for (std::size_t j=k+1; j<A.colsize(); ++j)
            A[i][j] -= qik * A[k][j];
        }
    }
}

template<class T>
void lr_fullpivot (DenseMatrix<T>& A, Vector<std::size_t>& p, Vector<std::size_t>& q)
{
  if (A.rowsize()!=A.colsize() || A.rowsize()==0)
    HDNUM_ERROR("need square and nonempty matrix");
  if (A.rowsize()!=p.size())
    HDNUM_ERROR("permutation vector incompatible with matrix");

  // initialize permutation
  for (std::size_t k=0; k<A.rowsize(); ++k)
    p[k] = q[k] = k;

  // transformation to upper triangular
  for (std::size_t k=0; k<A.rowsize()-1; ++k)
    {
      // find pivot element
      for (std::size_t r=k; r<A.rowsize(); ++r)
        for (std::size_t s=k; s<A.colsize(); ++s)
          if (abs(A[r][s])>abs(A[k][k]))
            {
              p[k] = r; // store permutation in step k
              q[k] = s;
            }

      if (p[k]>k) // exchange complete row if r!=k
        for (std::size_t j=0; j<A.colsize(); ++j)
          {
            T temp(A[k][j]);
            A[k][j] = A[p[k]][j];
            A[p[k]][j] = temp;
          }
      if (q[k]>k) // exchange complete column if s!=k
        for (std::size_t i=0; i<A.rowsize(); ++i)
          {
            T temp(A[i][k]);
            A[i][k] = A[i][q[k]];
            A[i][q[k]] = temp;
          }

      if (std::abs(A[k][k])==0) HDNUM_ERROR("matrix is singular");

      // modification
      for (std::size_t i=k+1; i<A.rowsize(); ++i)
        {
          T qik(A[i][k]/A[k][k]);
          A[i][k] = qik;
          for (std::size_t j=k+1; j<A.colsize(); ++j)
            A[i][j] -= qik * A[k][j];
        }
    }
}

template<class T>
void permute_forward (const Vector<std::size_t>& p, Vector<T>& b)
{
  if (b.size()!=p.size())
    HDNUM_ERROR("permutation vector incompatible with rhs");

  for (std::size_t k=0; k<b.size()-1; ++k)
    if (p[k]!=k) std::swap(b[k],b[p[k]]);
}

```

```
template<class T>
void permute_backward (const Vector<std::size_t>& q, Vector<T>& z)
{
  if (z.size()!=q.size())
    HDNUM_ERROR("permutation vector incompatible with z");

  for (int k=z.size()-2; k>=0; --k)
    if (q[k]!=std::size_t(k)) std::swap(z[k],z[q[k]]);
}

template<class T>
void row_equilibrate (DenseMatrix<T>& A, Vector<T>& s)
{
  if (A.rowsize()*A.colsize()==0)
    HDNUM_ERROR("need nonempty matrix");
  if (A.rowsize()!=s.size())
    HDNUM_ERROR("scaling vector incompatible with matrix");

  // equilibrate row sums
  for (std::size_t k=0; k<A.rowsize(); ++k)
    {
      s[k] = T(0.0);
      for (std::size_t j=0; j<A.colsize(); ++j)
        s[k] += abs(A[k][j]);
      if (std::abs(s[k])==0) HDNUM_ERROR("row sum is zero");
      for (std::size_t j=0; j<A.colsize(); ++j)
        A[k][j] /= s[k];
    }
}

template<class T>
void apply_equilibrate (Vector<T>& s, Vector<T>& b)
{
  if (s.size()!=b.size())
    HDNUM_ERROR("s and b incompatible");

  // equilibrate row sums
  for (std::size_t k=0; k<b.size(); ++k)
    b[k] /= s[k];
}

template<class T>
void solveL (const DenseMatrix<T>& A, Vector<T>& x, const Vector<T>& b)
{
  if (A.rowsize()!=A.colsize() || A.rowsize()==0)
    HDNUM_ERROR("need square and nonempty matrix");
  if (A.rowsize()!=b.size())
    HDNUM_ERROR("right hand side incompatible with matrix");

  for (std::size_t i=0; i<A.rowsize(); ++i)
    {
      T rhs(b[i]);
      for (std::size_t j=0; j<i; j++)
        rhs -= A[i][j] * x[j];
      x[i] = rhs;
    }
}

template<class T>
void solveR (const DenseMatrix<T>& A, Vector<T>& x, const Vector<T>& b)
{
  if (A.rowsize()!=A.colsize() || A.rowsize()==0)
    HDNUM_ERROR("need square and nonempty matrix");
  if (A.rowsize()!=b.size())
    HDNUM_ERROR("right hand side incompatible with matrix");

  for (int i=A.rowsize()-1; i>=0; --i)
    {
      T rhs(b[i]);
      for (std::size_t j=i+1; j<A.colsize(); j++)
        rhs -= A[i][j] * x[j];
      x[i] = rhs/A[i][i];
    }
}

template<class T>
void linsolve (DenseMatrix<T>& A, Vector<T>& x, Vector<T>& b)
{
  if (A.rowsize()!=A.colsize() || A.rowsize()==0)
    HDNUM_ERROR("need square and nonempty matrix");
  if (A.rowsize()!=b.size())
    HDNUM_ERROR("right hand side incompatible with matrix");

  Vector<T> s(x.size());
  Vector<std::size_t> p(x.size());
  Vector<std::size_t> q(x.size());
  row_equilibrate(A,s);
```

```
    lr_fullpivot(A,p,q);
    apply_equilibrate(s,b);
    permute_forward(p,b);
    solveL(A,b,b);
    solveR(A,x,b);
    permute_backward(q,x);
  }

}
#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.

```
// -*- tab-width: 4; indent-tabs-mode: nil -*-
#ifndef HDNUM_NEWTON_HH
#define HDNUM_NEWTON_HH

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

namespace hdnum {

  template<class N>
  class SquareRootProblem
  {
  public:
    typedef std::size_t size_type;

    typedef N number_type;

    SquareRootProblem (number_type a_)
      : a(a_)
    {}

    std::size_t size () const
    {
      return 1;
    }

    void F (const Vector<N>& x, Vector<N>& result) const
    {
      result[0] = x[0]*x[0] - a;
    }

    void F_x (const Vector<N>& x, DenseMatrix<N>& result) const
    {
      result[0][0] = number_type(2.0)*x[0];
    }

  private:
    number_type a;
  };


  template<typename Lambda, typename Vec>
  class GenericNonlinearProblem
  {
    Lambda lambda; // lambda defining the problem "lambda(x)=0"
    size_t s;
    typename Vec::value_type eps;

  public:
    typedef std::size_t size_type;

    typedef typename Vec::value_type number_type;

    GenericNonlinearProblem (const Lambda& l_, const Vec& x_, number_type eps_ = 1e-7)
      : lambda(l_), s(x_.size()), eps(eps_)
    {}

    std::size_t size () const
    {
      return s;
    }

    void F (const Vec& x, Vec& result) const
    {
      result = lambda(x);
    }

    void F_x (const Vec& x, DenseMatrix<number_type>& result) const
    {
```

```
        Vec Fx(x.size());
        F(x,Fx);
        Vec z(x);
         Vec Fz(x.size());

         // numerische Jacobimatrix
         for (int j=0; j<result.colsize(); ++j)
           {
             auto zj = z[j];
             auto dz = (1.0+abs(zj))*eps;
             z[j] += dz;
             F(z,Fz);
             for (int i=0; i<result.rowsize(); i++)
               result[i][j] = (Fz[i]-Fx[i])/dz;
             z[j] = zj;
           }
     }
 };

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

 class Newton
 {
   typedef std::size_t size_type;

 public:
   Newton ()
     : maxit(25), linesearchsteps(10), verbosity(0),
       reduction(1e-14), abslimit(1e-30), converged(false)
   {}

   void set_maxit (size_type n)
   {
     maxit = n;
   }

   void set_sigma (double sigma_)
   {

   }

   void set_linesearchsteps (size_type n)
   {
     linesearchsteps = n;
   }

   void set_verbosity (size_type n)
   {
     verbosity = n;
   }

   void set_abslimit (double l)
   {
     abslimit = l;
   }

   void set_reduction (double l)
   {
     reduction = l;
   }

   template<class M>
   void solve (const M& model, Vector<typename M::number_type> & x) const
   {
     typedef typename M::number_type N;
     // In complex case, we still need to use real valued numbers for residual norms etc.
     using Real = typename std::conditional<std::is_same<std::complex<double>, N>::value, double, N>::type;
     Vector<N> r(model.size());              // residual
     DenseMatrix<N> A(model.size(),model.size()); // Jacobian matrix
     Vector<N> y(model.size());              // temporary solution in line search
     Vector<N> z(model.size());              // solution of linear system
     Vector<N> s(model.size());              // scaling factors
     Vector<size_type> p(model.size());                // row permutations
     Vector<size_type> q(model.size());                // column permutations

     model.F(x,r);                                    // compute nonlinear residual
     Real R0(std::abs(norm(r)));                         // norm of initial residual
     Real R(R0);                               // current residual norm
     if (verbosity>=1)
       {
         std::cout « "Newton "
```

```
                    « "   norm=" « std::scientific « std::showpoint
                    « std::setprecision(4) « R0
                    « std::endl;
      }

    converged = false;
    for (size_type i=1; i<=maxit; i++)                // do Newton iterations
      {
        // check absolute size of residual
        if (R<=abslimit)
          {
            converged = true;
            return;
          }

        // solve Jacobian system for update
        model.F_x(x,A);                               // compute Jacobian matrix
        row_equilibrate(A,s);                         // equilibrate rows
        lr_fullpivot(A,p,q);                          // LR decomposition of A
        z = N(0.0);                                   // clear solution
        apply_equilibrate(s,r);                       // equilibration of right hand side
        permute_forward(p,r);                         // permutation of right hand side
        solveL(A,r,r);                                // forward substitution
        solveR(A,z,r);                                // backward substitution
        permute_backward(q,z);                        // backward permutation

        // line search
        Real lambda(1.0);                      // start with lambda=1
        for (size_type k=0; k<linesearchsteps; k++)
          {
            y = x;
            y.update(-lambda,z);                       // y = x+lambda*z
            model.F(y,r);                             // r = F(y)
            Real newR(std::abs(norm(r)));                // compute norm
            if (verbosity>=3)
              {
                std::cout « "    line search "  « std::setw(2) « k
                          « " lambda=" « std::scientific « std::showpoint
                          « std::setprecision(4) « lambda
                          « " norm=" « std::scientific « std::showpoint
                          « std::setprecision(4) « newR
                          « " red=" « std::scientific « std::showpoint
                          « std::setprecision(4) « newR/R
                          « std::endl;
              }
            if (newR<(1.0-0.25*lambda)*R)            // check convergence
              {
                if (verbosity>=2)
                  {
                    std::cout « "  step"  « std::setw(3) « i
                              « " norm=" « std::scientific « std::showpoint
                              « std::setprecision(4) « newR
                              « " red=" « std::scientific « std::showpoint
                              « std::setprecision(4) « newR/R
                              « std::endl;
                  }
                x = y;
                R = newR;
                break;                                // continue with Newton loop
              }
            else lambda *= 0.5;                       // reduce damping factor
            if (k==linesearchsteps-1)
              {
                if (verbosity>=3)
                  std::cout « "    line search not converged within " « linesearchsteps « " steps" « std::endl;
                return;
              }
          }

        // check convergence
        if (R<=reduction*R0)
          {
            if (verbosity>=1)
              {
                std::cout « "Newton converged in "  « i « " steps"
                          « " reduction=" « std::scientific « std::showpoint
                          « std::setprecision(4) « R/R0
                          « std::endl;
              }
            iterations_taken = i;
            converged = true;
            return;
          }
        if (i==maxit)
          {
            iterations_taken = i;
```

```
            if (verbosity>=1)
              std::cout « "Newton not converged within " « maxit « " iterations" « std::endl;
          }
      }
  }

  bool has_converged () const
  {
    return converged;
  }
  size_type iterations() const {
    return iterations_taken;
  }


private:
  size_type maxit;
  mutable size_type iterations_taken = -1;
  size_type linesearchsteps;
  size_type verbosity;
  double reduction;
  double abslimit;
  mutable bool converged;
};




class Banach
{
  typedef std::size_t size_type;

public:
  Banach ()
    : maxit(25), linesearchsteps(10), verbosity(0),
      reduction(1e-14), abslimit(1e-30),  sigma(1.0), converged(false)
  {}

  void set_maxit (size_type n)
  {
    maxit = n;
  }

  void set_sigma (double sigma_)
  {
    sigma = sigma_;
  }

  void set_linesearchsteps (size_type n)
  {
    linesearchsteps = n;
  }

  void set_verbosity (size_type n)
  {
    verbosity = n;
  }

  void set_abslimit (double l)
  {
    abslimit = l;
  }

  void set_reduction (double l)
  {
    reduction = l;
  }

  template<class M>
  void solve (const M& model, Vector<typename M::number_type>& x) const
  {
    typedef typename M::number_type N;
    Vector<N> r(model.size());              // residual
    Vector<N> y(model.size());              // temporary solution in line search

    model.F(x,r);                           // compute nonlinear residual
    N R0(norm(r));                          // norm of initial residual
    N R(R0);                                // current residual norm
    if (verbosity>=1)
      {
        std::cout « "Banach "
                  « " norm=" « std::scientific « std::showpoint
                  « std::setprecision(4) « R0
                  « std::endl;
      }

    converged = false;
```

```
      for (size_type i=1; i<=maxit; i++)                // do iterations
        {
          // check absolute size of residual
          if (R<=abslimit)
            {
              converged = true;
              return;
            }

          // next iterate
          y = x;
          y.update(-sigma,r);                       // y = x+lambda*z
          model.F(y,r);                             // r = F(y)
          N newR(norm(r));                // compute norm
          if (verbosity>=2)
            {
              std::cout « "    "  « std::setw(3) « i
                        « " norm=" « std::scientific « std::showpoint
                        « std::setprecision(4) « newR
                        « " red=" « std::scientific « std::showpoint
                        « std::setprecision(4) « newR/R
                        « std::endl;
            }
          x = y;                                  // accept new iterate
          R = newR;                               // remember new norm

          // check convergence
          if (R<=reduction*R0 || R<=abslimit)
            {
              if (verbosity>=1)
                {
                  std::cout « "Banach converged in "  « i « " steps"
                            « " reduction=" « std::scientific « std::showpoint
                            « std::setprecision(4) « R/R0
                            « std::endl;
                }
              converged = true;
              return;
            }
        }
    }

    bool has_converged () const
    {
      return converged;
    }

  private:
    size_type maxit;
    size_type linesearchsteps;
    size_type verbosity;
    double reduction;
    double abslimit;
    double sigma;
    mutable bool converged;
  };

} // namespace hdnum

#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.

```
// -*- tab-width: 4; indent-tabs-mode: nil -*-
#ifndef HDNUM_ODE_HH
#define HDNUM_ODE_HH

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

namespace hdnum {

  template<class M>
  class EE
  {
  public:
    typedef typename M::size_type size_type;

    typedef typename M::time_type time_type;

    typedef typename M::number_type number_type;

    EE (const M& model_)
```

```
    : model(model_), u(model.size()), f(model.size())
  {
    model.initialize(t,u);
    dt = 0.1;
  }

  void set_dt (time_type dt_)
  {
    dt = dt_;
  }

  void step ()
  {
    model.f(t,u,f);    // evaluate model
    u.update(dt,f);    // advance state
    t += dt;           // advance time
  }

  void set_state (time_type t_, const Vector<number_type>& u_)
  {
    t = t_;
    u = u_;
  }

  const Vector<number_type>& get_state () const
  {
    return u;
  }

  time_type get_time () const
  {
    return t;
  }

  time_type get_dt () const
  {
    return dt;
  }

  size_type get_order () const
  {
    return 1;
  }

private:
  const M& model;
  time_type t, dt;
  Vector<number_type> u;
  Vector<number_type> f;
};

template<class M>
class ModifiedEuler
{
public:
  typedef typename M::size_type size_type;

  typedef typename M::time_type time_type;

  typedef typename M::number_type number_type;

  ModifiedEuler (const M& model_)
    : model(model_), u(model.size()), w(model.size()), k1(model.size()), k2(model.size())
  {
    c2 = 0.5;
    a21 = 0.5;
    b2 = 1.0;
    model.initialize(t,u);
    dt = 0.1;
  }

  void set_dt (time_type dt_)
  {
    dt = dt_;
  }

  void step ()
  {
    // stage 1
    model.f(t,u,k1);

    // stage 2
    w = u;
    w.update(dt*a21,k1);
    model.f(t+c2*dt,w,k2);

    // final
```

```
    u.update(dt*b2,k2);
    t += dt;
  }

  void set_state (time_type t_, const Vector<number_type>& u_)
  {
    t = t_;
    u = u_;
  }

  const Vector<number_type>& get_state () const
  {
    return u;
  }

  time_type get_time () const
  {
    return t;
  }

  time_type get_dt () const
  {
    return dt;
  }

  size_type get_order () const
  {
    return 2;
  }

private:
  const M& model;
  time_type t, dt;
  time_type c2,a21,b2;
  Vector<number_type> u,w;
  Vector<number_type> k1,k2;
};


template<class M>
class Heun2
{
public:
  typedef typename M::size_type size_type;

  typedef typename M::time_type time_type;

  typedef typename M::number_type number_type;

  Heun2 (const M& model_)
    : model(model_), u(model.size()), w(model.size()), k1(model.size()), k2(model.size())
  {
    c2 = 1.0;
    a21 = 1.0;
    b1 = 0.5;
    b2 = 0.5;
    model.initialize(t,u);
    dt = 0.1;
  }

  void set_dt (time_type dt_)
  {
    dt = dt_;
  }

  void step ()
  {
    // stage 1
    model.f(t,u,k1);

    // stage 2
    w = u;
    w.update(dt*a21,k1);
    model.f(t+c2*dt,w,k2);

    // final
    u.update(dt*b1,k1);
    u.update(dt*b2,k2);
    t += dt;
  }

  void set_state (time_type t_, const Vector<number_type>& u_)
  {
    t = t_;
    u = u_;
  }

```

```
    const Vector<number_type>& get_state () const
    {
      return u;
    }

    time_type get_time () const
    {
      return t;
    }

    time_type get_dt () const
    {
      return dt;
    }

    size_type get_order () const
    {
      return 2;
    }

  private:
    const M& model;
    time_type t, dt;
    time_type c2,a21,b1,b2;
    Vector<number_type> u,w;
    Vector<number_type> k1,k2;
  };


  template<class M>
  class Heun3
  {
  public:
    typedef typename M::size_type size_type;

    typedef typename M::time_type time_type;

    typedef typename M::number_type number_type;

    Heun3 (const M& model_)
      : model(model_), u(model.size()), w(model.size()), k1(model.size()),
        k2(model.size()), k3(model.size())
    {
      c2 = time_type(1.0)/time_type(3.0);
      c3 = time_type(2.0)/time_type(3.0);
      a21 = time_type(1.0)/time_type(3.0);
      a32 = time_type(2.0)/time_type(3.0);
      b1 = 0.25;
      b2 = 0.0;
      b3 = 0.75;
      model.initialize(t,u);
      dt = 0.1;
    }

    void set_dt (time_type dt_)
    {
      dt = dt_;
    }

    void step ()
    {
      // stage 1
      model.f(t,u,k1);

      // stage 2
      w = u;
      w.update(dt*a21,k1);
      model.f(t+c2*dt,w,k2);

      // stage 3
      w = u;
      w.update(dt*a32,k2);
      model.f(t+c3*dt,w,k3);

      // final
      u.update(dt*b1,k1);
      u.update(dt*b3,k3);
      t += dt;
    }

    void set_state (time_type t_, const Vector<number_type>& u_)
    {
      t = t_;
      u = u_;
    }

    const Vector<number_type>& get_state () const
```

```
    {
      return u;
    }

    time_type get_time () const
    {
      return t;
    }

    time_type get_dt () const
    {
      return dt;
    }

    size_type get_order () const
    {
      return 3;
    }

  private:
    const M& model;
    time_type t, dt;
    time_type c2,c3,a21,a31,a32,b1,b2,b3;
    Vector<number_type> u,w;
    Vector<number_type> k1,k2,k3;
  };

  template<class M>
  class Kutta3
  {
  public:
    typedef typename M::size_type size_type;

    typedef typename M::time_type time_type;

    typedef typename M::number_type number_type;

    Kutta3 (const M& model_)
      : model(model_), u(model.size()), w(model.size()), k1(model.size()),
        k2(model.size()), k3(model.size())
    {
      c2 = 0.5;
      c3 = 1.0;
      a21 = 0.5;
      a31 = -1.0;
      a32 = 2.0;
      b1 = time_type(1.0)/time_type(6.0);
      b2 = time_type(4.0)/time_type(6.0);
      b3 = time_type(1.0)/time_type(6.0);
      model.initialize(t,u);
      dt = 0.1;
    }

    void set_dt (time_type dt_)
    {
      dt = dt_;
    }

    void step ()
    {
      // stage 1
      model.f(t,u,k1);

      // stage 2
      w = u;
      w.update(dt*a21,k1);
      model.f(t+c2*dt,w,k2);

      // stage 3
      w = u;
      w.update(dt*a31,k1);
      w.update(dt*a32,k2);
      model.f(t+c3*dt,w,k3);

      // final
      u.update(dt*b1,k1);
      u.update(dt*b2,k2);
      u.update(dt*b3,k3);
      t += dt;
    }

    void set_state (time_type t_, const Vector<number_type>& u_)
    {
      t = t_;
      u = u_;
    }

```

```
  const Vector<number_type>& get_state () const
  {
    return u;
  }

  time_type get_time () const
  {
    return t;
  }

  time_type get_dt () const
  {
    return dt;
  }

  size_type get_order () const
  {
    return 3;
  }

private:
  const M& model;
  time_type t, dt;
  time_type c2,c3,a21,a31,a32,b1,b2,b3;
  Vector<number_type> u,w;
  Vector<number_type> k1,k2,k3;
};

template<class M>
class RungeKutta4
{
public:
  typedef typename M::size_type size_type;

  typedef typename M::time_type time_type;

  typedef typename M::number_type number_type;

  RungeKutta4 (const M& model_)
    : model(model_), u(model.size()), w(model.size()), k1(model.size()),
      k2(model.size()), k3(model.size()), k4(model.size())
  {
    c2 = 0.5;
    c3 = 0.5;
    c4 = 1.0;
    a21 = 0.5;
    a32 = 0.5;
    a43 = 1.0;
    b1 = time_type(1.0)/time_type(6.0);
    b2 = time_type(2.0)/time_type(6.0);
    b3 = time_type(2.0)/time_type(6.0);
    b4 = time_type(1.0)/time_type(6.0);
    model.initialize(t,u);
    dt = 0.1;
  }

  void set_dt (time_type dt_)
  {
    dt = dt_;
  }

  void step ()
  {
    // stage 1
    model.f(t,u,k1);

    // stage 2
    w = u;
    w.update(dt*a21,k1);
    model.f(t+c2*dt,w,k2);

    // stage 3
    w = u;
    w.update(dt*a32,k2);
    model.f(t+c3*dt,w,k3);

    // stage 4
    w = u;
    w.update(dt*a43,k3);
    model.f(t+c4*dt,w,k4);

    // final
    u.update(dt*b1,k1);
    u.update(dt*b2,k2);
    u.update(dt*b3,k3);
    u.update(dt*b4,k4);
    t += dt;
```

```
    }

    void set_state (time_type t_, const Vector<number_type>& u_)
    {
      t = t_;
      u = u_;
    }

    const Vector<number_type>& get_state () const
    {
      return u;
    }

    time_type get_time () const
    {
      return t;
    }

    time_type get_dt () const
    {
      return dt;
    }

    size_type get_order () const
    {
      return 4;
    }

  private:
    const M& model;
    time_type t, dt;
    time_type c2,c3,c4,a21,a32,a43,b1,b2,b3,b4;
    Vector<number_type> u,w;
    Vector<number_type> k1,k2,k3,k4;
  };

  template<class M>
  class RKF45
  {
  public:
    typedef typename M::size_type size_type;

    typedef typename M::time_type time_type;

    typedef typename M::number_type number_type;

    RKF45 (const M& model_)
      : model(model_), u(model.size()), w(model.size()), ww(model.size()), k1(model.size()),
        k2(model.size()), k3(model.size()), k4(model.size()), k5(model.size()), k6(model.size()),
        steps(0), rejected(0)
    {
      TOL = time_type(0.0001);
      rho = time_type(0.8);
      alpha = time_type(0.25);
      beta = time_type(4.0);
      dt_min = 1E-12;

      c2 = time_type(1.0)/time_type(4.0);
      c3 = time_type(3.0)/time_type(8.0);
      c4 = time_type(12.0)/time_type(13.0);
      c5 = time_type(1.0);
      c6 = time_type(1.0)/time_type(2.0);

      a21 = time_type(1.0)/time_type(4.0);

      a31 = time_type(3.0)/time_type(32.0);
      a32 = time_type(9.0)/time_type(32.0);

      a41 = time_type(1932.0)/time_type(2197.0);
      a42 = time_type(-7200.0)/time_type(2197.0);
      a43 = time_type(7296.0)/time_type(2197.0);

      a51 = time_type(439.0)/time_type(216.0);
      a52 = time_type(-8.0);
      a53 = time_type(3680.0)/time_type(513.0);
      a54 = time_type(-845.0)/time_type(4104.0);

      a61 = time_type(-8.0)/time_type(27.0);
      a62 = time_type(2.0);
      a63 = time_type(-3544.0)/time_type(2565.0);
      a64 = time_type(1859.0)/time_type(4104.0);
      a65 = time_type(-11.0)/time_type(40.0);

      b1 = time_type(25.0)/time_type(216.0);
      b2 = time_type(0.0);
      b3 = time_type(1408.0)/time_type(2565.0);
      b4 = time_type(2197.0)/time_type(4104.0);
```

```
      b5 = time_type(-1.0)/time_type(5.0);

      bb1 = time_type(16.0)/time_type(135.0);
      bb2 = time_type(0.0);
      bb3 = time_type(6656.0)/time_type(12825.0);
      bb4 = time_type(28561.0)/time_type(56430.0);
      bb5 = time_type(-9.0)/time_type(50.0);
      bb6 = time_type(2.0)/time_type(55.0);

      model.initialize(t,u);
      dt = 0.1;
    }

    void set_dt (time_type dt_)
    {
      dt = dt_;
    }

    void set_TOL (time_type TOL_)
    {
      TOL = TOL_;
    }

    void step ()
    {
      steps++;

      // stage 1
      model.f(t,u,k1);

      // stage 2
      w = u;
      w.update(dt*a21,k1);
      model.f(t+c2*dt,w,k2);

      // stage 3
      w = u;
      w.update(dt*a31,k1);
      w.update(dt*a32,k2);
      model.f(t+c3*dt,w,k3);

      // stage 4
      w = u;
      w.update(dt*a41,k1);
      w.update(dt*a42,k2);
      w.update(dt*a43,k3);
      model.f(t+c4*dt,w,k4);

      // stage 5
      w = u;
      w.update(dt*a51,k1);
      w.update(dt*a52,k2);
      w.update(dt*a53,k3);
      w.update(dt*a54,k4);
      model.f(t+c5*dt,w,k5);

      // stage 6
      w = u;
      w.update(dt*a61,k1);
      w.update(dt*a62,k2);
      w.update(dt*a63,k3);
      w.update(dt*a64,k4);
      w.update(dt*a65,k5);
      model.f(t+c6*dt,w,k6);

      // compute order 4 approximation
      w = u;
      w.update(dt*b1,k1);
      w.update(dt*b2,k2);
      w.update(dt*b3,k3);
      w.update(dt*b4,k4);
      w.update(dt*b5,k5);

      // compute order 5 approximation
      ww = u;
      ww.update(dt*bb1,k1);
      ww.update(dt*bb2,k2);
      ww.update(dt*bb3,k3);
      ww.update(dt*bb4,k4);
      ww.update(dt*bb5,k5);
      ww.update(dt*bb6,k6);

      // estimate local error
      w -= ww;
      time_type error(norm(w));
      time_type dt_opt(dt*pow(rho*TOL/error,0.2));
      dt_opt = std::min(beta*dt,std::max(alpha*dt,dt_opt));
```

```
      //std::cout « "est. error=" « error « " dt_opt=" « dt_opt « std::endl;

      if (error<=TOL)
        {
          t += dt;
          u = ww;
          dt = dt_opt;
        }
      else
        {
          rejected++;
          dt = dt_opt;
          if (dt>dt_min) step();
        }
    }

    const Vector<number_type>& get_state () const
    {
      return u;
    }

    time_type get_time () const
    {
      return t;
    }

    time_type get_dt () const
    {
      return dt;
    }

    size_type get_order () const
    {
      return 5;
    }

    void get_info () const
    {
      std::cout « "RE: steps=" « steps « " rejected=" « rejected « std::endl;
    }

  private:
    const M& model;
    time_type t, dt;
    time_type TOL,rho,alpha,beta,dt_min;
    time_type c2,c3,c4,c5,c6;
    time_type a21,a31,a32,a41,a42,a43,a51,a52,a53,a54,a61,a62,a63,a64,a65;
    time_type b1,b2,b3,b4,b5; // 4th order
    time_type bb1,bb2,bb3,bb4,bb5,bb6; // 5th order
    Vector<number_type> u,w,ww;
    Vector<number_type> k1,k2,k3,k4,k5,k6;
    mutable size_type steps, rejected;
  };


  template<class M, class S>
  class RE
  {
  public:
    typedef typename M::size_type size_type;

    typedef typename M::time_type time_type;

    typedef typename M::number_type number_type;

    RE (const M& model_, S& solver_)
      : model(model_), solver(solver_), u(model.size()),
        wlow(model.size()), whigh(model.size()), ww(model.size()),
        steps(0), rejected(0)
    {
      model.initialize(t,u); // initialize state
      dt = 0.1;              // set initial time step
      two_power_m = 1.0;
      for (size_type i=0; i<solver.get_order(); i++)
        two_power_m *= 2.0;
      TOL = time_type(0.0001);
      rho = time_type(0.8);
      alpha = time_type(0.25);
      beta = time_type(4.0);
      dt_min = 1E-12;
    }

    void set_dt (time_type dt_)
    {
      dt = dt_;
    }

```

```
    void set_TOL (time_type TOL_)
    {
      TOL = TOL_;
    }

    void step ()
    {
      // count steps done
      steps++;

      // do 1 step with 2*dt
      time_type H(2.0*dt);
      solver.set_state(t,u);
      solver.set_dt(H);
      solver.step();
      wlow = solver.get_state();

      // do 2 steps with dt
      solver.set_state(t,u);
      solver.set_dt(dt);
      solver.step();
      solver.step();
      whigh = solver.get_state();

      // estimate local error
      ww = wlow;
      ww -= whigh;
      time_type error(norm(ww)/(pow(H,1.0+solver.get_order())*(1.0-1.0/two_power_m)));
      time_type dt_opt(pow(rho*TOL/error,1.0/((time_type)solver.get_order())));
      dt_opt = std::min(beta*dt,std::max(alpha*dt,dt_opt));
      //std::cout « "est. error=" « error « " dt_opt=" « dt_opt « std::endl;

      if (dt<=dt_opt)
        {
          t += H;
          u = whigh;
          u *= two_power_m;
          u -= wlow;
          u /= two_power_m-1.0;
          dt = dt_opt;
        }
      else
        {
          rejected++;
          dt = dt_opt;
          if (dt>dt_min) step();
        }
    }

    const Vector<number_type>& get_state () const
    {
      return u;
    }

    time_type get_time () const
    {
      return t;
    }

    time_type get_dt () const
    {
      return dt;
    }

    size_type get_order () const
    {
      return solver.get_order()+1;
    }

    void get_info () const
    {
      std::cout « "RE: steps=" « steps « " rejected=" « rejected « std::endl;
    }

  private:
    const M& model;
    S& solver;
    time_type t, dt;
    time_type two_power_m;
    Vector<number_type> u,wlow,whigh,ww;
    time_type TOL,rho,alpha,beta,dt_min;
    mutable size_type steps, rejected;
  };


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

```
{
  // h_n f(t_n, y_n) - y_n + y_{n-1} = 0
  class NonlinearProblem
  {
  public:
    typedef typename M::size_type size_type;

    typedef typename M::number_type number_type;

    NonlinearProblem (const M& model_, const Vector<number_type>& yold_,
                      typename M::time_type tnew_, typename M::time_type dt_)
      : model(model_), yold(yold_), tnew(tnew_), dt(dt_)
    {}

    std::size_t size () const
    {
      return model.size();
    }

    void F (const Vector<number_type>& x, Vector<number_type>& result) const
    {
      model.f(tnew,x,result);
      result *= dt;
      result -= x;
      result += yold;
    }

    void F_x (const Vector<number_type>& x, DenseMatrix<number_type>& result) const
    {
       model.f_x(tnew,x,result);
       result *= dt;
       for (size_type i=0; i<model.size(); i++) result[i][i] -= number_type(1.0);
     }

     void set_tnew_dt (typename M::time_type tnew_, typename M::time_type dt_)
     {
       tnew = tnew_;
       dt = dt_;
     }

   private:
     const M& model;
     const Vector<number_type>& yold;
     typename M::time_type tnew;
     typename M::time_type dt;
   };

 public:
   typedef typename M::size_type size_type;

   typedef typename M::time_type time_type;

   typedef typename M::number_type number_type;

   IE (const M& model_, const S& newton_)
     : verbosity(0), model(model_), newton(newton_), u(model.size()), unew(model.size())
   {
     model.initialize(t,u);
     dt = dtmax = 0.1;
   }

   void set_dt (time_type dt_)
   {
     dt = dtmax = dt_;
   }

   void set_verbosity (size_type verbosity_)
   {
     verbosity = verbosity_;
   }

   void step ()
   {
     if (verbosity>=2)
       std::cout << "IE: step" << " t=" << t << " dt=" << dt << std::endl;
     NonlinearProblem nlp(model,u,t+dt,dt);
     bool reduced = false;
     error = false;
     while (1)
       {
         unew = u;
         newton.solve(nlp,unew);
         if (newton.has_converged())
           {
             u = unew;
             t += dt;
             if (!reduced && dt<dtmax-1e-13)
```

```
                {
                  dt = std::min(2.0*dt,dtmax);
                  if (verbosity>0)
                    std::cout « "IE: increasing time step to " « dt « std::endl;
                }
              return;
            }
          else
            {
              if (dt<1e-12)
                {
                  HDNUM_ERROR("time step too small in implicit Euler");
                  error = true;
                  break;
                }
              dt *= 0.5;
              reduced = true;
              nlp.set_tnew_dt(t+dt,dt);
              if (verbosity>0) std::cout « "IE: reducing time step to " « dt « std::endl;
            }
        }
    }

    bool get_error () const
    {
      return error;
    }

    void set_state (time_type t_, const Vector<number_type>& u_)
    {
      t = t_;
      u = u_;
    }

    const Vector<number_type>& get_state () const
    {
      return u;
    }

    time_type get_time () const
    {
      return t;
    }

    time_type get_dt () const
    {
      return dt;
    }

    size_type get_order () const
    {
      return 1;
    }

    void get_info () const
    {
    }

  private:
    size_type verbosity;
    const M& model;
    const S& newton;
    time_type t, dt, dtmax;
    number_type reduction;
    size_type linesearchsteps;
    Vector<number_type> u;
    Vector<number_type> unew;
    mutable bool error;
  };

  template<class M, class S>
  class DIRK
  {
  public:

    typedef typename M::size_type size_type;

    typedef typename M::time_type time_type;

    typedef typename M::number_type number_type;

    typedef DenseMatrix<number_type> ButcherTableau;

  private:

    static ButcherTableau initTableau(const std::string method)
    {
```

```
      if(method.find("Implicit Euler") != std::string::npos){
        ButcherTableau butcher(2,2,0.0);
        butcher[1][1] = 1;
        butcher[0][1] = 1;
        butcher[0][0] = 1;

        return butcher;
      }
      else if(method.find("Alexander") != std::string::npos){
        ButcherTableau butcher(3,3,0.0);
        const number_type alpha = 1. - sqrt(2.)/2.;
        butcher[0][0] = alpha;
        butcher[0][1] = alpha;

        butcher[1][0] = 1.;
        butcher[1][1] = 1. - alpha;
        butcher[1][2] = alpha;

        butcher[2][1] = 1. - alpha;
        butcher[2][2] = alpha;

        return butcher;
      }
      else if(method.find("Crouzieux") != std::string::npos){
        ButcherTableau butcher(3,3,0.0);
        const number_type beta = 1./2./sqrt(3);
        butcher[0][0] = 0.5 + beta;
        butcher[0][1] = 0.5 + beta;

        butcher[1][0] = 0.5 - beta;
        butcher[1][1] = -1. / sqrt(3);
        butcher[1][2] = 0.5 + beta;

        butcher[2][1] = 0.5;
        butcher[2][2] = 0.5;

        return butcher;
      }
      else if(method.find("Midpoint Rule") != std::string::npos){
        ButcherTableau butcher(2,2,0.0);
        butcher[0][0] = 0.5;
        butcher[0][1] = 0.5;
        butcher[1][1] = 1;

        return butcher;
      }
      else if(method.find("Fractional Step Theta") != std::string::npos){
        ButcherTableau butcher(5,5,0.0);
        const number_type theta = 1 - sqrt(2.)/2.;
        const number_type alpha = 2. - sqrt(2.);
        const number_type beta = 1. - alpha;
        butcher[1][0] = theta;
        butcher[1][1] = beta * theta;
        butcher[1][2] = alpha * theta;

        butcher[2][0] = 1.-theta;
        butcher[2][1] = beta * theta;
        butcher[2][2] = alpha * (1.-theta);
        butcher[2][3] = alpha * theta;

        butcher[3][0] = 1.;
        butcher[3][1] = beta * theta;
        butcher[3][2] = alpha * (1.-theta);
        butcher[3][3] = (alpha + beta) * theta;
        butcher[3][4] = alpha * theta;

        butcher[4][1] = beta * theta;
        butcher[4][2] = alpha * (1.-theta);
        butcher[4][3] = (alpha + beta) * theta;
        butcher[4][4] = alpha * theta;

        return butcher;
      }
      else{
        HDNUM_ERROR("Order not available for Runge Kutta solver.");
      }
    }

    static int initOrder(const std::string method)
    {
      if(method.find("Implicit Euler") != std::string::npos){
        return 1;
      }
      else if(method.find("Alexander") != std::string::npos){
        return 2;
      }
      else if(method.find("Crouzieux") != std::string::npos){
```

```
        return 3;
      }
      else if(method.find("Midpoint Rule") != std::string::npos){
        return 2;
      }
      else if(method.find("Fractional Step Theta") != std::string::npos){
        return 2;
      }
      else{
        HDNUM_ERROR("Order not available for Runge Kutta solver.");
      }
    }


    // h_n f(t_n, y_n) - y_n + y_{n-1} = 0
    class NonlinearProblem
    {
    public:
      typedef typename M::size_type size_type;

      typedef typename M::number_type number_type;

      NonlinearProblem (const M& model_, const Vector<number_type>& yold_,
                        typename M::time_type told_, typename M::time_type dt_,
                        const ButcherTableau & butcher_, const int rk_step_,
                        const std::vector< Vector<number_type> > & k_)
        : model(model_), yold(yold_), told(told_),
          dt(dt_), butcher(butcher_), rk_step(rk_step_), k_old(model.size(),0)
      {
        for(int i=0; i<rk_step; ++i)
          k_old.update(butcher[rk_step][1+i] * dt, k_[i]);
      }

      std::size_t size () const
      {
        return model.size();
      }

      void F (const Vector<number_type>& x, Vector<number_type>& result) const
      {
        result = k_old;

        Vector<number_type> current_z(x);
        current_z.update(1.,yold);

        const number_type tnew = told + butcher[rk_step][0] * dt;

        Vector<number_type> current_k(model.size(),0.);
        model.f(tnew,current_z,current_k);
        result.update(butcher[rk_step][rk_step+1] * dt, current_k);

        result.update(-1.,x);
      }

      void F_x (const Vector<number_type>& x, DenseMatrix<number_type>& result) const
      {
        const number_type tnew = told + butcher[rk_step][0] * dt;

        Vector<number_type> current_z(x);
        current_z.update(1.,yold);

        model.f_x(tnew,current_z,result);

        result *= dt * butcher[rk_step][rk_step+1];

        for (size_type i=0; i<model.size(); i++) result[i][i] -= number_type(1.0);
      }

      void set_told_dt (typename M::time_type told_, typename M::time_type dt_)
      {
        told = told_;
        dt = dt_;
      }

    private:
      const M& model;
      const Vector<number_type>& yold;
      typename M::time_type told;
      typename M::time_type dt;
      const ButcherTableau & butcher;
      const int rk_step;
      Vector<number_type> k_old;
    };

  public:

    DIRK (const M& model_, const S& newton_, const ButcherTableau & butcher_, const int order_)
```

```
     : verbosity(0), butcher(butcher_), model(model_), newton(newton_),
       u(model.size()), order(order_)
    {
     model.initialize(t,u);
     dt = dtmax = 0.1;
    }

    DIRK (const M& model_, const S& newton_, const std::string method)
     : verbosity(0), butcher(initTableau(method)), model(model_), newton(newton_), u(model.size()),
       order(initOrder(method))
    {
     model.initialize(t,u);
     dt = dtmax = 0.1;
    }


    void set_dt (time_type dt_)
    {
     dt = dtmax = dt_;
    }

    void set_verbosity (size_type verbosity_)
    {
     verbosity = verbosity_;
    }

    void step ()
    {

     const size_type R = butcher.colsize()-1;

     bool reduced = false;
     error = false;
     if(verbosity>=2)
       std::cout « "DIRK: step to" « " t+dt=" « t+dt « " dt=" « dt « std::endl;

     while (1)
       {
        bool converged = true;

        // Perform R Runge-Kutta steps
        std::vector< Vector<number_type> > k;
        for(size_type i=0; i<R; ++i) {
          if (verbosity>=2)
            std::cout « "DIRK: step nr "« i « std::endl;

          Vector<number_type> current_z(model.size(),0.0);

          // Set starting value of k_i
          // model.f(t,u,current_k);

          // Solve nonlinear problem
          NonlinearProblem nlp(model,u,t,dt,butcher,i,k);

          newton.solve(nlp,current_z);

          converged = converged && newton.has_converged();
          if(!converged)
            break;

          current_z.update(1., u);
          const number_type t_i = t + butcher[i][0] * dt;
          Vector<number_type>current_k(model.size(),0.);
          model.f(t_i,current_z,current_k);

          k.push_back( current_k );
        }

        if (converged)
          {
            if(verbosity >= 2)
              std::cout « "DIRK finished"« std::endl;

            // Update to new solution
            for(size_type i=0; i<R; ++i)
              u.update(dt*butcher[R][1+i],k[i]);

            t += dt;
            if (!reduced && dt<dtmax-1e-13)
              {
                dt = std::min(2.0*dt,dtmax);
                if (verbosity>0)
                  std::cout « "DIRK: increasing time step to " « dt « std::endl;
              }
            return;
          }
        else
```

```
          {
            if (dt<1e-12)
              {
                HDNUM_ERROR("time step too small in implicit Euler");
                error = true;
                break;
              }
            dt *= 0.5;
            reduced = true;
            if (verbosity>0) std::cout « "DIRK: reducing time step to " « dt « std::endl;
          }
      }
  }

  bool get_error () const
  {
    return error;
  }

  void set_state (time_type t_, const Vector<number_type>& u_)
  {
    t = t_;
    u = u_;
  }

  const Vector<number_type>& get_state () const
  {
    return u;
  }

  time_type get_time () const
  {
    return t;
  }

  time_type get_dt () const
  {
    return dt;
  }

  size_type get_order () const
  {
    return order;
  }

  void get_info () const
  {
  }

private:
  size_type verbosity;
  const DenseMatrix<number_type> butcher;
  const M& model;
  const S& newton;
  time_type t, dt, dtmax;
  number_type reduction;
  size_type linesearchsteps;
  Vector<number_type> u;
  int order;
  mutable bool error;
};

template<class T, class N>
inline void gnuplot (const std::string& fname, const std::vector<T> t, const std::vector<Vector<N> > u)
{
  std::fstream f(fname.c_str(),std::ios::out);
  for (typename std::vector<T>::size_type n=0; n<t.size(); n++)
    {
      f « std::scientific « std::showpoint
        « std::setprecision(16) « t[n];
      for (typename Vector<N>::size_type i=0; i<u[n].size(); i++)
        f « " " « std::scientific « std::showpoint
          « std::setprecision(u[n].precision()) « u[n][i];
      f « std::endl;
    }
  f.close();
}

template<class T, class N>
inline void gnuplot (const std::string& fname, const std::vector<T> t, const std::vector<Vector<N> > u, const std::vector<T> dt)
{
  std::fstream f(fname.c_str(),std::ios::out);
  for (typename std::vector<T>::size_type n=0; n<t.size(); n++)
    {
      f « std::scientific « std::showpoint
```

```
            « std::setprecision(16) « t[n];
          for (typename Vector<N>::size_type i=0; i<u[n].size(); i++)
            f « " " « std::scientific « std::showpoint
              « std::setprecision(u[n].precision()) « u[n][i];
          f « " " « std::scientific « std::showpoint
            « std::setprecision(16) « dt[n];
          f « std::endl;
        }
      f.close();
    }

} // namespace hdnum

#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.

```
// -*- tab-width: 4; indent-tabs-mode: nil -*-
#ifndef __OPCOUNTER__
#define __OPCOUNTER__

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

namespace hdnum {
  namespace oc {
    template<typename F>
    class OpCounter;
  }
}

namespace hdnum {
  namespace oc {
    template<typename F>
    class OpCounter
    {

    public:

      using size_type = std::size_t;

      using value_type = F;

      OpCounter()
        : _v()
      {}

      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)
```

```
      : _v(t)
    {}

    OpCounter(const F& f)
      : _v(f)
    {}

    OpCounter(F&& f)
      : _v(f)
    {}

    explicit OpCounter(const char* s)
      : _v(strtod(s,nullptr))
    {}

    OpCounter& operator=(const char* s)
    {
      _v = strtod(s,nullptr);
      return *this;
    }

    explicit operator F() const
    {
      return _v;
    }

    OpCounter& operator=(const F& f)
    {
      _v = f;
      return *this;
    }

    OpCounter& operator=(F&& f)
    {
      _v = f;
      return *this;
    }

    friend std::ostream& operator<<(std::ostream& os, const OpCounter& f)
    {
      os << "OC(" << f._v << ")";
      return os;
    }

    friend std::istringstream& operator>>(std::istringstream& iss, OpCounter& f)
    {
      iss >> f._v;
      return iss;
    }

    F* data()
    {
      return &_v;
    }

    const F* data() const
    {
      return &_v;
    }

    F _v;

    struct Counters {

      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;

      Counters()
        : addition_count(0)
        , multiplication_count(0)
        , division_count(0)
        , exp_count(0)
        , pow_count(0)
        , sin_count(0)
        , sqrt_count(0)
        , comparison_count(0)
      {}

      void reset()
      {
        addition_count = 0;
```

```
        multiplication_count = 0;
        division_count = 0;
        exp_count = 0;
        pow_count = 0;
        sin_count = 0;
        sqrt_count = 0;
        comparison_count = 0;
      }

      template<typename Stream>
      void reportOperations(Stream& os, bool doReset = false)
      {
        os « "additions: " « addition_count « std::endl
           « "multiplications: " « multiplication_count « std::endl
           « "divisions: " « division_count « std::endl
           « "exp: " « exp_count « std::endl
           « "pow: " « pow_count « std::endl
           « "sin: " « sin_count « std::endl
           « "sqrt: " « sqrt_count « std::endl
           « "comparisons: " « comparison_count « std::endl
           « std::endl
           « "total: " « addition_count + multiplication_count + division_count + exp_count + pow_count + sin_count + sqrt_count + comparison_count « std::endl;

        if (doReset)
          reset();
      }

      size_type totalOperationCount(bool doReset=false)
      {
        if (doReset)
          reset();

        return addition_count + multiplication_count + division_count + exp_count + pow_count + sin_count + sqrt_count + comparison_count;
      }

      Counters& operator+=(const Counters& rhs)
      {
        addition_count += rhs.addition_count;
        multiplication_count += rhs.multiplication_count;
        division_count += rhs.division_count;
        exp_count += rhs.exp_count;
        pow_count += rhs.pow_count;
        sin_count += rhs.sin_count;
        sqrt_count += rhs.sqrt_count;
        comparison_count += rhs.comparison_count;
        return *this;
      }

      Counters operator-(const Counters& rhs)
      {
        Counters r;
        r.addition_count = addition_count - rhs.addition_count;
        r.multiplication_count = multiplication_count - rhs.multiplication_count;
        r.division_count = division_count - rhs.division_count;
        r.exp_count = exp_count - rhs.exp_count;
        r.pow_count = pow_count - rhs.pow_count;
        r.sin_count = sin_count - rhs.sin_count;
        r.sqrt_count = sqrt_count - rhs.sqrt_count;
        r.comparison_count = comparison_count - rhs.comparison_count;
        return r;
      }

    };

    static void additions(std::size_t n)
    {
      counters.addition_count += n;
    }

    static void multiplications(std::size_t n)
    {
      counters.multiplication_count += n;
    }

    static void divisions(std::size_t n)
    {
      counters.division_count += n;
    }

    static void reset()
    {
      counters.reset();
    }

    template<typename Stream>
```

```
    static void reportOperations(Stream& os, bool doReset = false)
    {
      counters.reportOperations(os,doReset);
    }

    static size_type totalOperationCount(bool doReset=false)
    {
      return counters.totalOperationCount(doReset);
    }

    static Counters counters;

  };

  template<typename F>
  typename OpCounter<F>::Counters OpCounter<F>::counters;

  // ********************************************************************************
  // negation
  // ********************************************************************************

  template<typename F>
  OpCounter<F> operator-(const OpCounter<F>& a)
  {
    ++OpCounter<F>::counters.addition_count;
    return {-a._v};
  }


  // ********************************************************************************
  // addition
  // ********************************************************************************

  template<typename F>
  OpCounter<F> operator+(const OpCounter<F>& a, const OpCounter<F>& b)
  {
    ++OpCounter<F>::counters.addition_count;
    return {a._v + b._v};
  }

  template<typename F>
  OpCounter<F> operator+(const OpCounter<F>& a, const F& b)
  {
    ++OpCounter<F>::counters.addition_count;
    return {a._v + b};
  }

  template<typename F>
  OpCounter<F> operator+(const F& a, const OpCounter<F>& b)
  {
    ++OpCounter<F>::counters.addition_count;
    return {a + b._v};
  }

  template<typename F, typename T>
  typename std::enable_if<
    std::is_arithmetic<T>::value,
    OpCounter<F>
    >::type
  operator+(const OpCounter<F>& a, const T& b)
  {
    ++OpCounter<F>::counters.addition_count;
    return {a._v + b};
  }

  template<typename F, typename T>
  typename std::enable_if<
    std::is_arithmetic<T>::value,
    OpCounter<F>
    >::type
  operator+(const T& a, const OpCounter<F>& b)
  {
    ++OpCounter<F>::counters.addition_count;
    return {a + b._v};
  }

  template<typename F>
  OpCounter<F>& operator+=(OpCounter<F>& a, const OpCounter<F>& b)
  {
    ++OpCounter<F>::counters.addition_count;
    a._v += b._v;
    return a;
  }

  template<typename F>
  OpCounter<F>& operator+=(OpCounter<F>& a, const F& b)
  {
```

```
  ++OpCounter<F>::counters.addition_count;
  a._v += b;
  return a;
}

template<typename F, typename T>
typename std::enable_if<
  std::is_arithmetic<T>::value,
  OpCounter<F>&
  >::type
operator+=(OpCounter<F>& a, const T& b)
{
  ++OpCounter<F>::counters.addition_count;
  a._v += b;
  return a;
}

// ********************************************************************************
// subtraction
// ********************************************************************************

template<typename F>
OpCounter<F> operator-(const OpCounter<F>& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.addition_count;
  return {a._v - b._v};
}

template<typename F>
OpCounter<F> operator-(const OpCounter<F>& a, const F& b)
{
  ++OpCounter<F>::counters.addition_count;
  return {a._v - b};
}

template<typename F>
OpCounter<F> operator-(const F& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.addition_count;
  return {a - b._v};
}

template<typename F, typename T>
typename std::enable_if<
  std::is_arithmetic<T>::value,
  OpCounter<F>
  >::type
operator-(const OpCounter<F>& a, const T& b)
{
  ++OpCounter<F>::counters.addition_count;
  return {a._v - b};
}

template<typename F, typename T>
typename std::enable_if<
  std::is_arithmetic<T>::value,
  OpCounter<F>
  >::type
operator-(const T& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.addition_count;
  return {a - b._v};
}

template<typename F>
OpCounter<F>& operator-=(OpCounter<F>& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.addition_count;
  a._v -= b._v;
  return a;
}

template<typename F>
OpCounter<F>& operator-=(OpCounter<F>& a, const F& b)
{
  ++OpCounter<F>::counters.addition_count;
  a._v -= b;
  return a;
}

template<typename F, typename T>
typename std::enable_if<
  std::is_arithmetic<T>::value,
  OpCounter<F>&
  >::type
operator-=(OpCounter<F>& a, const T& b)
{
```

```
    ++OpCounter<F>::counters.addition_count;
    a._v -= b;
    return a;
  }


  // ********************************************************************************
  // multiplication
  // ********************************************************************************

  template<typename F>
  OpCounter<F> operator*(const OpCounter<F>& a, const OpCounter<F>& b)
  {
    ++OpCounter<F>::counters.multiplication_count;
    return {a._v * b._v};
  }

  template<typename F>
  OpCounter<F> operator*(const OpCounter<F>& a, const F& b)
  {
    ++OpCounter<F>::counters.multiplication_count;
    return {a._v * b};
  }

  template<typename F>
  OpCounter<F> operator*(const F& a, const OpCounter<F>& b)
  {
    ++OpCounter<F>::counters.multiplication_count;
    return {a * b._v};
  }

  template<typename F, typename T>
  typename std::enable_if<
    std::is_arithmetic<T>::value,
    OpCounter<F>
    >::type
  operator*(const OpCounter<F>& a, const T& b)
  {
    ++OpCounter<F>::counters.multiplication_count;
    return {a._v * b};
  }

  template<typename F, typename T>
  typename std::enable_if<
    std::is_arithmetic<T>::value,
    OpCounter<F>
    >::type
  operator*(const T& a, const OpCounter<F>& b)
  {
    ++OpCounter<F>::counters.multiplication_count;
    return {a * b._v};
  }

  template<typename F>
  OpCounter<F>& operator*=(OpCounter<F>& a, const OpCounter<F>& b)
  {
    ++OpCounter<F>::counters.multiplication_count;
    a._v *= b._v;
    return a;
  }

  template<typename F>
  OpCounter<F>& operator*=(OpCounter<F>& a, const F& b)
  {
    ++OpCounter<F>::counters.multiplication_count;
    a._v *= b;
    return a;
  }

  template<typename F, typename T>
  typename std::enable_if<
    std::is_arithmetic<T>::value,
    OpCounter<F>&
    >::type
  operator*=(OpCounter<F>& a, const T& b)
  {
    ++OpCounter<F>::counters.multiplication_count;
    a._v *= b;
    return a;
  }


  // ********************************************************************************
  // division
  // ********************************************************************************

  template<typename F>
```

```
OpCounter<F> operator/(const OpCounter<F>& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.division_count;
  return {a._v / b._v};
}

template<typename F>
OpCounter<F> operator/(const OpCounter<F>& a, const F& b)
{
  ++OpCounter<F>::counters.division_count;
  return {a._v / b};
}

template<typename F>
OpCounter<F> operator/(const F& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.division_count;
  return {a / b._v};
}

template<typename F, typename T>
typename std::enable_if<
  std::is_arithmetic<T>::value,
  OpCounter<F>
  >::type
operator/(const OpCounter<F>& a, const T& b)
{
  ++OpCounter<F>::counters.division_count;
  return {a._v / b};
}

template<typename F, typename T>
typename std::enable_if<
  std::is_arithmetic<T>::value,
  OpCounter<F>
  >::type
operator/(const T& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.division_count;
  return {a / b._v};
}

template<typename F>
OpCounter<F>& operator/=(OpCounter<F>& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.division_count;
  a._v /= b._v;
  return a;
}

template<typename F>
OpCounter<F>& operator/=(OpCounter<F>& a, const F& b)
{
  ++OpCounter<F>::counters.division_count;
  a._v /= b;
  return a;
}

template<typename F, typename T>
typename std::enable_if<
  std::is_arithmetic<T>::value,
  OpCounter<F>&
  >::type
operator/=(OpCounter<F>& a, const T& b)
{
  ++OpCounter<F>::counters.division_count;
  a._v /= b;
  return a;
}



// ********************************************************************************
// comparisons
// ********************************************************************************


// ********************************************************************************
// less
// ********************************************************************************

template<typename F>
bool operator<(const OpCounter<F>& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a._v < b._v};
}
```

```
  template<typename F>
  bool operator<(const OpCounter<F>& a, const F& b)
  {
    ++OpCounter<F>::counters.comparison_count;
    return {a._v < b};
  }

  template<typename F>
  bool operator<(const F& a, const OpCounter<F>& b)
  {
    ++OpCounter<F>::counters.comparison_count;
    return {a < b._v};
  }

  template<typename F, typename T>
  bool operator<(const OpCounter<F>& a, const T& b)
  {
    ++OpCounter<F>::counters.comparison_count;
    return {a._v < b};
  }

  template<typename F, typename T>
  bool operator<(const T& a, const OpCounter<F>& b)
  {
    ++OpCounter<F>::counters.comparison_count;
    return {a < b._v};
  }


  // ********************************************************************************
  // less_or_equals
  // ********************************************************************************

  template<typename F>
  bool operator<=(const OpCounter<F>& a, const OpCounter<F>& b)
  {
    ++OpCounter<F>::counters.comparison_count;
    return {a._v <= b._v};
  }

  template<typename F>
  bool operator<=(const OpCounter<F>& a, const F& b)
  {
    ++OpCounter<F>::counters.comparison_count;
    return {a._v <= b};
  }

  template<typename F>
  bool operator<=(const F& a, const OpCounter<F>& b)
  {
    ++OpCounter<F>::counters.comparison_count;
    return {a <= b._v};
  }

  template<typename F, typename T>
  bool operator<=(const OpCounter<F>& a, const T& b)
  {
    ++OpCounter<F>::counters.comparison_count;
    return {a._v <= b};
  }

  template<typename F, typename T>
  bool operator<=(const T& a, const OpCounter<F>& b)
  {
    ++OpCounter<F>::counters.comparison_count;
    return {a <= b._v};
  }


  // ********************************************************************************
  // greater
  // ********************************************************************************

  template<typename F>
  bool operator>(const OpCounter<F>& a, const OpCounter<F>& b)
  {
    ++OpCounter<F>::counters.comparison_count;
    return {a._v > b._v};
  }

  template<typename F>
  bool operator>(const OpCounter<F>& a, const F& b)
  {
    ++OpCounter<F>::counters.comparison_count;
    return {a._v > b};
  }
```

```

template<typename F>
bool operator>(const F& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a > b._v};
}

template<typename F, typename T>
bool operator>(const OpCounter<F>& a, const T& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a._v > b};
}

template<typename F, typename T>
bool operator>(const T& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a > b._v};
}


// ********************************************************************************
// greater_or_equals
// ********************************************************************************

template<typename F>
bool operator>=(const OpCounter<F>& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a._v >= b._v};
}

template<typename F>
bool operator>=(const OpCounter<F>& a, const F& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a._v >= b};
}

template<typename F>
bool operator>=(const F& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a >= b._v};
}

template<typename F, typename T>
bool operator>=(const OpCounter<F>& a, const T& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a._v >= b};
}

template<typename F, typename T>
bool operator>=(const T& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a >= b._v};
}


// ********************************************************************************
// inequals
// ********************************************************************************

template<typename F>
bool operator!=(const OpCounter<F>& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a._v != b._v};
}

template<typename F>
bool operator!=(const OpCounter<F>& a, const F& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a._v != b};
}

template<typename F>
bool operator!=(const F& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a != b._v};
}
```

```
template<typename F, typename T>
bool operator!=(const OpCounter<F>& a, const T& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a._v != b};
}

template<typename F, typename T>
bool operator!=(const T& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a != b._v};
}


// *******************************************************************************
// equals
// *******************************************************************************

template<typename F>
bool operator==(const OpCounter<F>& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a._v == b._v};
}

template<typename F>
bool operator==(const OpCounter<F>& a, const F& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a._v == b};
}

template<typename F>
bool operator==(const F& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a == b._v};
}

template<typename F, typename T>
bool operator==(const OpCounter<F>& a, const T& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a._v == b};
}

template<typename F, typename T>
bool operator==(const T& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.comparison_count;
  return {a == b._v};
}



// *******************************************************************************
// functions
// *******************************************************************************

template<typename F>
OpCounter<F> exp(const OpCounter<F>& a)
{
  ++OpCounter<F>::counters.exp_count;
  return {std::exp(a._v)};
}

template<typename F>
OpCounter<F> pow(const OpCounter<F>& a, const OpCounter<F>& b)
{
  ++OpCounter<F>::counters.pow_count;
  return {std::pow(a._v,b._v)};
}

template<typename F>
OpCounter<F> pow(const OpCounter<F>& a, const F& b)
{
  ++OpCounter<F>::counters.pow_count;
  return {std::pow(a._v,b)};
}

template<typename F, typename T>
OpCounter<F> pow(const OpCounter<F>& a, const T& b)
{
  ++OpCounter<F>::counters.pow_count;
  return {std::pow(a._v,b)};
```

```
    }

    template<typename F>
    OpCounter<F> pow(const F& a, const OpCounter<F>& b)
    {
      ++OpCounter<F>::counters.pow_count;
      return {std::pow(a,b._v)};
    }

    template<typename F, typename T>
    OpCounter<F> pow(const T& a, const OpCounter<F>& b)
    {
      ++OpCounter<F>::counters.pow_count;
      return {std::pow(a,b._v)};
    }

    template<typename F>
    OpCounter<F> sin(const OpCounter<F>& a)
    {
      ++OpCounter<F>::counters.sin_count;
      return {std::sin(a._v)};
    }

    template<typename F>
    OpCounter<F> cos(const OpCounter<F>& a)
    {
      ++OpCounter<F>::counters.sin_count;
      return {std::cos(a._v)};
    }

    template<typename F>
    OpCounter<F> sqrt(const OpCounter<F>& a)
    {
      ++OpCounter<F>::counters.sqrt_count;
      return {std::sqrt(a._v)};
    }

    template<typename F>
    OpCounter<F> abs(const OpCounter<F>& a)
    {
      ++OpCounter<F>::counters.comparison_count;
      return {std::abs(a._v)};
    }
  }
}

#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.

```
// -*- tab-width: 4; indent-tabs-mode: nil -*-
#ifndef HDNUM_PDE_HH
#define HDNUM_PDE_HH

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

namespace hdnum {

  template<class M>
  class StationarySolver
  {
  public:
    typedef typename M::size_type size_type;

    typedef typename M::time_type time_type;

    typedef typename M::number_type number_type;

    StationarySolver (const M& model_)
      : model(model_), x(model.size())
    {
    }

    void solve ()
    {
      const size_t n_dofs = model.size();

      DenseMatrix<number_type> A(n_dofs,n_dofs,0.);
      Vector<number_type> b(n_dofs,0.);

      Vector<number_type> s(n_dofs);              // scaling factors
      Vector<size_t> p(n_dofs);                  // row permutations
      Vector<size_t> q(n_dofs);                  // column permutations

      number_type t = 0.;

      x = 0.;

      model.f_x(t, x, A);
      model.f(t, x, b);

      b*=-1.;

      row_equilibrate(A,s);                      // equilibrate rows
      lr_fullpivot(A,p,q);                       // LR decomposition of A
      apply_equilibrate(s,b);                    // equilibration of right hand side
      permute_forward(p,b);                      // permutation of right hand side
      solveL(A,b,b);                             // forward substitution
      solveR(A,x,b);                             // backward substitution
      permute_backward(q,x);                     // backward permutation
    }

    const Vector<number_type>& get_state () const
    {
      return x;
    }

    size_type get_order () const
    {
      return 2;
    }

  private:
    const M& model;
    Vector<number_type> x;
  };


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

```
  {

    const std::vector<Vector<N> > coords = grid.getNodeCoordinates();
    Vector<typename G::size_type> gsize = grid.getGridSize();

    std::fstream f(fname.c_str(),std::ios::out);
    // f « "set dgrid3d ";

     // f « gsize[0] « "," « gsize[1] « std::endl;

     // f « "set hidden3d" « std::endl;
     f « "set ticslevel 0" « std::endl;
     f « "splot \"-\" using 1:2:3 with points" « std::endl;
     f « "#" « std::endl;
     for (typename Vector<N>::size_type n=0; n<solution.size(); n++)
       {
         for (typename Vector<N>::size_type d=0; d<coords[n].size(); d++){
           f « std::scientific « std::showpoint
             « std::setprecision(16) « coords[n][d] « " ";
         }

         f « std::scientific « std::showpoint
           « std::setprecision(solution.precision()) « solution[n];

         f « std::endl;
       }
     f « "end" « std::endl;
     f « "pause -1" « std::endl;
     f.close();
   }


}
#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.

```
// -*- tab-width: 4; indent-tabs-mode: nil -*-
#ifndef HDNUM_PRECISION_HH
#define HDNUM_PRECISION_HH

namespace hdnum {

   // find largest eps such that 0.5 + eps > 0.5
   template<typename X>
   int precision (X& eps)
   {
         X x,large,largex,two;
         large = 0.5;
         two = 2.0;
         x = 0.5;
         largex = large+x;
```

```
        int i(0);
        while (largex>large)
          {
            eps = x;
            i = i+1;
            //              std::cout « i « " " « std::scientific « std::showpoint
            //                « std::setprecision(15) « large+x « " " « x « std::endl;
            x = x/two;
            largex = large+x;
          }
        return i;
  }

} // namespace hdnum

#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.

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
/*
 * File:   qr.hh
 * Author: Raphael Vogt <cx238@stud.uni-heidelberg.de>
 *
 * Created on August 30, 2020
 */

#ifndef HDNUM_QR_HH
#define HDNUM_QR_HH

#include <cmath>
#include <utility>

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

namespace hdnum {

template <class T>
DenseMatrix<T> gram_schmidt(const DenseMatrix<T>& A) {
    DenseMatrix<T> Q(A);

    // for all columns except the first
    for (int k = 1; k < Q.colsize(); k++) {
        // orthogonalize column k against all previous
        for (int j = 0; j < k; j++) {
            // compute factor
            T sum_nom(0.0);
            T sum_denom(0.0);
            for (int i = 0; i < Q.rowsize(); i++) {
                sum_nom += A[i][k] * Q[i][j];
                sum_denom += Q[i][j] * Q[i][j];
            }
            // modify
            T alpha = sum_nom / sum_denom;
            for (int i = 0; i < Q.rowsize(); i++) Q[i][k] -= alpha * Q[i][j];
        }
    }
    for (int j = 0; j < Q.colsize(); j++) {
        // compute norm of column j
        T sum(0.0);
        for (int i = 0; i < Q.rowsize(); i++) sum += Q[i][j] * Q[i][j];
        sum = sqrt(sum);
        // scale
        for (int i = 0; i < Q.rowsize(); i++) Q[i][j] = Q[i][j] / sum;
```

```
    }
    return Q;
}

template <class T>
DenseMatrix<T> modified_gram_schmidt(const DenseMatrix<T>& A) {
    DenseMatrix<T> Q(A);

    for (int k = 0; k < Q.colsize(); k++) {
        // modify all later columns with column k
        for (int j = k + 1; j < Q.colsize(); j++) {
            // compute factor
            T sum_nom(0.0);
            T sum_denom(0.0);
            for (int i = 0; i < Q.rowsize(); i++) {
                sum_nom += Q[i][j] * Q[i][k];
                sum_denom += Q[i][k] * Q[i][k];
            }
            // modify
            T alpha = sum_nom / sum_denom;
            for (int i = 0; i < Q.rowsize(); i++) Q[i][j] -= alpha * Q[i][k];
        }
    }
    for (int j = 0; j < Q.colsize(); j++) {
        // compute norm of column j
        T sum(0.0);
        for (int i = 0; i < Q.rowsize(); i++) sum += Q[i][j] * Q[i][j];
        sum = sqrt(sum);
        // scale
        for (int i = 0; i < Q.rowsize(); i++) Q[i][j] = Q[i][j] / sum;
    }
    return Q;
}

template <class T>
DenseMatrix<T> qr_gram_schmidt_simple(DenseMatrix<T>& Q) {
    // save matrix A, before it's replaced with Q
    DenseMatrix<T> A(Q);

    // create matrix R
    DenseMatrix<T> R(Q.colsize(), Q.colsize());

    // start orthogonalizing
    for (int k = 0; k < Q.colsize(); k++) {
        // modify all later columns with column k
        for (int j = k + 1; j < Q.colsize(); j++) {
            // compute factor
            T sum_nom(0.0);
            T sum_denom(0.0);
            for (int i = 0; i < Q.rowsize(); i++) {
                sum_nom += Q(i, j) * Q(i, k);
                sum_denom += Q(i, k) * Q(i, k);
            }

            T alpha = sum_nom / sum_denom;
            for (int i = 0; i < Q.rowsize(); i++) Q(i, j) -= alpha * Q(i, k);
        }
    }

    // add values to R, except main diagonal
    for (int i = 1; i < R.colsize(); i++) {
        for (int j = 0; j < i; j++) {
            T sum_nom(0.0);
            T sum_l2nom(0.0);
            for (int k = 0; k < Q.rowsize(); k++) {
                sum_nom += A(k, i) * Q(k, j);
                sum_l2nom += Q(k, j) * Q(k, j);
            }
            sum_l2nom = sqrt(sum_l2nom);
            // add element
            R(j, i) = sum_nom / sum_l2nom;
        }
    }

    // add missing values and scale
    for (int j = 0; j < Q.colsize(); j++) {
        // compute norm of column j
        T sum(0.0);
        for (int i = 0; i < Q.rowsize(); i++) sum += Q(i, j) * Q(i, j);
        sum = sqrt(sum);
        // add main diagonal to R
        R(j, j) = sum;
        // scale Q
        for (int i = 0; i < Q.rowsize(); i++) Q(i, j) = Q(i, j) / sum;
    }
    return R;
}
```

```
template <class T>
DenseMatrix<T> qr_gram_schmidt(DenseMatrix<T>& Q) {
    // create matrix R
    DenseMatrix<T> R(Q.colsize(), Q.colsize());

    // start orthogonalizing
    for (int k = 0; k < Q.colsize(); k++) {
        // compute norm of column k
        T sum_denom(0.0);
        for (int i = 0; i < Q.rowsize(); i++) {
            sum_denom += Q(i, k) * Q(i, k);
        }

        // fill the main diagonal of R with elements
        sum_denom = sqrt(sum_denom);
        R(k, k) = sum_denom;

        // scale column k to the main diagonal
        for (int i = 0; i < Q.rowsize(); i++) {
            Q(i, k) /= R(k, k);
        }

        // modify all later columns with column k
        for (int j = k + 1; j < Q.colsize(); j++) {
            // compute norm of column j
            T sum_nom(0.0);
            for (int i = 0; i < Q.rowsize(); i++) {
                sum_nom += Q(i, k) * Q(i, j);
            }
            // insert missing elements to R
            R(k, j) = sum_nom;

            // orthogonalize column j
            for (int i = 0; i < Q.rowsize(); i++) {
                Q(i, j) -= Q(i, k) * R(k, j);
            }
        }
    }
    return R;
}

template <class T>
DenseMatrix<T> qr_gram_schmidt_pivoting(DenseMatrix<T>& Q, Vector<int>& p, int& rank, T
   threshold=0.00000000001) {
    // check if permutation vector has the right size
    if (p.size() != Q.colsize()) {
        HDNUM_ERROR("Permutation Vector incompatible with Matrix!");
    }

    // initialize permutation vector
    for (int i = 0; i < p.size(); i++) {
        p[i] = i;
    }

    // initialize rank
    rank = 0;

    // save matrix A, before it's replaced with Q
    DenseMatrix<T> A(Q);

    // create Matrix R
    hdnum::DenseMatrix<T> R(A.colsize(), A.colsize());

    // start orthogonalizing
    for (int k = 0; k < Q.colsize(); k++) {
        // find column with highest norm
        // compute norm of column k
        T norm_k(0.0);
        for (int r = 0; r < Q.rowsize(); r++) {
                norm_k += Q(r, k) * Q(r, k);
            }
        norm_k = sqrt(norm_k);

        // compare norm of column k to the following column norms
        for (int c = k+1; c < Q.colsize(); c++) {
            T norm(0.0);
            for (int r = 0; r < Q.rowsize(); r++) {
                norm += Q(r, c) * Q(r, c);
            }
            norm = sqrt(norm);
            // store permutation
            if (norm > norm_k) {
                p[k] = c;
            }
        }

```

```
        // swap columns if necessary
        if (p[k] > k) {
            for (int r = 0; r < Q.rowsize(); r++) {
                T temp_Q = Q(r, k);
                Q(r, k) = Q(r, p[k]);
                Q(r, p[k]) = temp_Q;
            }
            p[p[k]] = k;

            // compute norm of the new column k
            norm_k = 0;
            for (int i = 0; i < Q.rowsize(); i++) {
                norm_k += Q(i, k) * Q(i, k);
            }
            norm_k = sqrt(norm_k);
        }

        // if norm of column k > threshold -> column k is linear independent
        if (norm_k > threshold) {
            rank++;
        } else {
            break;
        }

        // modify all later columns with column k
        for (int j = k + 1; j < Q.colsize(); j++) {
            // compute factor
            T sum_nom(0.0);
            T sum_denom(0.0);
            for (int i = 0; i < Q.rowsize(); i++) {
                sum_nom += Q(i, j) * Q(i, k);
                sum_denom += Q(i, k) * Q(i, k);
            }

            T alpha = sum_nom / sum_denom;
            for (int i = 0; i < Q.rowsize(); i++) Q(i, j) -= alpha * Q(i, k);
        }
    }

    // add values to R, except main diagonal
    for (int i = 1; i < R.colsize(); i++) {
        for (int j = 0; j < i; j++) {
            T sum_nom(0.0);
            T sum_l2nom(0.0);
            for (int k = 0; k < Q.rowsize(); k++) {
                sum_nom += A(k, p[i]) * Q(k, j);
                sum_l2nom += Q(k, j) * Q(k, j);
            }
            sum_l2nom = sqrt(sum_l2nom);
            // add element
            R(j, i) = sum_nom / sum_l2nom;
        }
    }

    // add missing values and scale
    for (int j = 0; j < Q.colsize(); j++) {
        // compute norm of column j
        T sum(0.0);
        for (int i = 0; i < Q.rowsize(); i++) sum += Q(i, j) * Q(i, j);
        sum = sqrt(sum);
        // add main diagonal to R
        R(j, j) = sum;
        // scale Q
        for (int i = 0; i < Q.rowsize(); i++) Q(i, j) = Q(i, j) / sum;
    }

    return R;
}

template<typename T>
void permute_forward(DenseMatrix<T>& A, Vector<int>& p) {
    // check if permutation vector has the right size
    if (p.size() != A.colsize()) {
        HDNUM_ERROR("Permutation Vector incompatible with Matrix!");
    }

    // permutate the columns
    for (int k = 0; k < p.size(); k++) {
        if (p[k] != k) {
            // swap column
            for (int j=0; j < A.rowsize(); j++) {
                T temp_A = A(j, k);
                A(j, k) = A(j, p[k]);
                A(j, p[k]) = temp_A;
            }
            // swap inside permutation vector
            int temp_p = p[k];
```

```
            p[k] = p[temp_p];
            p[temp_p] = temp_p;
        }
    }
}

}  // namespace hdnum

#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.

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
/*
 * File:   qrhousholder
 * Author: Ahmad Fadl <abohmaid@windowslive.com>
 *
 * Created on August 25, 2020
 */

#ifndef HDNUM_QRHOUSHOLDER_HH
#define HDNUM_QRHOUSHOLDER_HH
#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>

#include "densematrix.hh"
#include "vector.hh"
namespace hdnum {
template <class REAL>
DenseMatrix<REAL> creat_I_matrix(size_t n) {
    DenseMatrix<REAL> res(n, n, 0);
    for (size_t i = 0; i < n; i++) {
        res(i, i) = 1;
    }
    return res;
}

```

```
template <typename REAL>
size_t sgn(REAL val) {
    return (REAL(0) < val) - (val < REAL(0));
}
template <class REAL>
void qrhousholder(DenseMatrix<REAL>& A, hdnum::Vector<REAL>& v) {
    auto m = A.rowsize();
    auto n = A.colsize();
    for (size_t j = 0; j < n; j++) {
        REAL s = 0;
        for (size_t i = j; i < m; i++) {
            s = s + pow(A(i, j), 2);
        }
        s = sqrt(s);
        v[j] = (-1.0) * sgn(A(j, j)) * s;
        REAL fak = sqrt(s * (s + std::abs(A(j, j))));
        A(j, j) = A(j, j) - v[j];
        for (size_t k = j; k < m; k++) {
            A(k, j) = A(k, j) / fak;
        }
        for (size_t i = j + 1; i < n; i++) {
            s = 0;
            for (size_t k = j; k < m; k++) {
                s = s + (A(k, j) * A(k, i));
            }
            for (size_t k = j; k < m; k++) {
                A(k, i) = A(k, i) - (A(k, j) * s);
            }
        }
        // normalize the vi vectors again
        for (size_t i = m; i >= 0; i--) {
            A(i, j) = A(i, j) * fak;
            if (i == j) {
                break;
            }
        }
    }
}
template <class REAL>
DenseMatrix<REAL> qrhousholderexplizitQ(DenseMatrix<REAL>& A,
                                        hdnum::Vector<REAL>& v,
                                        bool show_Hi = false) {
    auto m = A.rowsize();
    auto n = A.colsize();
    auto I = creat_I_matrix<REAL>(std::max(m, n));

    DenseMatrix<REAL> Q(m, m, 0);
    for (size_t j = 0; j < n; j++) {
        REAL s = 0;
        for (size_t i = j; i < m; i++) {
            s = s + pow(A(i, j), 2);
        }
        s = sqrt(s);
        v[j] = (-1.0) * sgn(A(j, j)) * s;
        REAL fak = sqrt(s * (s + std::abs(A(j, j))));
        A(j, j) = A(j, j) - v[j];
        for (size_t k = j; k < m; k++) {
            A(k, j) = A(k, j) / fak;
        }
        for (size_t i = j + 1; i < n; i++) {
            s = 0;
            for (size_t k = j; k < m; k++) {
                s = s + (A(k, j) * A(k, i));
            }
            for (size_t k = j; k < m; k++) {
                A(k, i) = A(k, i) - (A(k, j) * s);
            }
        }
        // normalize the vi vectors again
        for (size_t i = m; i >= 0; i--) {
            A(i, j) = A(i, j) * fak;
            if (i == j) {
                break;
            }
        }
    }
    // create qi and multiply them
    if (m >= n) {
        for (size_t j = 0; j < n; j++) {
            DenseMatrix<REAL> TempQ(m, m, 0.0);
            DenseMatrix<REAL> v1(m, 1, 0.0);
            DenseMatrix<REAL> v1t(1, m, 0.0);
            hdnum::Vector<double> v__i(m, 0);
            for (size_t i = 0; i < m; i++) {
                if (i < j) {
                    v1(i, 0) = 0;

```

```
                    v__i[i] = 0;
                    continue;
                }
                v1(i, 0) = A(i, j);

                v__i[i] = A(i, j);
            }
            v1t = v1.transpose();

            TempQ = (v1 * v1t);

            TempQ *= (-2.0);

            TempQ /= v__i.two_norm_2();

            TempQ += I;
            if (show_Hi) {
                std::cout « "H[" « j + 1 « "]" « TempQ;
            }
            if (j == 0) {
                Q = TempQ;
            }
            if (j > 0) {
                Q = Q * TempQ;
            }
        }
    }
    if (n > m) {
        for (size_t j = 0; j < m; j++) {
            DenseMatrix<REAL> TempQ(m, m, 0.0);
            DenseMatrix<REAL> v1(m, 1, 0.0);
            DenseMatrix<REAL> v1t(1, m, 0.0);
            hdnum::Vector<double> v__i(m, 0);
            for (size_t i = 0; i < m; i++) {
                if (i < j) {
                    v1(i, 0) = 0;

                    v__i[i] = 0;
                    continue;
                }
                v1(i, 0) = A(i, j);

                v__i[i] = A(i, j);
            }
            v1t = v1.transpose();

            TempQ = (v1 * v1t);

            TempQ *= (-2.0);

            TempQ /= v__i.two_norm_2();

            TempQ += I;
            if (show_Hi) {
                std::cout « "H[" « j + 1 « "]" « TempQ;
            }
            if (j == 0) {
                Q = TempQ;
            }
            if (j > 0) {
                Q = Q * TempQ;
            }
        }
    }
    return Q;
}
}  // namespace hdnum
#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.

```
// -*- tab-width: 4; indent-tabs-mode: nil -*-
#ifndef HDNUM_RUNGEKUTTA_HH
#define HDNUM_RUNGEKUTTA_HH

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

namespace hdnum {
  template<class M>
  class ImplicitRungeKuttaStepProblem
  {
  public:
    typedef typename M::size_type size_type;

    typedef typename M::time_type time_type;

    typedef typename M::number_type number_type;

    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_)
        : model(model_) , u(model.size())
      {
        A = A_;
        b = b_;
        c = c_;
        s = A_.rowsize ();
        dt = dt_;
        n = model.size();
        t = t_;
        u = u_;
      }

    std::size_t size () const
    {
      return n*s;
    }

    void F (const Vector<number_type>& x, Vector<number_type>& result) const
    {
      Vector<Vector<number_type> > xx (s);
      for (int i = 0; i < s; i++)
      {
        xx[i].resize(n,number_type(0));
        for(int k = 0; k < n; k++)
        {
          xx[i][k] = x[i*n + k];
        }
      }
      Vector<Vector<number_type> > f (s);
      for (int i = 0; i < s; i++)
      {
        f[i].resize(n, number_type(0));
        model.f(t + c[i] * dt, u + xx[i], f[i]);
      }
      Vector<Vector<number_type> > hr (s);
      for (int i = 0; i < s; i++)
      {
        hr[i].resize(n, number_type(0));
      }
      for (int i = 0; i < s; i++)
      {
        Vector<number_type> sum (n, number_type(0));
        for (int j = 0; j < s; j++)
        {
          sum.update(dt*A[i][j], f[j]);
        }
        hr[i]  = xx[i] - sum;
      }
      //translating hr into result
      for (int i = 0; i < s; i++)
      {
        for (int j = 0; j < n; j++)
        {
          result[i*n + j] = hr[i][j];
        }
      }
    }
```

```

  void F_x (const Vector<number_type>& x, DenseMatrix<number_type>& result) const
  {
    Vector<Vector<number_type> > xx (s);
     for (int i = 0; i < s; i++)
     {
       xx[i].resize(n);
       for(int k = 0; k < n; k++)
       {
         xx[i][k] = x[i*n + k];
       }
     }
     DenseMatrix<number_type> I (n, n, 0.0);
     for (int i = 0; i < n; i++)
     {
       I[i][i] = 1.0;
     }
     for (int i = 0; i < s; i++)
     {
       for (int j = 0; j < s; j++)
       {
         DenseMatrix<number_type> J (n, n, number_type(0));
         DenseMatrix<number_type> H (n, n, number_type(0));
         model.f_x(t+c[j]*dt, u + xx[j],H);
         J.update(-dt*A[i][j],H);
         if(i==j)                              //add I on diagonal
         {
           J+=I;
         }
         for (int k = 0; k < n; k++)
         {
           for (int l = 0; l < n; l++)
           {
             result[n * i + k][n * j + l] = J[k][l];
           }
         }
       }
     }
   }

private:
  const M& model;
  time_type t, dt;
  Vector<number_type> u;
  int n, s;                                         // dimension of matrix A and model.size
  DenseMatrix<number_type> A;                   // A, b, c as in the butcher tableau
  Vector<number_type> b;
  Vector<number_type> c;
};


template<class M, class S = Newton>
class RungeKutta
{
public:
  typedef typename M::size_type size_type;

  typedef typename M::time_type time_type;

  typedef typename M::number_type number_type;

  RungeKutta (const M& model_,
              DenseMatrix<number_type> A_,
              Vector<number_type> b_,
              Vector<number_type> c_)
    : model(model_), u(model.size()), w(model.size()), K(A_.rowsize ())
   {
     A = A_;
     b = b_;
     c = c_;
     s = A_.rowsize ();
     n = model.size();
     model.initialize(t,u);
     dt = 0.1;
     for (int i = 0; i < s; i++)
     {
       K[i].resize(n, number_type(0));
     }
     sigma = 0.01;
     verbosity = 0;

     if (A_.rowsize()!=A_.colsize())
     HDNUM_ERROR("need square and nonempty matrix");
     if (A_.rowsize()!=b_.size())
     HDNUM_ERROR("vector incompatible with matrix");
     if (A_.colsize()!=c_.size())
     HDNUM_ERROR("vector incompatible with matrix");
```

```
  }

  RungeKutta (const M& model_,
              DenseMatrix<number_type> A_,
              Vector<number_type> b_,
              Vector<number_type> c_,
              number_type sigma_)
    : model(model_), u(model.size()), w(model.size()), K(A_.rowsize ())
  {
    A = A_;
    b = b_;
    c = c_;
    s = A_.rowsize ();
    n = model.size();
    model.initialize(t,u);
    dt = 0.1;
    for (int i = 0; i < s; i++)
    {
      K[i].resize(n, number_type(0));
    }
    sigma = sigma_;
    verbosity = 0;
    if (A_.rowsize()!=A_.colsize())
    HDNUM_ERROR("need square and nonempty matrix");
    if (A_.rowsize()!=b_.size())
    HDNUM_ERROR("vector incompatible with matrix");
    if (A_.colsize()!=c_.size())
    HDNUM_ERROR("vector incompatible with matrix");
 }

 void set_dt (time_type dt_)
 {
   dt = dt_;
 }

 bool check_explicit ()
 {
   bool is_explicit = true;
   for (int i = 0; i < s; i++)
   {
     for (int j = i; j < s; j++)
     {
       if (A[i][j] != 0.0)
       {
         is_explicit = false;
       }
     }
   }
   return is_explicit;
 }

 void step ()
 {
   if (check_explicit())
   {
     // compute k_1
     w = u;
     model.f(t, w, K[0]);
     for (int i = 0; i < s; i++)
     {
       Vector<number_type> sum (K[0].size(), 0.0);
       sum.update(b[0], K[0]);
       //compute k_i
       for (int j = 0; j < i+1; j++)
       {
         sum.update(A[i][j],K[j]);
       }
       Vector<number_type> wert = w.update(dt,sum);
       model.f(t + c[i]*dt, wert, K[i]);
       u.update(dt *b[i], K[i]);
     }
   }
   if (not check_explicit())
   {
     // In the implicit case we need to solve a nonlinear problem
     // to do a time step.
     ImplicitRungeKuttaStepProblem<M> problem(model, A, b, c, t, u, dt);
     bool last_row_eq_b = true;
     for (int i = 0; i<s; i++)
     {
       if (A[s-1][i] != b[i])
       {
         last_row_eq_b = false;
       }
     }

     // Solve nonlinear problem and determine coefficients
```

```
     S solver;
     solver.set_maxit(2000);
     solver.set_verbosity(verbosity);
     solver.set_reduction(1e-10);
     solver.set_abslimit(1e-10);
     solver.set_linesearchsteps(10);
     solver.set_sigma(0.01);
     Vector<number_type> zij (s*n,0.0);
     solver.solve(problem,zij);


     DenseMatrix<number_type> Ainv (s,s,number_type(0));
     if (not last_row_eq_b)
     {
       // Compute LR decomposition of A
       Vector<number_type> w (s, number_type(0));
       Vector<number_type> x (s, number_type(0));
       Vector<number_type> z (s, number_type(0));
       Vector<std::size_t> p(s);
       Vector<std::size_t> q(s);
       DenseMatrix<number_type> Temp (s,s,0.0);
       Temp = A;
       row_equilibrate(Temp,w);
       lr_fullpivot(Temp,p,q);

       // Use LR decomposition to calculate inverse of A
       for (int i=0; i<s; i++)
       {
         Vector<number_type> e (s, number_type(0));
         e[i]=number_type(1);
         apply_equilibrate(w,e);
         permute_forward(p,e);
         solveL(Temp,e,e);
         solveR(Temp,z,e);
         permute_backward(q,z);
         for (int j = 0; j < s; j++)
         {
               Ainv[j][i] = z[j];
         }
       }
     }

     Vector<Vector<number_type> > Z (s, 0.0);
     for(int i=0; i<s; i++)
     {
       Vector<number_type> zero(n,number_type(0));
       Z[i] = zero;
       for (int j = 0; j < n; j++)
       {
         Z[i][j] = zij[i*n+j];
       }
     }
     if (last_row_eq_b)
     {
       u += Z[s-1];
     }
     else
     {
       // compute ki
       Vector<number_type> zero(n,number_type(0));
       for (int i = 0; i < s; i++)
       {
         K[i] = zero;
         for (int j=0; j < s; j++)
         {
           K[i].update(Ainv[i][j],Z[j]);
         }
         K[i]*= (1.0/dt);

         // compute u
         u.update(dt*b[i], K[i]);
       }
     }
   }
     t = t+dt;
  }

  void set_state (time_type t_, const Vector<number_type>& u_)
  {
    t = t_;
    u = u_;
  }

  const Vector<number_type>& get_state () const
  {
    return u;
  }
```

```
  time_type get_time () const
  {
    return t;
  }

  time_type get_dt () const
  {
    return dt;
  }

  void set_verbosity(int verbosity_)
  {
    verbosity = verbosity_;
  }

private:
  const M& model;
  time_type t, dt;
  Vector<number_type> u;
  Vector<number_type> w;
  Vector<Vector<number_type> > K;                          // save ki
  int n;                                                                                    // dimension of matrix A
  int s;
  DenseMatrix<number_type> A;                                          // A, b, c as in the butcher tableau
      Vector<number_type> b;
      Vector<number_type> c;
  number_type sigma;
  int verbosity;
};


template<class M, class S>
void ordertest(const M& model,
               S solver,
               typename M::number_type T,
               typename M::number_type h_0,
               int l)
{
  // Get types
  typedef typename M::time_type time_type;
  typedef typename M::number_type number_type;

  // error_array[i] = ||u(T)-u_i(T)||
  number_type error_array[l];

  Vector<number_type> exact_solution;
  model.exact_solution(T, exact_solution);

  for (int i=0; i<l; i++)
  {
    // Set initial time and value
    time_type t_start;
    Vector<number_type> initial_solution(1);
    model.initialize(t_start, initial_solution);
    solver.set_state(t_start, initial_solution);

    // Initial time step
    time_type dt = h_0/pow(2,i) ;
    solver.set_dt(dt);

    // Time loop
    while (solver.get_time()<T-2*solver.get_dt())
    {
      solver.step();
    }

    // Last steps
    if (solver.get_time()<T-solver.get_dt())
    {
      solver.set_dt((T-solver.get_time())/2.0);
      for(int i=0; i<2; i++)
      {
        solver.step();
      }
    }
    else
    {
      solver.set_dt(T-solver.get_time());
      solver.step();
    }

    // Error
    Vector<number_type> state = solver.get_state();
    error_array[i] = norm(exact_solution-state);

```

```
      if(i==0)
      {
        std::cout « "dt: "
                  « std::scientific « std::showpoint « std::setprecision(8)
                  « dt
                  « "  "
                  « "Error: "
                  « error_array[0] « std::endl;
      }
      if(i>0)
      {
        number_type rate = log(error_array[i-1]/error_array[i])/log(2);
        std::cout « "dt: "
                  « std::scientific « std::showpoint « std::setprecision(8)
                  « dt
                  « "  "
                  « "Error: "
                  « error_array[i]
                  « "  "
                  « "Rate: "
                  « rate « std::endl;
      }
    }
  }

} // namespace hdnum

#endif
```

5.22 sgrid.hh

```
#ifndef HDNUM_SGRID_HH
#define HDNUM_SGRID_HH
#include <limits>
#include <assert.h>

namespace hdnum {
  template<class N, class DF, int dimension>
  class SGrid
  {
  public:

    typedef std::size_t size_type;

    typedef N number_type;

    typedef DF DomainFunction;

    enum { dim = dimension };

    static const int positive = 1;
    static const int negative = -1;


  private:

    const Vector<number_type> extent;
    const Vector<size_type>   size;
    const DomainFunction & df;
    Vector<number_type> h;
    Vector<size_type> offsets;
    std::vector<size_type> node_map;
    std::vector<size_type> grid_map;
    std::vector<bool> inside_map;
    std::vector<bool> boundary_map;

    size_t n_nodes;

    inline Vector<size_type> index2grid(size_type index) const
    {
      Vector<size_type> c(dim);
      for(int d=dim-1; d>=0; --d){
        c[d] = index / offsets[d];
        index -= c[d] * offsets[d];
      }
      return c;
    }

    inline Vector<number_type> grid2world(const Vector<size_type> & c) const
    {
      Vector<number_type> w(dim);
      for(int d=dim-1; d>=0; --d)
        w[d] = c[d] * h[d];
```

```
      return w;
    }

    inline Vector<number_type> index2world(size_type index) const
    {
      Vector<number_type> w(dim);
      Vector<size_type> c = index2grid(index);
      return grid2world(c);
    }


  public:

    const size_type invalid_node;

    SGrid(const Vector<number_type> extent_,
          const Vector<size_type> size_,
          const DomainFunction & df_)
      : extent(extent_), size(size_), df(df_),
        h(dim), offsets(dim),
        invalid_node(std::numeric_limits<size_type>::max())
    {
       // Determine total number of nodes, increment offsets, and cell
       // widths.
       n_nodes = 1;
       offsets.resize(dim);
       h.resize(dim);
       for(int d=0; d<dim; ++d){
         n_nodes *= size[d];
         offsets[d] = d==0 ? 1 : size[d-1] * offsets[d-1];
         h[d] = extent[d] / number_type(size[d]-1);
       }

       // Initialize maps.
       node_map.resize(0);
       inside_map.resize(n_nodes);
       grid_map.resize(n_nodes);
       boundary_map.resize(0);
       boundary_map.resize(n_nodes,false);

       for(size_type n=0; n<n_nodes; ++n){
         Vector<size_type> c = index2grid(n);
         Vector<number_type> x = grid2world(c);

         inside_map[n] = df.evaluate(x);
         if(inside_map[n]){
           node_map.push_back(n);
           grid_map[n] = node_map.size()-1;
         }
         else
           grid_map[n] = invalid_node;
       }

       // Find boundary nodes
       for(size_type n=0; n<node_map.size(); ++n){
         for(int d=0; d<dim; ++d){
           for(int s=0; s<2; ++s){
             const int side = s*2-1;
             const size_type neighbor = getNeighborIndex(n,d,side,1);
             if(neighbor == invalid_node)
               boundary_map[node_map[n]] = true;
           }
         }
       }

     }

     size_type getNeighborIndex(const size_type ln, const size_type n_dim, const int n_side, const int k = 1) const
     {
       const size_type n = node_map[ln];
       const Vector<size_type> c = index2grid(n);
       size_type neighbors[2];
       neighbors[0] = c[n_dim];
       neighbors[1] = size[n_dim]-c[n_dim]-1;

       assert(n_side == 1 || n_side == -1);
       if(size_type(k) > neighbors[(n_side+1)/2])
         return invalid_node;

       const size_type neighbor = n + offsets[n_dim] * n_side * k;

       if(!inside_map[neighbor])
         return invalid_node;

       return grid_map[neighbor];
     }
```

```
    bool isBoundaryNode(const size_type ln) const
    {
      return boundary_map[node_map[ln]];
    }

    size_type getNumberOfNodes() const
    {
      return node_map.size();
    }

    Vector<size_type> getGridSize() const
    {
      return size;
    }

    Vector<number_type> getCellWidth() const
    {
      return h;
    }

    Vector<number_type> getCoordinates(const size_type ln) const
    {
      return index2world(node_map[ln]);
    }

    std::vector<Vector<number_type> > getNodeCoordinates() const
    {
      std::vector<Vector<number_type> > coords;
      for(size_type n=0; n<node_map.size(); ++n){
        coords.push_back(Vector<number_type>(dim));
        coords.back() = index2world(node_map[n]);
      }
      return coords;
    }

  };

}

#endif // HDNUM_SGRID_HH
```

5.23 sparsematrix.hh

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
/*
 * File:   sparsematrix.hh
 * Author: Christian Heusel <christian@heusel.eu>
 *
 * Created on August 25, 2020
 */

#ifndef SPARSEMATRIX_HH
#define SPARSEMATRIX_HH

#include <algorithm>
#include <complex>
#include <functional>
#include <iomanip>
#include <iostream>
#include <map>
#include <numeric>
#include <string>
#include <type_traits>
#include <vector>

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

namespace hdnum {

template <typename REAL>
class SparseMatrix {
public:
    using size_type = std::size_t;

    using column_iterator = typename std::vector<REAL>::iterator;
    using const_column_iterator = typename std::vector<REAL>::const_iterator;

private:
    // Matrix data is stored in an STL vector!
    std::vector<REAL> _data;

```

```
    // The non-null indices are stored in STL vectors with the size_type!
    // Explanation on how the mapping works can be found here:
    // https://de.wikipedia.org/wiki/Compressed_Row_Storage
    std::vector<size_type> _colIndices;
    std::vector<size_type> _rowPtr;

    size_type m_rows = 0;  // Number of Matrix rows
    size_type m_cols = 0;  // Number of Matrix columns

    static bool bScientific;
    static size_type nIndexWidth;
    static size_type nValueWidth;
    static size_type nValuePrecision;
    static const REAL _zero;

    // !function that converts container contents into
    // { 1, 2, 3, 4 }
    template <typename T>
    [[nodiscard]] std::string comma_fold(T container) const {
        return "{ " +
               std::accumulate(
                   std::next(container.cbegin()), container.cend(),
                   std::to_string(container[0]),  // start with first element
                   [](const std::string &a, REAL b) {
                       return a + ", " + std::to_string(b);
                   }) +
               " }";
    }

    // This code was copied from StackOverflow to gerneralize a check whether a
    // template is a specialization i.e. for std::complex
    // https://stackoverflow.com/questions/31762958/check-if-class-is-a-template-specialization
    template <class T, template <class...> class Template>
    struct is_specialization : std::false_type {};

    template <template <class...> class Template, class... Args>
    struct is_specialization<Template<Args...>, Template> : std::true_type {};

    bool checkIfAccessIsInBounds(const size_type row_index,
                                 const size_type col_index) const {
        if (not (row_index < m_rows)) {
            HDNUM_ERROR("Out of bounds access: row too big! -> " +
                        std::to_string(row_index) + " is not < " +
                        std::to_string(m_rows));
            return false;
        }
        if (not (col_index < m_cols)) {
            HDNUM_ERROR("Out of bounds access: column too big! -> " +
                        std::to_string(col_index) + " is not < " +
                        std::to_string(m_cols));
            return false;
        }
        return true;
    }

public:
    SparseMatrix() = default;

    SparseMatrix(const size_type _rows, const size_type _cols)
        : _rowPtr(_rows + 1), m_rows(_rows), m_cols(_cols) {}

    [[nodiscard]] size_type rowsize() const { return m_rows; }

    [[nodiscard]] size_type colsize() const { return m_cols; }

    [[nodiscard]] bool scientific() const { return bScientific; }

    class column_index_iterator {
    public:
        using self_type = column_index_iterator;

        // conform to the iterator traits
        // https://en.cppreference.com/w/cpp/iterator/iterator_traits
        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;

        column_index_iterator(typename std::vector<REAL>::iterator valIter,
                              std::vector<size_type>::iterator colIndicesIter)
            : _valIter(valIter), _colIndicesIter(colIndicesIter) {}

        // prefix
        self_type &operator++() {
            _valIter++;
            _colIndicesIter++;
```

```
            return *this;
        }

        // postfix
        self_type &operator++(int junk) {
            self_type cached = *this;
            _valIter++;
            _colIndicesIter++;
            return cached;
        }

        [[nodiscard]] value_type operator*() {
            return std::make_pair(std::ref(*_valIter),
                                  std::cref(*_colIndicesIter));
        }
        // [[nodiscard]] value_type operator->() {
        //     return std::make_pair(std::ref(*_valIter),
        //                           std::cref(*_colIndicesIter));
        // }

        [[nodiscard]] typename value_type::first_type value() {
            return std::ref(*_valIter);
        }

        [[nodiscard]] typename value_type::second_type index() {
            return std::cref(*_colIndicesIter);
        }

        [[nodiscard]] bool operator==(const self_type &other) {
            return (_valIter == other._valIter) and
                   (_colIndicesIter == other._colIndicesIter);
        }
        [[nodiscard]] bool operator!=(const self_type &other) {
            return not (*this == other);
        }

    private:
        typename std::vector<REAL>::iterator _valIter;
        std::vector<size_type>::iterator _colIndicesIter;
    };

    class const_column_index_iterator {
    public:
        using self_type = const_column_index_iterator;

        // conform to the iterator traits
        // https://en.cppreference.com/w/cpp/iterator/iterator_traits
        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;

        const_column_index_iterator(
            typename std::vector<REAL>::const_iterator valIter,
            std::vector<size_type>::const_iterator colIndicesIter)
            : _valIter(valIter), _colIndicesIter(colIndicesIter) {}

        // prefix
        self_type &operator++() {
            _valIter++;
            _colIndicesIter++;
            return *this;
        }

        // postfix
        self_type operator++(int junk) {
            self_type cached = *this;
            _valIter++;
            _colIndicesIter++;
            return cached;
        }

        [[nodiscard]] value_type operator*() {
            return std::make_pair(std::ref(*_valIter),
                                  std::cref(*_colIndicesIter));
        }
        // TODO: This is wrong
        // [[nodiscard]] value_type operator->() {
        //     return std::make_pair(*_valIter, *_colIndicesIter);
        // }

        [[nodiscard]] typename value_type::first_type value() {
            return std::ref(*_valIter);
        }

        [[nodiscard]] typename value_type::second_type index() {
```

```
            return std::cref(*_colIndicesIter);
        }

        [[nodiscard]] bool operator==(const self_type &other) {
            return (_valIter == other._valIter) and
                   (_colIndicesIter == other._colIndicesIter);
        }
        [[nodiscard]] bool operator!=(const self_type &other) {
            return not (*this == other);
        }

    private:
        typename std::vector<REAL>::const_iterator _valIter;
        std::vector<size_type>::const_iterator _colIndicesIter;
    };

    class row_iterator {
    public:
        using self_type = row_iterator;

        // conform to the iterator traits
        // https://en.cppreference.com/w/cpp/iterator/iterator_traits
        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;

        row_iterator(std::vector<size_type>::iterator rowPtrIter,
                     std::vector<size_type>::iterator colIndicesIter,
                     typename std::vector<REAL>::iterator valIter)
            : _rowPtrIter(rowPtrIter), _colIndicesIter(colIndicesIter),
              _valIter(valIter) {}

        [[nodiscard]] column_iterator begin() {
            return column_iterator((_valIter + *_rowPtrIter));
        }
        [[nodiscard]] column_iterator end() {
            return column_iterator((_valIter + *(_rowPtrIter + 1)));
        }

        [[nodiscard]] column_index_iterator ibegin() {
            return column_index_iterator((_valIter + *_rowPtrIter),
                                         (_colIndicesIter + *_rowPtrIter));
        }
        [[nodiscard]] column_index_iterator iend() {
            return column_index_iterator(
                (_valIter + *(_rowPtrIter + 1)),
                (_colIndicesIter + *(_rowPtrIter + 1)));
        }

        // prefix
        self_type &operator++() {
            _rowPtrIter++;
            return *this;
        }

        // postfix
        self_type operator++(int junk) {
            self_type cached = *this;
            _rowPtrIter++;
            return cached;
        }

        self_type &operator+=(difference_type offset) {
            _rowPtrIter += offset;
            return *this;
        }

        self_type &operator-=(difference_type offset) {
            _rowPtrIter -= offset;
            return *this;
        }

        // iter - n
        self_type operator-(difference_type offset) {
            self_type cache(*this);
            cache -= offset;
            return cache;
        }

        // iter + n
        self_type operator+(difference_type offset) {
            self_type cache(*this);
            cache += offset;
            return cache;
        }
```

```
        // n + iter
        friend self_type operator+(const difference_type &offset,
                                   const self_type &sec) {
            self_type cache(sec);
            cache += offset;
            return cache;
        }

        reference operator[](difference_type offset) {
            return *(*this + offset);
        }

        bool operator<(const self_type &other) {
            return other - (*this) > 0;  //
        }

        bool operator>(const self_type &other) {
            return other < (*this);  //
        }

        [[nodiscard]] self_type &operator*() { return *this; }
        // [[nodiscard]] self_type &operator->() { return *this; }

        [[nodiscard]] bool operator==(const self_type &rhs) {
            return _rowPtrIter == rhs._rowPtrIter;
        }
        [[nodiscard]] bool operator!=(const self_type &rhs) {
            return _rowPtrIter != rhs._rowPtrIter;
        }

    private:
        std::vector<size_type>::iterator _rowPtrIter;
        std::vector<size_type>::iterator _colIndicesIter;
        typename std::vector<REAL>::iterator _valIter;
    };

    class const_row_iterator {
    public:
        using self_type = const_row_iterator;

        // conform to the iterator traits
        // https://en.cppreference.com/w/cpp/iterator/iterator_traits
        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;

        const_row_iterator(
            std::vector<size_type>::const_iterator rowPtrIter,
            std::vector<size_type>::const_iterator colIndicesIter,
            typename std::vector<REAL>::const_iterator valIter)
            : _rowPtrIter(rowPtrIter), _colIndicesIter(colIndicesIter),
               _valIter(valIter) {}

        [[nodiscard]] const_column_iterator begin() const {
            return const_column_iterator((_valIter + *_rowPtrIter));
        }
        [[nodiscard]] const_column_iterator end() const {
            return const_column_iterator((_valIter + *(_rowPtrIter + 1)));
        }

        [[nodiscard]] const_column_index_iterator ibegin() const {
            return const_column_index_iterator(
                (_valIter + *_rowPtrIter), (_colIndicesIter + *_rowPtrIter));
        }
        [[nodiscard]] const_column_index_iterator iend() const {
            return const_column_index_iterator(
                (_valIter + *(_rowPtrIter + 1)),
                (_colIndicesIter + *(_rowPtrIter + 1)));
        }

        [[nodiscard]] const_column_iterator cbegin() const {
            return this->begin();
        }
        [[nodiscard]] const_column_iterator cend() const {
            return this->end();  //
        }

        // prefix
        self_type &operator++() {
            _rowPtrIter++;
            return *this;
        }

        // postfix
        self_type &operator++(int junk) {
```

```
            self_type cached = *this;
            _rowPtrIter++;
            return cached;
        }

        self_type &operator+=(difference_type offset) {
            _rowPtrIter += offset;
            return *this;
        }

        self_type &operator-=(difference_type offset) {
            _rowPtrIter -= offset;
            return *this;
        }

        // iter - n
        self_type operator-(difference_type offset) {
            self_type cache(*this);
            cache -= offset;
            return cache;
        }

        // iter + n
        self_type operator+(difference_type offset) {
            self_type cache(*this);
            cache += offset;
            return cache;
        }
        // n + iter
        friend self_type operator+(const difference_type &offset,
                                   const self_type &sec) {
            self_type cache(sec);
            cache += offset;
            return cache;
        }

        reference operator[](difference_type offset) {
            return *(*this + offset);
        }

        bool operator<(const self_type &other) {
            return other - (*this) > 0;  //
        }

        bool operator>(const self_type &other) {
            return other < (*this);  //
        }

        [[nodiscard]] self_type &operator*() { return *this; }
        // [[nodiscard]] self_type &operator->() { return this; }

        [[nodiscard]] bool operator==(const self_type &rhs) {
            return _rowPtrIter == rhs._rowPtrIter;
        }
        [[nodiscard]] bool operator!=(const self_type &rhs) {
            return _rowPtrIter != rhs._rowPtrIter;
        }

    private:
        std::vector<size_type>::const_iterator _rowPtrIter;
        std::vector<size_type>::const_iterator _colIndicesIter;
        typename std::vector<REAL>::const_iterator _valIter;
    };

    [[nodiscard]] row_iterator begin() {
        return row_iterator(_rowPtr.begin(), _colIndices.begin(),
                            _data.begin());
    }

    [[nodiscard]] row_iterator end() {
        return row_iterator(_rowPtr.end() - 1, _colIndices.begin(),
                            _data.begin());
    }

    [[nodiscard]] const_row_iterator cbegin() const {
        return const_row_iterator(_rowPtr.cbegin(), _colIndices.cbegin(),
                                  _data.cbegin());
    }

    [[nodiscard]] const_row_iterator cend() const {
        return const_row_iterator(_rowPtr.cend() - 1, _colIndices.cbegin(),
                                  _data.cbegin());
    }

    [[nodiscard]] const_row_iterator begin() const { return this->cbegin(); }
    [[nodiscard]] const_row_iterator end() const { return this->cend(); }
```

```cpp
    void scientific(bool b) const { bScientific = b; }

    size_type iwidth() const { return nIndexWidth; }

    size_type width() const { return nValueWidth; }

    size_type precision() const { return nValuePrecision; }

    void iwidth(size_type i) const { nIndexWidth = i; }

    void width(size_type i) const { nValueWidth = i; }

    void precision(size_type i) const { nValuePrecision = i; }

    column_iterator find(const size_type row_index,
                         const size_type col_index) const {
        checkIfAccessIsInBounds(row_index, col_index);

        using value_pair = typename const_column_index_iterator::value_type;
        auto row = const_row_iterator(_rowPtr.begin() + row_index,
                                      _colIndices.begin(), _data.begin());
        return std::find_if(row.ibegin(), row.iend(),
                            [col_index](value_pair el) {
                                // only care for the index here since the value
                                // is unknown
                                return el.second == col_index;
                            });
    }

    bool exists(const size_type row_index, const size_type col_index) const {
        auto row = const_row_iterator(_rowPtr.begin() + row_index,
                                      _colIndices.begin(), _data.begin());
        return find(row_index, col_index) != row.iend();
    }

    REAL &get(const size_type row_index, const size_type col_index) {
        checkIfAccessIsInBounds(row_index, col_index);
        // look for the entry
        using value_pair = typename const_column_index_iterator::value_type;
        auto row = row_iterator(_rowPtr.begin() + row_index,
                                _colIndices.begin(), _data.begin());
        auto result =
            std::find_if(row.ibegin(), row.iend(), [col_index](value_pair el) {
                // only care for the index here
                // since the value is unknown
                // anyways
                return el.second == col_index;
            });
        // we found something within the right row
        if (result != row.iend()) {
            return result.value();
        }
        throw std::out_of_range(
            "There is no non-zero element for these given indicies!");
    }

    const REAL &operator()(const size_type row_index,
                           const size_type col_index) const {
        checkIfAccessIsInBounds(row_index, col_index);

        using value_pair = typename const_column_index_iterator::value_type;
        auto row = const_row_iterator(_rowPtr.begin() + row_index,
                                      _colIndices.begin(), _data.begin());
        auto result =
            std::find_if(row.ibegin(), row.iend(), [col_index](value_pair el) {
                // only care for the index here since the value is unknown
                return el.second == col_index;
            });
        // we found something within the right row
        if (result != row.iend()) {
            return result.value();
        }
        return _zero;
    }

    [[nodiscard]] bool operator==(const SparseMatrix &other) const {
        return (_data == other._data) and              //
               (_rowPtr == other._rowPtr) and          //
               (_colIndices == other._colIndices) and  //
               (m_cols == other.m_cols) and            //
               (m_rows == other.m_rows);
    }

    [[nodiscard]] bool operator!=(const SparseMatrix &other) const {
        return not (*this == other);
    }

```

```
    // delete all the invalid comparisons
    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 {
        // TODO: remove / find bug here!
        SparseMatrix::builder builder(m_cols, m_rows);
        SparseMatrix::size_type curr_row = 0;
        for (auto &row : (*this)) {
            for (auto it = row.ibegin(); it != row.iend(); it++) {
                builder.addEntry(it.index(), curr_row, it.value());
            }
            curr_row++;
        }

        return builder.build();
    }

    [[nodiscard]] SparseMatrix operator*=(const REAL scalar) {
        // This could also be done out of order
        std::transform(_data.cbegin(), _data.cend(), _data.begin(),
                       [&](REAL value) { return value * scalar; });
    }

    [[nodiscard]] SparseMatrix operator/=(const REAL scalar) {
        // This could also be done out of order
        std::transform(_data.cbegin(), _data.cend(), _data.begin(),
                       [&](REAL value) { return value / scalar; });
    }

    template <class V>
    void mv(Vector<V> &result, const Vector<V> &x) const {
        static_assert(std::is_convertible<V, REAL>::value,
                      "The types in the Matrix vector multiplication cant be "
                      "converted properly!");

        if (result.size() != this->colsize()) {
            HDNUM_ERROR(
                (std::string("The result vector has the wrong dimension! ") +
                 "Vector dimension " + std::to_string(result.size()) +
                 " != " + std::to_string(this->colsize()) + " colsize"));
        }

        if (x.size() != this->colsize()) {
            HDNUM_ERROR(
                (std::string("The input vector has the wrong dimension! ") +
                 "Vector dimension " + std::to_string(x.size()) +
                 " != " + std::to_string(this->colsize()) + " colsize"));
        }

        size_type curr_row = 0;
        for (auto row : (*this)) {
            result[curr_row] = std::accumulate(
                row.ibegin(), row.iend(), V {}, [&](V result, auto el) -> V {
                    return result + (x[el.second] * el.first);
                });
            curr_row++;
        }
    }

    [[nodiscard]] Vector<REAL> operator*(const Vector<REAL> &x) const {
        hdnum::Vector<REAL> result(this->colsize(), 0);
        this->mv(result, x);
        return result;
    }

    template <class V>
    void umv(Vector<V> &result, const Vector<V> &x) const {
        static_assert(std::is_convertible<V, REAL>::value,
                      "The types in the Matrix vector multiplication cant be "
                      "converted properly!");

        if (result.size() != this->colsize()) {
            HDNUM_ERROR(
                (std::string("The result vector has the wrong dimension! ") +
                 "Vector dimension " + std::to_string(result.size()) +
                 " != " + std::to_string(this->colsize()) + " colsize"));
        }

        if (x.size() != this->colsize()) {
            HDNUM_ERROR(
                (std::string("The input vector has the wrong dimension! ") +
                 "Vector dimension " + std::to_string(result.size()) +
                 " != " + std::to_string(this->colsize()) + " colsize"));
        }
```

```
        size_type curr_row {};
        for (auto row : (*this)) {
            result[curr_row] += std::accumulate(
                row.ibegin(), row.iend(), V {}, [&](V result, auto el) -> V {
                    return result + (x[el.second] * el.first);
                });
            curr_row++;
        }
    }

private:
    template <typename norm_type>
    norm_type norm_infty_impl() const {
        norm_type norm {};
        for (auto row : *this) {
            norm_type rowsum =
                std::accumulate(row.begin(), row.end(), norm_type {},
                                [](norm_type res, REAL value) -> norm_type {
                                    return res + std::abs(value);
                                });
            if (norm < rowsum) {
                norm = rowsum;
            }
        }
        return norm;
    }

public:
    auto norm_infty() const {
        if constexpr (is_specialization<REAL, std::complex> {}) {
            return norm_infty_impl<double>();
        } else {
            return norm_infty_impl<REAL>();
        }
    }

    [[nodiscard]] std::string to_string() const noexcept {
        return "values=" + comma_fold(_data) + "\n" +          //
               "colInd=" + comma_fold(_colIndices) + "\n" +   //
               "rowPtr=" + comma_fold(_rowPtr) + "\n";        //
    }

    void print() const noexcept { std::cout « *this; }

    static SparseMatrix identity(const size_type dimN) {
        auto builder = typename SparseMatrix<REAL>::builder(dimN, dimN);
        for (typename SparseMatrix<REAL>::size_type i = 0; i < dimN; ++i) {
            builder.addEntry(i, i, REAL {1});
        }
        return builder.build();
    }

    SparseMatrix<REAL> matchingIdentity() const { return identity(m_cols); }

    class builder {
        size_type m_rows {};  // Number of Matrix rows, 0 by default
        size_type m_cols {};  // Number of Matrix columns, 0 by default
        std::vector<std::map<size_type, REAL» _rows;

    public:
        builder(size_type new_m_rows, size_type new_m_cols)
            : m_rows {new_m_rows}, m_cols {new_m_cols}, _rows {m_rows} {}

        builder(const std::initializer_list<std::initializer_list<REAL» &v)
            : m_rows {v.size()}, m_cols {v.begin()->size()}, _rows(m_rows) {
            size_type i = 0;
            for (auto &row : v) {
                size_type j = 0;
                for (const REAL &element : row) {
                    addEntry(i, j, element);
                    j++;
                }
                i++;
            }
        }

        builder() = default;

        std::pair<typename std::map<size_type, REAL>::iterator, bool> addEntry(
            size_type i, size_type j, REAL value) {
            return _rows.at(i).emplace(j, value);
        }

        std::pair<typename std::map<size_type, REAL>::iterator, bool> addEntry(
            size_type i, size_type j) {
            return addEntry(i, j, REAL {});
```

```
        };

        [[nodiscard]] bool operator==(
            const SparseMatrix::builder &other) const {
            return (m_rows == other.m_rows) and  //
                   (m_cols == other.m_cols) and  //
                   (_rows == other._rows);
        }

        [[nodiscard]] bool operator!=(
            const SparseMatrix::builder &other) const {
            return not (*this == other);
        }

        [[nodiscard]] size_type colsize() const noexcept { return m_cols; }
        [[nodiscard]] size_type rowsize() const noexcept { return m_rows; }

        size_type setNumCols(size_type new_m_cols) noexcept {
            m_cols = new_m_cols;
            return m_cols;
        }
        size_type setNumRows(size_type new_m_rows) {
            m_rows = new_m_rows;
            _rows.resize(m_cols);
            return m_rows;
        }

        void clear() noexcept {
            for (auto &row : _rows) {
                row.clear();
            }
        }

        [[nodiscard]] std::string to_string() const {
            std::string output;
            for (std::size_t i = 0; i < _rows.size(); i++) {
                for (const auto &indexpair : _rows[i]) {
                    output += std::to_string(i) + ", " +
                              std::to_string(indexpair.first) + " => " +
                              std::to_string(indexpair.second) + "\n";
                }
            }
            return output;
        }

        [[nodiscard]] SparseMatrix build() {
            auto result = SparseMatrix<REAL>(m_rows, m_cols);

             for (std::size_t i = 0; i < _rows.size(); i++) {
                 result._rowPtr[i + 1] = result._rowPtr[i];
                 for (const auto &indexpair : _rows[i]) {
                     result._colIndices.push_back(indexpair.first);
                     result._data.push_back(indexpair.second);
                     result._rowPtr[i + 1]++;
                 }
             }
             return result;
         }
    };
};

template <typename REAL>
bool SparseMatrix<REAL>::bScientific = true;
template <typename REAL>
std::size_t SparseMatrix<REAL>::nIndexWidth = 10;
template <typename REAL>
std::size_t SparseMatrix<REAL>::nValueWidth = 10;
template <typename REAL>
std::size_t SparseMatrix<REAL>::nValuePrecision = 3;
template <typename REAL>
const REAL SparseMatrix<REAL>::_zero {};

template <typename REAL>
std::ostream &operator«(std::ostream &s, const SparseMatrix<REAL> &A) {
    using size_type = typename SparseMatrix<REAL>::size_type;

    s « std::endl;
    s « " " « std::setw(A.iwidth()) « " "
      « "  ";
    for (size_type j = 0; j < A.colsize(); ++j) {
        s « std::setw(A.width()) « j « " ";
    }
    s « std::endl;

    for (size_type i = 0; i < A.rowsize(); ++i) {
        s « " " « std::setw(A.iwidth()) « i « "  ";
        for (size_type j = 0; j < A.colsize(); ++j) {
```

```
            if (A.scientific()) {
                s « std::setw(A.width()) « std::scientific « std::showpoint
                  « std::setprecision(A.precision()) « A(i, j) « " ";
            } else {
                s « std::setw(A.width()) « std::fixed « std::showpoint
                  « std::setprecision(A.precision()) « A(i, j) « " ";
            }
        }
        s « std::endl;
    }
    return s;
}

template <typename REAL>
inline void zero(SparseMatrix<REAL> &A) {
    A = SparseMatrix<REAL>(A.rowsize(), A.colsize());
}

template <class REAL>
inline void identity(SparseMatrix<REAL> &A) {
    if (A.rowsize() != A.colsize()) {
        HDNUM_ERROR("Will not overwrite A since Dimensions are not equal!");
    }
    A = SparseMatrix<REAL>::identity(A.colsize());
}

template <typename REAL>
inline void readMatrixFromFile(const std::string &filename,
                               SparseMatrix<REAL> &A) {
    // Format taken from here:
    // https://math.nist.gov/MatrixMarket/formats.html#coord

    using size_type = typename SparseMatrix<REAL>::size_type;
    std::string buffer;
    std::ifstream fin(filename);
    size_type i = 0;
    size_type j = 0;
    size_type non_zeros = 0;

    if (fin.is_open()) {
        // ignore all comments from the file (starting with %)
        while (fin.peek() == '%') fin.ignore(2048, '\n');

        std::getline(fin, buffer);
        std::istringstream first_line(buffer);
        first_line » i » j » non_zeros;

        auto builder = typename SparseMatrix<REAL>::builder(i, j);

        while (std::getline(fin, buffer)) {
            std::istringstream iss(buffer);

            REAL value {};
            iss » i » j » value;
            // i-1, j-1, because matrix market does not use zero based indexing
            builder.addEntry(i - 1, j - 1, value);
        }
        A = builder.build();
        fin.close();
    } else {
        HDNUM_ERROR(("Could not osspen file! \"" + filename + "\""));
    }
}

}  // namespace hdnum

#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.

```
#ifndef DUNE_TIMER_HH
#define DUNE_TIMER_HH

#ifndef TIMER_USE_STD_CLOCK
// headers for getrusage(2)
#include <sys/resource.h>
#endif

#include <ctime>

// headers for stderror(3)
#include <cstring>

// access to errno in C++
#include <cerrno>

#include "exceptions.hh"

namespace hdnum {

  class TimerError : public SystemError {} ;


class Timer
{
public:
        Timer ()
        {
          reset();
        }

        void reset()
        {
#ifdef TIMER_USE_STD_CLOCK
          cstart = std::clock();
#else
          rusage ru;
          if (getrusage(RUSAGE_SELF, &ru))
            HDNUM_THROW(TimerError, strerror(errno));
          cstart = ru.ru_utime;
#endif
        }

        double elapsed () const
        {
#ifdef TIMER_USE_STD_CLOCK
          return (std::clock()-cstart) / static_cast<double>(CLOCKS_PER_SEC);
#else
          rusage ru;
          if (getrusage(RUSAGE_SELF, &ru))
            HDNUM_THROW(TimerError, strerror(errno));
          return 1.0 * (ru.ru_utime.tv_sec - cstart.tv_sec) + (ru.ru_utime.tv_usec - cstart.tv_usec) / (1000.0 * 1000.0);
#endif
        }

private:
#ifdef TIMER_USE_STD_CLOCK
  std::clock_t cstart;
```

```
#else
  struct timeval cstart;
#endif
}; // end class Timer

} // end namespace

#endif
```

5.26 vector.hh

```
// -*- tab-width: 4; indent-tabs-mode: nil; c-basic-offset: 2 -*-
/*
 * File:   vector.hh
 * Author: ngo
 *
 * Created on April 14th, 2011
 */

#ifndef _VECTOR_HH
#define _VECTOR_HH

#include <assert.h>

#include <cmath>
#include <cstdlib>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <sstream>
#include <vector>

#include "exceptions.hh"

namespace hdnum {

  template<typename REAL>
  class Vector : public std::vector<REAL>  // inherit from the STL vector
  {
  public:
    typedef std::size_t size_type;

  private:
    static bool bScientific;
    static std::size_t nIndexWidth;
    static std::size_t nValueWidth;
    static std::size_t nValuePrecision;

  public:

    Vector() : std::vector<REAL>()
    {
    }

    Vector( const size_t size,                 // user must specify the size
            const REAL defaultvalue_ = 0    // if not specified, the value 0 will take effect
            )
      : std::vector<REAL>( size, defaultvalue_ )
    {
    }

    Vector (const std::initializer_list<REAL> &v)
    {
      for (auto elem : v) this->push_back(elem);
    }

    // Methods:

    Vector& operator=( const REAL value )
    {
      const size_t s = this->size();
      Vector & self = *this;
      for(size_t i=0; i<s; ++i)
        self[i] = value;
      return *this;
    }

    Vector sub (size_type i, size_type m)
    {
      Vector v(m);
      Vector &self = *this;
      size_type k=0;
      for (size_type j=i; j<i+m; j++){
```

```
        v[k]=self[j];
        k++;
      }
    return v;
    }



#ifdef DOXYGEN
    Vector& operator=( const Vector& y )
    {
      // It is already implemented in the STL vector class itself!
    }
#endif



    Vector& operator*=( const REAL value )
    {
      Vector &self = *this;
      for (size_t i = 0; i < this->size(); ++i)
        self[i] *= value;
      return *this;
    }


    Vector& operator/=( const REAL value )
    {
      Vector &self = *this;
      for (size_t i = 0; i < this->size(); ++i)
        self[i] /= value;
      return *this;
    }


    Vector& operator+=( const Vector & y )
    {
      assert( this->size() == y.size());
      Vector &self = *this;
      for (size_t i = 0; i < this->size(); ++i)
        self[i] += y[i];
      return *this;
    }


    Vector& operator-=( const Vector & y )
    {
      assert( this->size() == y.size());
      Vector &self = *this;
      for (size_t i = 0; i < this->size(); ++i)
        self[i] -= y[i];
      return *this;
    }


    Vector & update(const REAL alpha, const Vector & y)
    {
      assert( this->size() == y.size());
      Vector &self = *this;
      for (size_t i = 0; i < this->size(); ++i)
        self[i] += alpha * y[i];
      return *this;
    }


    REAL operator*(Vector & x) const
    {
      assert( x.size() == this->size() );   // checks if the dimensions of the two vectors are equal
      REAL sum( 0 );
      const Vector & self = *this;
      for( size_t i = 0; i < this->size(); ++i )
        sum += self[i] * x[i];
      return sum;
    }




    Vector operator+(Vector & x) const
    {
      assert( x.size() == this->size() );   // checks if the dimensions of the two vectors are equal
      Vector sum( *this );
      sum += x;
      return sum;
    }


```

```
  Vector operator-(Vector & x) const
  {
    assert( x.size() == this->size() );   // checks if the dimensions of the two vectors are equal
    Vector sum( *this );
    sum -= x;
    return sum;
  }



  REAL two_norm_2() const
  {
    REAL sum( 0 );
    const Vector & self = *this;
    for (size_t i = 0; i < (size_t) this->size(); ++i)
      sum += self[i] * self[i];
    return sum;
  }

  REAL two_norm() const
  {
    return sqrt(two_norm_2());
  }

  bool scientific() const
  {
    return bScientific;
  }

  void scientific(bool b) const
  {
    bScientific=b;
  }

  std::size_t iwidth () const
  {
    return nIndexWidth;
  }

  std::size_t width () const
  {
    return nValueWidth;
  }

  std::size_t precision () const
  {
    return nValuePrecision;
  }

  void iwidth (std::size_t i) const
  {
    nIndexWidth=i;
  }

  void width (std::size_t i) const
  {
    nValueWidth=i;
  }

  void precision (std::size_t i) const
  {
    nValuePrecision=i;
  }

};



template<typename REAL>
bool Vector<REAL>::bScientific = true;

template<typename REAL>
std::size_t Vector<REAL>::nIndexWidth = 2;

template<typename REAL>
std::size_t Vector<REAL>::nValueWidth = 15;

template<typename REAL>
std::size_t Vector<REAL>::nValuePrecision = 7;


template <typename REAL>
inline std::ostream & operator «(std::ostream & os, const Vector<REAL> & x)
{
  os « std::endl;

```

```
  for (size_t r = 0; r < x.size(); ++r)
    {
      if( x.scientific() )
        {
          os « "["
             « std::setw(x.iwidth())
             « r
             « "]"
             « std::scientific
             « std::showpoint
             « std::setw( x.width() )
             « std::setprecision( x.precision() )
            « x[r]
             « std::endl;
        }
      else
        {
          os « "["
             « std::setw(x.iwidth())
             « r
             « "]"
             « std::fixed
             « std::showpoint
             « std::setw( x.width() )
             « std::setprecision( x.precision() )
             « x[r]
             « std::endl;
        }
    }
  return os;
}



template<typename REAL>
inline void gnuplot(
                    const std::string& fname,
                    const Vector<REAL> x
                    )
{
  std::fstream f(fname.c_str(),std::ios::out);
  for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
    {
      if( x.scientific() )
        {
          f « std::setw(x.width())
            « i
            « std::scientific
            « std::showpoint
            « std::setw( x.width() )
            « std::setprecision( x.precision() )
            « x[i]
            « std::endl;
        }
      else
        {
          f « std::setw(x.width())
            « i
            « std::fixed
            « std::showpoint
            « std::setw( x.width() )
            « std::setprecision( x.precision() )
            « x[i]
            « std::endl;
        }
    }
  f.close();
}

template<typename REAL>
inline void gnuplot(
                    const std::string& fname,
                    const std::vector<std::string>& t,
                    const Vector<REAL> x
                    )
{
  std::fstream f(fname.c_str(),std::ios::out);
  for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
    {
      if( x.scientific() )
        {
          f « t[i] « " "
            « std::scientific
            « std::showpoint
            « std::setw( x.width() )
            « std::setprecision( x.precision() )
            « x[i]
```

```
            « std::endl;
        }
      else
        {
          f « t[i] « " "
            « std::fixed
            « std::showpoint
            « std::setw( x.width() )
            « std::setprecision( x.precision() )
            « x[i]
            « std::endl;
        }
    }
  f.close();
}

template<typename REAL>
inline void gnuplot(
                    const std::string& fname,
                    const Vector<REAL> x,
                    const Vector<REAL> y
                    )
{
  std::fstream f(fname.c_str(),std::ios::out);
  for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
    {
      if( x.scientific() )
        {
          f « std::setw(x.width())
            « i
            « std::scientific
            « std::showpoint
            « std::setw( x.width() )
            « std::setprecision( x.precision() )
            « x[i]
            « " "
            « std::setw( x.width() )
            « std::setprecision( x.precision() )
            « y[i]
            « std::endl;
        }
      else
        {
          f « std::setw(x.width())
            « i
            « std::fixed
            « std::showpoint
            « std::setw( x.width() )
            « std::setprecision( x.precision() )
            « x[i]
            « " "
            « std::setw( x.width() )
            « std::setprecision( x.precision() )
            « y[i]
            « std::endl;
        }
    }

  f.close();
}



template<typename REAL>
inline void readVectorFromFile (const std::string& filename, Vector<REAL> &vector)
{
  std::string buffer;
  std::ifstream fin( filename.c_str() );
  if( fin.is_open() ){
    while( fin ){
      std::string sub;
      fin » sub;
      //std::cout « " sub = " « sub.c_str() « ": ";
      if( sub.length()>0 ){
        REAL a = atof(sub.c_str());
        //std::cout « std::fixed « std::setw(10) « std::setprecision(5) « a;
        vector.push_back(a);
      }
    }
    fin.close();
  }
  else{
    HDNUM_ERROR("Could not open file!");
  }
}


```

```
  template<class REAL>
  inline void zero (Vector<REAL>& x)
  {
    for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
      x[i] = REAL(0);
  }

  template<class REAL>
  inline REAL norm (Vector<REAL> x)
  {
    REAL sum(0.0);
    for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
      sum += x[i]*x[i];
    return sqrt(sum);
  }

  template<class REAL>
  inline void fill (Vector<REAL>& x, const REAL t)
  {
    for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
      x[i] = t;
  }

  template<class REAL>
  inline void fill (Vector<REAL>& x, const REAL& t, const REAL& dt)
  {
    REAL myt(t);
    for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
      {
        x[i] = myt;
        myt += dt;
      }
  }


  template<class REAL>
  inline void unitvector (Vector<REAL> & x, std::size_t j)
  {
    for (typename Vector<REAL>::size_type i=0; i<x.size(); i++)
      if (i==j)
        x[i] = REAL(1);
      else
        x[i] = REAL(0);
  }


} // end of namespace hdnum

#endif  /* _VECTOR_HH */
```

Index