

# 1 Preface

## 1.1 Brief history

CVM C++ class library<sup>1</sup> encapsulates concepts of vector and different kinds of matrices including square, band, symmetric and hermitian ones, in Euclidean space of real and complex numbers. The library is distributed under the [Boost Software License, Version 1.0](#). First version of the library was released in 1992. Originally it allowed to simplify a code dealing with matrices and vectors in the following way:

```
rvector a(10), b(20);
rmatrix A(20,10);
.....
b = A * a;
cout << "Norm of b equals " << b.norm() << endl;
```

By overloading of arithmetic operators and parentheses relative readability of source code was achieved.

In 1995<sup>2</sup> Russian office of Intel Corporation began to distribute a CD with last software products from Intel. Among others there was a freely (at that time) distributed library called “Intel BLAS Library”. More than 20 years [BLAS library](#) (Basic Linear Algebra Subprograms) is known to experts in numerical algorithms programming in FORTRAN language. This library features common vector-matrix operations for data types REAL and DOUBLE PRECISION. Both implementations of the BLAS are utilized since version 2.0 of CVM library. For example, operator of addition of two vectors utilizes subroutine DAXPY. Later Intel changed the name “BLAS” to “Intel Math Kernel Library”. Since version 3.0 [LAPACK](#) subroutines were added to the Intel MKL library. Version 4.0 featured complex numbers. And since version 5.0 band, symmetric and hermitian matrices are implemented as well. Version 6.0 has many additional algorithms implemented, performance enhancements and bug fixes. Version 6.1 brought transposed versions of linear solvers as well some vector–matrix operators.

## 1.2 Libraries Supported

The [last version](#) of the Intel MKL library at the moment of writing of this document for Win32 and Linux is 11.0 Update 1. Version 8.0 of the CVM library was built and tested with that MKL library implementation.

Besides that [AMD ACML library](#) is supported as well. Current CVM Library was built and tested with AMD ACML 4.4.0 for 32 bit platforms and ACML 5.3.0 for 64 bit.

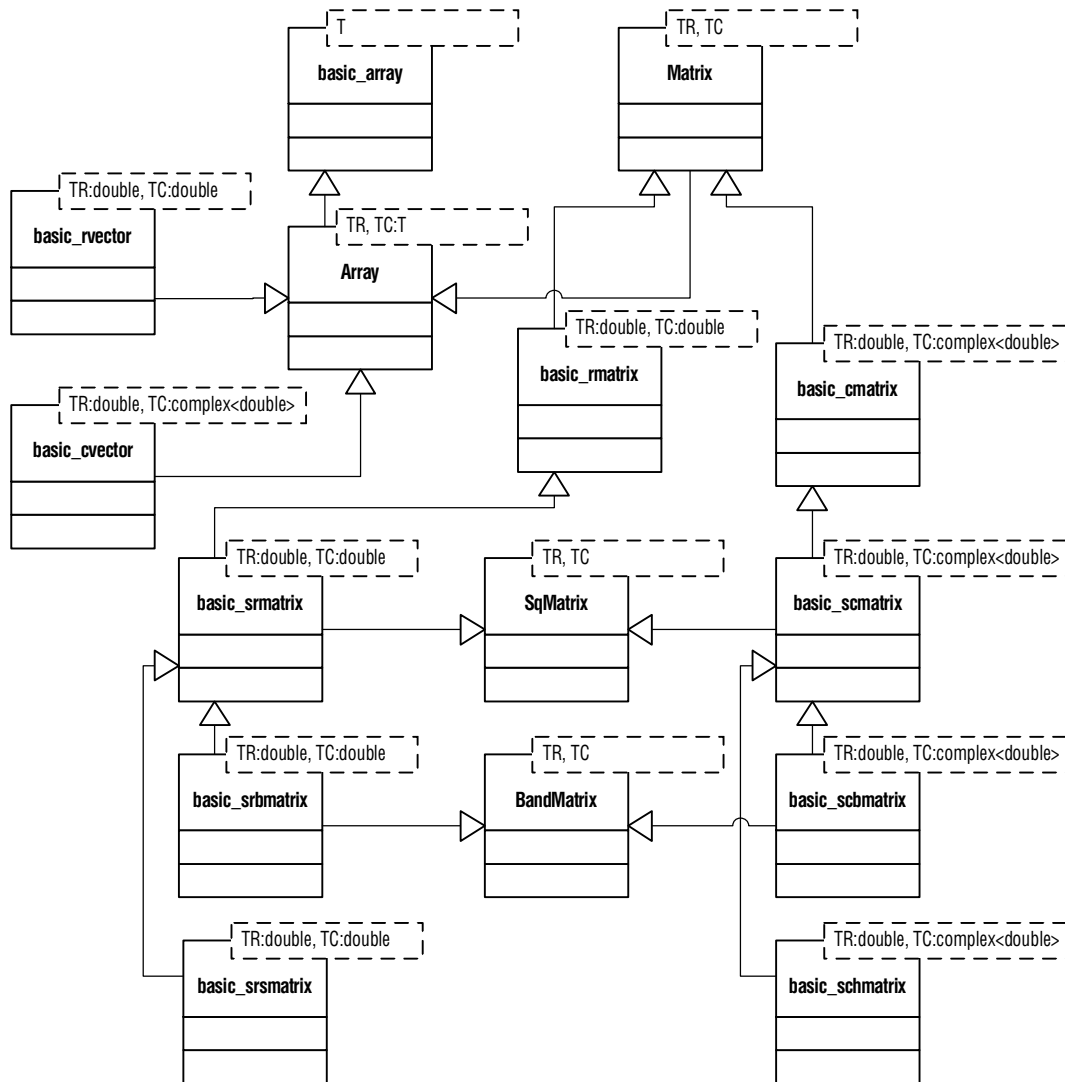
Finally, [native BLAS and LAPACK](#) libraries are supported and starting from CVM 5.7 they are part of binary packages as well.

<sup>1</sup> This document describes version 8.0 and it does not cover function classes yet. Use online documentation for them. Copyright © Sergei Nikolaev, 1992–2012, <http://cvmlib.com>

<sup>2</sup> I’m not sure actually, may be that was 1994

## 1.3 Object Model

Object model of the CVM library is shown on the following picture



Base class's names are beginning with capital letters. They implement common interfaces and are *not* designed to be instantiated. This is the list of end-user classes:

- `basic_array` is an abstract array of elements of any type. It's used mostly as an array of integers like `basic_array<int>`, but you can use it in your projects within any other types as well.
- `basic_rvector` is a class encapsulating vector of real numbers.
- `basic_cvector` is a class encapsulating vector of complex numbers.
- `basic_rmatrix` is a class encapsulating matrix of real numbers.

- `basic_cmatrix` is a class encapsulating matrix of complex numbers.
- `basic_srmatrix` is a class encapsulating square matrix of real numbers.
- `basic_scmatrix` is a class encapsulating square matrix of complex numbers.
- `basic_srbmatrix` is a class encapsulating square band matrix of real numbers. Packed storage is used.
- `basic_scbmatrix` is a class encapsulating square band matrix of complex numbers. Packed storage is used.
- `basic_srsmatrix` is a class encapsulating symmetric matrix of real numbers.
- `basic_schmatrix` is a class encapsulating hermitian matrix of complex numbers.

You don't need to use those class names directly unless you want to typedef your own ones. Otherwise you should use the following pre-defined classes:

```
typedef basic_array    <int>          iarray;
typedef basic_rvector  <treal>        rvector;
typedef basic_rmatrix  <treal>        rmatrix;
typedef basic_srmatrix <treal>        srmatrix;
typedef basic_cvector  <treal, tcomplex> cvector;
typedef basic_cmatrix  <treal, tcomplex> cmatrix;
typedef basic_scmatrix <treal, tcomplex> scmatrix;
typedef basic_srbmatrix<treal>        srbmatrix;
typedef basic_scbmatrix<treal, tcomplex> scbmatrix;
typedef basic_srsmatrix<treal>        srsmatrix;
typedef basic_schmatrix<treal, tcomplex> schmatrix;
```

The rest of this manual describes them in details.

## 1.4 Installation

### 1.4.1 Directory Structure

The CVM library distribution has the following directory structure.

- `.*2010.sln` MS Visual Studio 2010 solution files.
- `.*2012.sln` MS Visual Studio 2012 solution files.
- `ftn` contains FORTRAN and project files for Intel Fortran compiler and GNU `gfortran`. This source code is the part of CVM library, it contains some numerical algorithms implementation.
- `lib` is the place for 32-bit libraries.
- `lib64` is the place for 64-bit libraries.
- `test` contains regression test code and projects.
- `src` contains source code of the library along with `cvm.h` header file.

### 1.4.2 Usage Notes

These are definitions and data types used in the library.

<code>CVM_ZERO_BASED</code>	define this macro to build the library with <a href="#">0-based indexing</a>
<code>CVM_ACML</code>	define this macro to link against AMD ACML and/or ACML MP library
<code>CVM_FLOAT</code>	define this macro in order to build a float version
<code>CVM_NO_NAMESPACE</code>	define this macro if you don't want to use namespace
<code>tint</code>	is typedef'ed as <code>long long int</code> (or as <code>__int64</code> for MSVC) if ILP64 is defined and as <code>int</code> otherwise (by default)
<code>treal</code>	is typedef'ed as <code>float</code> if <code>CVM_FLOAT</code> is defined and as <code>double</code> otherwise (by default)
<code>tcomplex</code>	is typedef'ed as <code>std::complex&lt;treal&gt;</code>

In order to use the library just include its header file:

```
#include <cvm.h>
```

You should also link your project with one of `cvm*.lib` for Microsoft's C++ compilers and `cvm*.so` for GNU C++ compilers (debug versions are `*_debug.lib` and `*_debug.so` respectively).

### 1.4.3 Installation – Win32

If you don't want to rebuild the library just download an appropriate version of `cvm*.dll` and `cvm*.lib` files from binaries section. If you want to rebuild the whole library you'll need Intel Fortran 11.1 and Intel C++ 11.1 compilers (or higher) along with MS Visual Studio 2010 or 2012. You'll also need the Intel MKL 10.2 (or higher) library. You will also need STL library coming with MS VC++ or the [STLport](#) library. Open `.\cvmlib*.sln` solution and choose the library version you want to build.

### 1.4.4 Installation – Unix and MinGW

Use the Makefile provided in the root directory:

```
make [release|debug] [IFORT=1] [ICC=1] [MKL=1] [MKL_PATH=/opt/...]
    [ACML=1] [ACML_MP=1] [ACML_PATH=/opt/...] [EM64T=1] [ILP64=1]
    [CVM_FLOAT=1] [ICCT=1] [STATIC_ONLY=1] [IFORT_PATH=/opt/...]
    [MAC=1]
```

Here

- `release|debug` is a target (by default it builds both)
- `IFORT=1` instructs to use `ifort` compiler (by default it's `gfortran`)
- `ICC=1` instructs to use `icpc` (by default it's `g++`)
- `MKL=1` instructs to use Intel MKL library (by default it uses native BLAS and LAPACK libraries)
- `MKL_PATH=path` specifies the directory where the MKL is installed to. By default it's equal to `/opt/intel/composerxe/mkl/lib`.
- `ACML=1` instructs to use AMD ACML library (overrides `MKL=1` and `ACML_MP=1`).

- `ACML_MP=1` instructs to use AMD ACML MP library (overrides `MKL=1`).
- `ACML_PATH=path` specifies the directory where the ACML is installed to. (overrides `MKL_PATH=path`).
- `EM64T=1` instructs to build EM64T version of the library.
- `ILP64=1` instructs to build INT64-based version of the library (EM64T must be turned on).
- `CVM_FLOAT=1` instructs to build float version of the library.
- `ICCT=1` instructs to use Intel C++ compiler for building the regression test utility (by default it's g++)
- `STATIC_ONLY=1` instructs to build static libraries only. Both `.so` and `.a` will be built otherwise.
- `IFORT_PATH=path` specifies the directory where the Intel Fortran is installed to. It's required only when you build ACML version within Intel Fortran and Intel C++ compilers.
- `MAC=1` instructs to build Mac OS X version of the library.

On Unix platforms Intel MKL, AMD ACML and native BLAS/LAPACK libraries as well as both Intel's and GNU compilers are supported. On MinGW platform the native libraries and compilers are only supported.

## 1.5 Storage

The way of storage of matrices units is the same as the FORTRAN's one. Units are stored by columns, see the following example:

```
cvm::rmatrix m(2,2);
m(1,1) = 1.;      // first row, first column
m(1,2) = 2.;      // first row, second column
m(2,1) = 3.;      // second row, first column
m(2,2) = 4.;      // second row, second column
double* p = m;
cout << p[0] << " " << p[1] << " " << p[2] << " " << p[3] << endl;
```

Output will be the following:

```
1 3 2 4
```

Since version 5.0 band matrices are introduced. Band storage can be described as follows (cited from MKL manual): *an  $m$  by  $n$  band matrix with  $k_l$  sub-diagonals and  $k_u$  super-diagonals is stored compactly in a two-dimensional array with  $k_l + k_u + 1$  rows and  $n$  columns. Columns of the matrix are stored in the corresponding columns of the array, and diagonals of the matrix are stored in rows of the array.* This way of storage can be illustrated as follows (referenced

elements are shown as “\*”, not referenced as “—”, zeros are not stored):

$$\begin{array}{l}
 m = n = 3, k_l = 0, k_u = 0 : \begin{bmatrix} * & 0 & 0 \\ 0 & * & 0 \\ 0 & 0 & * \end{bmatrix} \\
 m = n = 4, k_l = 1, k_u = 0 : \begin{bmatrix} * & 0 & 0 & 0 \\ * & * & 0 & 0 \\ 0 & * & * & 0 \\ 0 & 0 & * & * \\ & & & - \end{bmatrix} \\
 m = n = 6, k_l = 1, k_u = 2 : \begin{bmatrix} - & & & & & \\ - & - & & & & \\ * & * & * & 0 & 0 & 0 \\ * & * & * & * & 0 & 0 \\ 0 & * & * & * & * & 0 \\ 0 & 0 & * & * & * & * \\ 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & * & * \\ & & & & & - \end{bmatrix}
 \end{array}$$

CVM library implements square band matrices only, therefore  $m = n$  is satisfied for them.

## 1.6 Indexing

By default index numbering in CVM library corresponds to the FORTRAN's one: index of the first unit is equal to 1:

```

cvm::vector v(2);
cvm::matrix m(2,3);
v[1] = 1.3;      // first vector unit
v(2) = 2.1;     // second vector unit
m(1) = v;       // first column of matrix
m(1,2) = 3.7;   // element located on the first row
                // and the second column of matrix

```

Starting from version 5.7 0-based indexing is supported as well. In order to use it you'd need to re-build the library with CVM\_ZERO\_BASED macro defined. To distinguish this fact, *ℓ-based* notation will be used below in this document, which means 1-based by default and 0-based if the macro is used.

## 1.7 Polymorphism

The majority of CVM Class Library member functions are not declared as virtual, but it doesn't mean that the classes are not polymorphic. Those member functions just wrap virtual ones. For example, the following code

```
void print_solution (const srmatrix& a, const rvector& b)
{
    std::cout << a.solve(b);
}
...
rvector b(3);
srmatrix m(3);
srsrmatrix ms(3);
...
print_solution(m, b);
print_solution(ms, b);
```

will use symmetric solver for symmetric matrix ms.

## 1.8 Multi-threading

The library fully supports multi-threading environments. Its binary files are linked with multi-threaded versions of appropriate run-time libraries.

## 1.9 Regression test utility

The library is shipped with regression utility testing almost all its functions and operators. It's strongly recommended to build it upon installation and verify (see test directory for workspace and make files). It has the following syntax:

```
regtest_* [-t<Number of threads to run>] [-r<Number of executions>]
```

Example:

```
D:\cvmlib\lib>regtest_cvm_ia32.exe -t2 -r2
TESTS STARTED
TESTS STARTED
ALL TESTS SUCEEDED
ALL TESTS SUCEEDED
TESTS STARTED
TESTS STARTED
ALL TESTS SUCEEDED
ALL TESTS SUCEEDED
TOTAL TIME 1.83e+000 sec.
```

## 2 CVM Class Library Reference

### 2.1 basic\_array

This class contains array-specific member functions inherited in other classes. It can be utilized as a standalone class too. It also provides STL-compatible functions and type definitions, so itself and derived classes can be used in the same way as `std::vector<T>`. Since version 5.0 the `iarray` class is defined as

```
typedef basic_array<int> iarray;

template <typename T>
class basic_array {
public:
    int basic_array ();
    explicit basic_array (int nSize, bool bZeroMemory = true);
    basic_array (T* p, int nSize);
    basic_array (const T* p, int nSize);
    basic_array (const T* first, const T* last);
    basic_array (const basic_array& a);
    int size () const;
    T* get ();
    const T* get () const;
    operator T* ();
    operator const T* () const;
    T& operator () (int i) throw (cvmexception);
    T operator () (int i) const throw (cvmexception);
    T& operator [] (size_type i) throw (cvmexception);
    T operator [] (size_type i) const throw (cvmexception);
    T& operator [] (int i) throw (cvmexception);
    T operator [] (int i) const throw (cvmexception);
    basic_array& operator = (const basic_array& a) throw (cvmexception);
    basic_array& assign (const T* p);
    basic_array& set (T x);
    basic_array& resize (int nNewSize) throw (cvmexception);
    // STL-specific type definitions
    typedef T value_type;
    typedef value_type* pointer;
    typedef const value_type* const_pointer;
    typedef value_type* iterator;
    typedef const value_type* const_iterator;
    typedef value_type& reference;
    typedef const value_type& const_reference;
    typedef size_t size_type;
```



```
typedef ptrdiff_t difference_type;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
typedef std::reverse_iterator<iterator> reverse_iterator;
// STL-specific functions
iterator begin ();
const_iterator begin () const;
iterator end ();
const_iterator end () const;
reverse_iterator rbegin ();
const_reverse_iterator rbegin () const;
reverse_iterator rend ();
const_reverse_iterator rend () const;
size_type max_size () const;
size_type capacity () const;
bool empty () const;
reference front ();
const_reference front ();
reference back ();
const_reference back () const;
void assign (size_type n, const T& val) throw (cvmexception);
void assign (const_iterator first,
             const_iterator last) throw (cvmexception);
void resize (size_type nNewSize) throw (cvmexception);
void clear ();
void swap (basic_array& v);
reference at (size_type n) throw (cvmexception);
const_reference at (size_type n) const throw (cvmexception);
void push_back (const T& x) throw (cvmexception);
void pop_back () throw (cvmexception);
iterator insert (iterator position, const T& x) throw (cvmexception);
iterator erase (iterator position) throw (cvmexception);

template <typename T>
friend std::istream& operator >> <> (const std::istream& is,
                                     basic_array<T>& aIn);

template <typename T>
friend std::ostream& operator << <> (std::ostream& os,
                                     const basic_array<T>& aOut);
};
```

### 2.1.1 `basic_array()`

Default constructor

```
basic_array<T>::basic_array();
```

creates empty `basic_array` object. See also [basic\\_array](#). Example:

```
using namespace cvm;
```

```
iarray a;  
std::cout << a.size() << std::endl;  
a.resize(10);  
std::cout << a.size() << std::endl;
```

prints

```
0  
10
```

### 2.1.2 `basic_array(int, bool)`

Constructor

```
explicit basic_array<T>::basic_array(int nSize, bool bZeroMemory = true);
```

creates `basic_array` object of size equal to `nSize`. Allocated memory is initialized with zero values by default (you can pass `false` in second argument in order to avoid this initialization). Constructor throws `cvmexception` in case of non-positive size passed or memory allocation failure. See also `basic_array`. Example:

```
using namespace cvm;
```

```
iarray a(5);  
std::cout << a.size() << " " << a[1] << std::endl;
```

prints

```
5 0
```

### 2.1.3 basic\_array(T\*, int)

Constructor

```
basic_array<T>::basic_array (T* p, int nSize);
```

creates basic\_array object of size equal to nSize and shares memory (shallow copy) with array pointed to by p parameter. Constructor throws **cvmexception** in case of non-positive size passed. See also **basic\_array**. Example:

```
using namespace cvm;
```

```
int a[] = {1, 2, 3, 4};  
iarray v (a, 3);  
std::cout << v;  
a[0] = 77;  
std::cout << v;
```

prints

```
1 2 3  
77 2 3
```

#### 2.1.4 `basic_array(const T*, int)`

Constructor

```
basic_array<T>::basic_array (const T* p, int nSize);
```

creates `basic_array` object of size equal to `nSize` and copies `nSize` elements of an array `p` to it. Constructor throws `cvmexception` in case of non-positive size passed or memory allocation failure. See also `basic_array`. Example:

```
using namespace cvm;
```

```
const int a[] = {1, 2, 3, 4};  
iarray v (a, 3);  
std::cout << v;
```

prints

```
1 2 3
```

### 2.1.5 `basic_array(const T*, const T*)`

Constructor

```
basic_array<T>::basic_array (const T* first, const T* last);
```

creates `basic_array` object of size equal to `last-first` and copies all elements in the range of `[first,last)` to it. Constructor throws `cvmexception` in case of wrong range passed or memory allocation failure. See also `basic_array`. Example:

```
using namespace cvm;
```

```
const int a[] = {1, 2, 3, 4};  
const iarray v (a+1, a+3);  
std::cout << v << std::endl;
```

prints

```
2 3
```

### 2.1.6 `basic_array(const basic_array&)`

Copy constructor

```
basic_array<T>::basic_array (const basic_array& a);
```

creates `basic_array` object of size equal to size of vector `a` and sets every element of created array to value of appropriate element of `a`. See also [basic\\_array](#). Example:

```
using namespace cvm;
```

```
iarray a(5);  
a.set(3);  
iarray b(a);  
std::cout << b;
```

prints

```
3 3 3 3 3
```

### 2.1.7 `size()`

Function

```
int basic_array<T>::size () const;
```

returns number of elements of calling array. This function is *inherited* in all classes of the library: `rvector`, `cvector`, `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srbmatrix`, `scbmatrix`, `srsmatrix` and `schmatrix`. See also `basic_array`. Example:

```
using namespace cvm;
```

```
rvector v(3);  
cmatrix m(10,20);  
cout << v.size() << " " << m.size() << endl;
```

prints

```
3 200
```



**2.1.8 `get()`, operator `T*()`**

Functions and operators

```
T* basic_array<T>::get ();
const T* basic_array<T>::get () const;
basic_array<T>::operator T* ();
basic_array<T>::operator const T* () const;
```

return pointer to beginning (first element) of calling array. These functions and operators are *inherited* in all classes of the library: `rvector`, `cvector`, `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srbmatrix`, `scbmatrix`, `srsmatrix` and `schmatrix`. See also `basic_array`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

void cprint (const std::complex<double>* p, int size)
{
    for (int i = 0; i < size; ++i)
        std::cout << p[i] << " ";
    std::cout << std::endl;
}
...
iarray a(10);
scmatrix m(3);
a[2] = 1;
m(3,1) = std::complex<double>(1., 2.);
std::cout << a.get()[1] << std::endl;
cprint(m, 3);

prints
1
(0.00e+00,0.00e+00) (0.00e+00,0.00e+00) (1.00e+00,2.00e+00)
```

### 2.1.9 Indexing operators

Indexing operators

```
T& basic_array<T>::operator () (int i) throw (cvmexception);
T basic_array<T>::operator () (int i) const throw (cvmexception);
T& basic_array<T>::operator [] (size_type i) throw (cvmexception);
T basic_array<T>::operator [] (size_type i) const throw (cvmexception);
T& basic_array<T>::operator [] (int i) throw (cvmexception);
T basic_array<T>::operator [] (int i) const throw (cvmexception);
```

return reference (or value for constant versions) to *i*-th element of calling array. Please note that *all indexing operators of the library are ℓ-based*. These operators are *inherited* in `rvector` and `cvector` classes of the library but *reimplemented* in other ones: `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srbmatrix`, `scbmatrix`, `srsmatrix` and `schmatrix`. See also `basic_array`. Example:

```
using namespace cvm;
```

```
try {
    rvector v (10);
    v[1] = 1.;
    v(2) = 2.;
    std::cout << v;

    double a[] = {1., 2., 3., 4.};
    const rvector vc (a, 4);
    std::cout << vc(1) << " " << vc[2] << std::endl;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
1 2 0 0 0 0 0 0 0 0
1 2
```

**2.1.10 operator = (const basic\_array&)**

Assignment operator

```
basic_array<T>&  
basic_array<T>::operator = (const basic_array& a) throw (cvmexception);
```

sets every element of calling array to value of appropriate element of array `a` and returns reference to the object changed. This operator is *reimplemented* in all classes of the library: `rvector`, `cvector`, `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srbmatrix`, `scbmatrix`, `srsmatrix` and `schmatrix`. See also `basic_array`. Operator throws `cvmexception` in case of different sizes of the arrays. Example:

```
using namespace cvm;
```

```
iarray a(5), b(5);  
a.set(3);  
b = a;  
std::cout << b;
```

prints

```
3 3 3 3 3
```

### 2.1.11 `assign(const T*)`

Function

```
basic_array<T>& basic_array<T>::assign (const T* p);
```

sets every element of calling array to value of appropriate element of array pointed to by parameter `p` and returns reference to the object changed. This function is *reimplemented* in all classes of the library: `rvector`, `cvector`, `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srbmatrix`, `schmatrix`, `srsmatrix` and `schmatrix`. See also `basic_array`. Example:

```
using namespace cvm;
```

```
const int a[] = {1, 2, 3, 4, 5, 6, 7};  
iarray v (5);
```

```
v.assign(a);  
std::cout << v;
```

prints

```
1 2 3 4 5
```

### 2.1.12 `set(T)`

Function

```
basic_array<T>& basic_array<T>::set (T x);
```

sets every element of calling array to value of parameter `x` and returns reference to the object changed. This function is *reimplemented* in all classes of the library: `rvector`, `cvector`, `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srbmatrix`, `scbmatrix`, `srsmatrix` and `schmatrix`. See also `basic_array`. Example:

```
using namespace cvm;
```

```
iarray a(5);  
a.set(3);  
std::cout << a;
```

prints

```
3 3 3 3 3
```

### 2.1.13 **resize**

Function

```
basic_array<T>&  
basic_array<T>::resize (int nNewSize) throw (cvmexception);
```

changes size of calling array to be equal to `nNewSize` and returns reference to the array changed. Calling array will be filled with zeroes in case of increasing of its size. This function is *reimplemented* in all classes of the library: `rvector`, `cvector`, `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srbmatrix`, `scbmatrix`, `srsmatrix` and `schmatrix`. Function throws `cvmexception` in case of non-positive size passed or memory allocation failure. See also `basic_array`. Example:

```
using namespace cvm;  
  
try {  
    const int a[] = {1, 2, 3, 4};  
    iarray v (a, 3);  
    std::cout << v;  
    v.resize(2);  
    std::cout << v;  
    v.resize(4);  
    std::cout << v;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}  
  
prints  
  
1 2 3  
1 2  
1 2 0 0
```

### 2.1.14 STL-specific type definitions

Type definitions

```
typedef T value_type;
typedef value_type* pointer;
typedef const value_type* const_pointer;
typedef value_type* iterator;
typedef const value_type* const_iterator;
typedef value_type& reference;
typedef const value_type& const_reference;
typedef size_t size_type;
typedef ptrdiff_t difference_type;
typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
typedef std::reverse_iterator<iterator> reverse_iterator;
```

are provided for every class of the library to be compatible with STL algorithms and methods. See also [basic\\_array](#). Example:

```
using namespace cvm;

rvector vs1(5);
vs1[1] = 1.; vs1[2] = 2.; vs1[3] = 3.; vs1[4] = 4.; vs1[5] = 5.;
std::cout << vs1;

rvector::iterator it = vs1.begin() + 1;
rvector::iterator ite = vs1.erase(it);
std::cout << vs1;
std::cout << *ite << std::endl;

ite = vs1.insert(ite, 10.);
std::cout << vs1;
std::cout << *ite << std::endl;

vs1.push_back(11.);
std::cout << vs1;

vs1.randomize(0., 2.);
std::cout << vs1;
std::sort(vs1.begin(), vs1.end());
std::cout << vs1;
std::cout << *std::max_element(vs1.begin(), vs1.end()) << std::endl;

std::reverse(vs1.begin(), vs1.end());
std::cout << vs1;
```

prints

```
1.00e+000 2.00e+000 3.00e+000 4.00e+000 5.00e+000
1.00e+000 3.00e+000 4.00e+000 5.00e+000
3.00e+000
1.00e+000 1.00e+001 3.00e+000 4.00e+000 5.00e+000
1.00e+001
1.00e+000 1.00e+001 3.00e+000 4.00e+000 5.00e+000 1.10e+001
1.11e-001 4.96e-001 1.70e+000 1.91e+000 1.19e-001 1.11e+000
1.11e-001 1.19e-001 4.96e-001 1.11e+000 1.70e+000 1.91e+000
1.91e+000
1.91e+000 1.70e+000 1.11e+000 4.96e-001 1.19e-001 1.11e-001
```



**2.1.15 STL-specific functions:**

**begin(), end(), rbegin(), rend(),  
max\_size(), capacity(), empty(), front(), back(),  
assign(), resize(), clear(), swap()**

## Functions

```
basic_array<T>::iterator basic_array<T>::begin();
basic_array<T>::const_iterator basic_array<T>::begin() const;
basic_array<T>::iterator basic_array<T>::end();
basic_array<T>::const_iterator basic_array<T>::end() const;
basic_array<T>::reverse_iterator basic_array<T>::rbegin();
basic_array<T>::const_reverse_iterator basic_array<T>::rbegin() const;
basic_array<T>::reverse_iterator basic_array<T>::rend();
basic_array<T>::const_reverse_iterator basic_array<T>::rend();
basic_array<T>::size_type basic_array<T>::max_size() const;
basic_array<T>::size_type basic_array<T>::capacity() const;
bool basic_array<T>::empty() const;
basic_array<T>::reference basic_array<T>::front();
basic_array<T>::const_reference basic_array<T>::front();
basic_array<T>::reference basic_array<T>::back();
basic_array<T>::const_reference basic_array<T>::back();
void basic_array<T>::assign (size_type n,
                           const T& val) throw (cvmexception);
void basic_array<T>::assign (const_iterator first,
                           const_iterator last) throw (cvmexception);
void basic_array<T>::resize (size_type nNewSize) throw (cvmexception);
void basic_array<T>::clear();
void basic_array<T>::swap (basic_array& v) throw (cvmexception);
```

are provided for every class of the library to be compatible with STL algorithms and methods. See also [basic\\_array](#) and [STL documentation](#) for further details. Example:

```
using namespace cvm;

iarray a(5);
a[1] = 1; a[2] = 2; a[3] = 3; a[4] = 4; a[5] = 5;

for (iarray::reverse_iterator it = a.rbegin(); it != a.rend(); ++it)
{
    std::cout << *it << " ";
}
std::cout << std::endl;
std::cout << a.front() << std::endl;
std::cout << a.back() << std::endl;
```

prints

5 4 3 2 1

1

5

### 2.1.16 `at()`

Functions

```
basic_array<T>::reference  
basic_array<T>::at(size_type n) throw (cvmexception);
```

```
basic_array<T>::const_reference  
basic_array<T>::at(size_type n) throw (cvmexception);
```

return reference to (n-1)-th element of calling array, i.e., unlike [indexing operators](#), these functions are *0-based*. They are provided for every class of the library to be compatible with STL algorithms and methods. Functions throw [cvmexception](#) in case of negative parameter passed. See also [basic\\_array](#) and [STL documentation](#) for further details. Example:

```
using namespace cvm;  
  
iarray a(5);  
a[1] = 1; a[2] = 2; a[3] = 3; a[4] = 4; a[5] = 5;  
std::cout << a.at(0) << " " << a.at(1) << std::endl;
```

prints

```
1 2
```

### 2.1.17 `push_back(const T&)`, `pop_back()`

Functions

```
void basic_array<T>::push_back (const T& x) throw (cvmexception);  
void basic_array<T>::pop_back () throw (cvmexception);
```

add and remove an element to/from calling array. They are provided for every class of the library to be compatible with STL algorithms and methods. Since CVM doesn't pre-allocate memory for extra storage, these functions *will require memory reallocation every time they are being executed* and may slow down your application. Please consider usage of `std::vector<T>` in such cases. Functions throw `cvmexception` in case of memory allocation failure. See also [basic\\_array](#) and [STL documentation](#) for further details. Example:

```
using namespace cvm;
```

```
iarray a(5);  
a.push_back(77);  
std::cout << a;  
a.pop_back();  
std::cout << a;
```

prints

```
0 0 0 0 0 77  
0 0 0 0 0
```

**2.1.18 `insert (iterator, const T&), erase (iterator)`**

Functions

```
basic_array<T>::iterator  
basic_array<T>::insert (iterator pos, const T& x) throw (cvmexception);
```

```
basic_array<T>::iterator  
basic_array<T>::erase (iterator pos) throw (cvmexception);
```

`insert` and `remove` an element to/from calling array at given position `pos`. They are provided for every class of the library to be compatible with STL algorithms and methods. Since CVM doesn't pre-allocate memory for extra storage, these functions *will require memory reallocation every time they are executed* and may slow down your application. Please consider usage of `std::vector<T>` in such cases. Functions throw `cvmexception` in case of memory allocation failure. See also [basic\\_array](#) and [STL documentation](#) for further details. Example:

```
using namespace cvm;
```

```
iarray a(5);  
iarray::iterator pos = a.begin() + 2;  
a.insert(pos, 77);  
std::cout << a;  
pos = a.begin() + 1;  
a.erase(pos);  
std::cout << a;
```

prints

```
0 0 77 0 0 0  
0 77 0 0 0
```

**2.1.19 operator >> <> (std::istream& is, basic\_array<T>& aIn)**

Friend template operator

```
template <typename T>
friend std::istream& operator >> <> (std::istream& is,
                                     basic_array<T>& aIn);
```

fills object referenced by aIn with numbers from is stream. Operator is **redefined** in the class **Array**. See also **basic\_array** Example:

```
using namespace cvm;
```

```
try {
    std::ofstream os;
    os.open ("in.txt");
    os << 1 << " " << 2 << std::endl << 3;
    os.close ();

    std::ifstream is("in.txt");
    iarray v(5);
    is >> v;

    std::cout << v;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
1 2 3 0 0
```

**2.1.20 operator << <> (std::ostream& os, const basic\_array<T>& aOut)**

Friend template operator

```
template <typename T>
friend std::ostream& operator << <> (std::ostream& os,
                                     const basic_array<T>& aOut);
```

outputs object referenced by aOut into os stream. Operator is **redefined** in the class **Array**. See also **basic\_array** Example:

```
using namespace cvm;
```

```
iarray v(5);
v(1) = 1;
v(2) = 2;
```

```
std::cout << v;
```

prints

```
1 2 0 0 0
```

## 2.2 Array

This class contains a couple of common for arrays member functions inherited in user classes. This class is not designed to be instantiated.

```
template <typename TR, typename TC>
class Array : public basic_array<TC> {
public:
    int incr () const;
    int indofmax () const;
    int indofmin () const;
    virtual TR norm () const;
    virtual TR norminf () const;
    virtual TR norm1 () const;
    virtual TR norm2 () const;

    <typename TR, typename TC>
    friend std::istream& operator >> <> (std::istream& is,
                                          Array<TR,TC>& aIn);

    <typename TR, typename TC>
    friend std::ostream& operator << <> (std::ostream& os,
                                          const Array<TR,TC>& aOut);
};
```



### 2.2.1 incr

Function

```
int Array<TR,TC>::incr () const;
```

returns increment between elements of calling array. This function is *inherited* in all classes of the library: `rvector`, `cvector`, `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srbmatrix`, `scbmatrix`, `srsmatrix` and `schmatrix`. It always returns 1 for matrices. See also `Array`.  
Example:

```
using namespace cvm;  
  
double a[] = {1., 2., 3., 4., 5., 6.};  
rvector v1 (a, 3, 2);  
rvector v2(10);  
  
std::cout << v1 << v1.incr () << std::endl;  
std::cout << v2.incr () << std::endl;
```

prints

```
1 3 5  
2  
1
```

### 2.2.2 indofmax

Function

```
int Array<TR,TC>::indofmax () const;
```

returns *l*-based index of calling array's element with maximum absolute value. Function is *inherited*<sup>3</sup> in all classes of the library: `rvector`, `cvector`, `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srbmatrix`, `scbmatrix`, `srsmatrix` and `schmatrix`. See also `Array`. Example:

```
using namespace cvm;
```

```
double a[] = {3., 2., -5., -4., 5., -6.};  
const rvector v (a, 4);  
const rmatrix m (a, 2, 3);
```

```
std::cout << v << v.indofmax () << std::endl << std::endl;  
std::cout << m << m.indofmax () << std::endl;
```

prints

```
3 2 -5 -4  
3
```

```
3 -5 5  
2 -4 -6  
6
```

---

<sup>3</sup>Calls *virtual function* inside

### 2.2.3 indofmin

Function

```
int Array<TR,TC>::indofmin () const;
```

returns *l*-based index of calling array's element with minimum absolute value. Function is *inherited*<sup>4</sup> in all classes of the library: `rvector`, `cvector`, `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srbmatrix`, `scbmatrix`, `srsmatrix` and `schmatrix`. See also `Array`. Example:

```
using namespace cvm;
```

```
double a[] = {3., 2., -5., 0., 0., -6.};  
const rvector v (a, 4);  
const rmatrix m (a, 2, 3);
```

```
std::cout << v << v.indofmin () << std::endl << std::endl;  
std::cout << m << m.indofmin () << std::endl;
```

prints

```
3 2 -5 0  
4
```

```
3 -5 0  
2 0 -6  
4
```

---

<sup>4</sup>Calls *virtual function* inside

**2.2.4 norm**

Virtual function

```
virtual TR Array<TR,TC>::norm() const;
```

returns Euclidean norm of calling array that for vectors is defined as

$$\|x\|_E = \left( \sum_{i=1}^n |x_i|^2 \right)^{1/2}$$

and for matrices as

$$\|A\|_E = \left( \sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2 \right)^{1/2},$$

where  $A$  is  $m \times n$  matrix. Function is *inherited* in the following classes of the library: `rvector`, `cvector`, `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srsmatrix` and `schmatrix`. It's *redefined* in `srbmatrix` and `scbmatrix`. See also `rvector::normalize`, `cvector::normalize`, `rmatrix::normalize`, `cmatrix::normalize` and `Array`. Example:

```
using namespace cvm;
std::cout.setf (ios::scientific | ios::showpos);
std::cout.precision (12);

double a[] = {1., 2., 3., -4., 5., -6.};
const rvector v (a, 3);
const rmatrix m (a, 2, 3);

std::cout << v << v.norm () << std::endl << std::endl;
std::cout << m << m.norm () << std::endl;
```

prints

```
+1.000000000000e+000 +2.000000000000e+000 +3.000000000000e+000
+3.741657386774e+000

+1.000000000000e+000 +3.000000000000e+000 +5.000000000000e+000
+2.000000000000e+000 -4.000000000000e+000 -6.000000000000e+000
+9.539392014169e+000
```

### 2.2.5 norminf

Virtual function

```
virtual TR Array<TR,TC>::norminf () const;
```

returns infinity norm of calling array that for vectors is defined as

$$\|x\|_{\infty} = \max_{i=1,\dots,n} |x_i|$$

and for matrices as

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

where  $A$  is  $m \times n$  matrix. Function is *inherited* in `rvector` and `cvector` classes of the library. It's *redefined* in `Matrix`, `srbmatrix`, `scbmatrix`, `srsmatrix` and `schmatrix`. See also `Matrix::norm1`, `Array`. Example:

```
using namespace cvm;

double a[] = {1., 2., 3., -4., 5., -6.};
const rvector v (a, 3);
const rmatrix m (a, 2, 3);

std::cout << v << v.norminf () << std::endl;
std::cout << m << m.norminf () << std::endl;
```

prints

```
1 2 3
3

1 3 5
2 -4 -6
12
```

**2.2.6 norm1**

Virtual function

```
virtual TR Array<TR,TC>::norm1 () const;
```

returns 1-norm of calling array that for vectors is defined as

$$\|x\|_1 = \sum_{i=1}^n |x_i|$$

and for matrices as

$$\|A\|_1 = \max_{j=1,\dots,n} \sum_{i=1}^m |a_{ij}|,$$

where  $x$  is vector of size  $n$  and  $A$  is  $m \times n$  matrix. Function is *inherited* in **rvector** and **cvector** classes. It's *redefined* in **rmatrix** and **cmatrix** and inherited thereafter. See also **Array**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (12);

double a[] = {1., 2., 3., -4., 5., -6.};
const rvector v (a, 3);
const rmatrix m (a, 2, 3);

std::cout << v << v.norm1 () << std::endl << std::endl;
std::cout << m << m.norm1 () << std::endl;

prints

+1.000000000000e+000 +2.000000000000e+000 +3.000000000000e+000
+6.000000000000e+000

+1.000000000000e+000 +3.000000000000e+000 +5.000000000000e+000
+2.000000000000e+000 -4.000000000000e+000 -6.000000000000e+000
+1.100000000000e+001
```

**2.2.7 norm2**

Virtual function

```
virtual TR Array<TR,TC>::norm2 () const;
```

returns 2-norm of calling array that for vectors is defined as

$$\|x\|_2 = \|x\|_E = \left( \sum_{i=1}^n |x_i|^2 \right)^{1/2}$$

and for matrices as

$$\|A\|_2 = \max_i \sigma_i = \left( \max_{|x|=1} (Ax \cdot Ax) \right)^{1/2},$$

where  $\sigma_i$  is  $i$ -th singular value of  $m \times n$  matrix  $A$ ,  $i = 1, \dots, \min(m, n)$ . Function is *inherited* in `rvector` and `cvector` classes. It's *redefined* in `rmatrix` and `cmatrix` and inherited thereafter. See also `Array`. Example:

```
using namespace cvm;
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (12);

double a[] = {1., 2., 3., -4., 5., -6.};
const rvector v (a, 3);
const rmatrix m (a, 2, 3);

std::cout << v << v.norm2 () << std::endl << std::endl;
std::cout << m << m.norm2 () << std::endl;

prints

+1.000000000000e+000 +2.000000000000e+000 +3.000000000000e+000
+3.741657386774e+000

+1.000000000000e+000 +3.000000000000e+000 +5.000000000000e+000
+2.000000000000e+000 -4.000000000000e+000 -6.000000000000e+000
+9.319612060784e+000
```

### 2.2.8 operator >> <> (std::istream& is, Array<TR,TC>& aIn)

Friend template operator

```
template <typename TR, typename TC>
friend std::istream& operator >> <> (std::istream& is,
                                       Array<TR,TC>& aIn);
```

fills calling object referenced by parameter aIn with numbers from is stream. See also [basic\\_array::operator >>](#) , [Array](#). Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    std::ofstream os;
    os.open ("in.txt");
    os << 1.2 << " " << 2.3 << std::endl << 3.4;
    os.close ();

    std::ifstream is("in.txt");
    rvector v(5);
    is >> v;

    std::cout << v;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

1.20e+000 2.30e+000 3.40e+000 0.00e+000 0.00e+000
```



### 2.2.9 operator << <> (std::ostream& os, const Array<TR,TC>& aOut)

Friend template operator

```
template <typename TR, typename TC>
friend std::ostream& operator << <> (std::ostream& os,
                                     const Array<TR,TC>& aOut);
```

outputs calling object referenced by aOut into os stream. Operator is **redefined** in the class **Matrix**. See also **basic\_array::operator << , Array**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

cvector v(3);
v(1) = tcomplex (1., 2.);
v(2) = tcomplex (3., 4.);

std::cout << v;

prints

(1.00e+000,2.00e+000) (3.00e+000,4.00e+000) (0.00e+000,0.00e+000)
```

## 2.3 rvector

This is end-user class encapsulating vector of real numbers.

```

template <typename TR>
class rvector : public Array<TR,TR> {
public:
    rvector ();
    explicit rvector (int nSize);
    rvector (int nSize, TR d);
    rvector (TR* pD, int nSize, int nIncr = 1);
    rvector (const TR* pD, int nSize, int nIncr = 1);
    rvector (const rvector& v);
    rvector& operator = (const rvector& v) throw (cvmexception);
    rvector& assign (const TR* p, int nIncr = 1);
    rvector& assign (int n, const TR* p, int nIncr = 1);
    rvector& assign (int n, const TR* p, int nSize, int nIncr);
    rvector& assign (int n, const rvector& v) throw (cvmexception);
    rvector& set (TR x);
    rvector& resize (int nNewSize) throw (cvmexception);
    bool operator == (const rvector& v) const;
    bool operator != (const rvector& v) const;
    rvector& operator << (const rvector& v) throw (cvmexception);
    rvector operator + (const rvector& v) const throw (cvmexception);
    rvector operator - (const rvector& v) const throw (cvmexception);
    rvector& sum (const rvector& v1,
                 const rvector& v2) const throw (cvmexception);
    rvector& diff (const rvector& v1,
                 const rvector& v2) const throw (cvmexception);
    rvector& operator += (const rvector& v) throw (cvmexception);
    rvector& operator -= (const rvector& v) throw (cvmexception);
    rvector operator - () const;
    rvector operator * (TR d) const throw (cvmexception);
    rvector operator / (TR d) const throw (cvmexception);
    rvector& operator *= (TR d);
    rvector& operator /= (TR d) throw (cvmexception);
    rvector& normalize ();
    TR operator * (const rvector& v) const throw (cvmexception);
    rvector operator * (const rmatrix& m) const
        throw (cvmexception);
    rvector& mult (const rvector& v, const rmatrix& m)
        throw (cvmexception);
    rvector& mult (const rmatrix& m, const rvector& v)

```

```

        throw (cvmexception);
rmatrix ranklupdate (const rvector& v) const;
rvector& solve (const srmatrix& mA,
               const rvector& vB, TR& dErr)
               throw (cvmexception);
rvector& solve (const srmatrix& mA,
               const rvector& vB) throw (cvmexception);
rvector& solve_tran (const srmatrix& mA,
                   const rvector& vB, TR& dErr)
                   throw (cvmexception);
rvector& solve_tran (const srmatrix& mA,
                   const rvector& vB) throw (cvmexception);
rvector operator / (const srmatrix& mA) const throw (cvmexception);
rvector operator % (const srmatrix& mA) const throw (cvmexception);
rvector& solve_lu (const srmatrix& mA, const srmatrix& mLU,
                 const int* pPivots, const rvector& vB, TR& dErr)
                 throw (cvmexception);
rvector& solve_lu (const srmatrix& mA, const srmatrix& mLU,
                 const int* pPivots, const rvector& vB)
                 throw (cvmexception);
rvector& gels (bool transpose, const rmatrix& mA, const rvector& mB,
              TR& dErr) throw (cvmexception);
rvector& gelsy (const rmatrix& mA, const rvector& mB,
               int& rank, TR tol = cvmMachSp())
               throw (cvmexception);
rvector& gelss (const rmatrix& mA, const rvector& mB,
               rvector& sv, int& rank, TR tol = cvmMachSp())
               throw (cvmexception);
rvector& gelsd (const rmatrix& mA, const rvector& mB,
               rvector& sv, int& rank, TR tol = cvmMachSp())
               throw (cvmexception);
rvector& svd (const rmatrix& mArg) throw (cvmexception);
rvector& svd (const cmatrix& mArg) throw (cvmexception);
rvector& svd (const rmatrix& mArg,
             srmatrix& mU, srmatrix& mVH) throw (cvmexception);
rvector& svd (const cmatrix& mArg,
             scmatrix& mU, scmatrix& mVH) throw (cvmexception);
rvector& eig (const srmatrix& mArg) throw (cvmexception);
rvector& eig (const srmatrix& mArg,
             srmatrix& mEigVect) throw (cvmexception);
rvector& eig (const schmatrix& mArg) throw (cvmexception);
rvector& eig (const schmatrix& mArg,
             scmatrix& mEigVect) throw (cvmexception);

```

```
rvector& gemv (bool bLeft, const rmatrix& m, TR dAlpha,  
             const rvector& v, TR dBeta) throw (cvmexception);  
rvector& gbm (bool bLeft, const srbmatrix& m, TR dAlpha,  
            const rvector& v, TR dBeta) throw (cvmexception);  
rvector& randomize (TR dFrom, TR dTo);  
};
```

### 2.3.1 *rvector* ()

Constructor

```
rvector::rvector ();
```

creates empty *rvector* object. See also *rvector*. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);
```

```
rvector v;  
std::cout << v.size() << std::endl;
```

```
v.resize (5);  
v(1) = 1.5;  
std::cout << v;
```

prints

```
0  
1.50e+000 0.00e+000 0.00e+000 0.00e+000 0.00e+0000
```

### 2.3.2 rvector (int)

Constructor

```
explicit rvector::rvector (int nSize);
```

creates rvector object of size equal to nSize. It throws `cvmexception` in case of non-positive size passed or memory allocation failure. See also `rvector`. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);
```

```
rvector v (5);  
std::cout << v;
```

prints

```
0.00e+000 0.00e+000 0.00e+000 0.00e+000 0.00e+000
```

### 2.3.3 rvector (int, TR)

Constructor

```
rvector::rvector (int nSize, TR d);
```

creates rvector object of size equal to nSize and fills it with value of d. It throws `cvmexception` in case of non-positive size passed or memory allocation failure. See also `rvector`. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);
```

```
rvector v (5, 1.5);  
std::cout << v;
```

prints

```
1.50e+00 1.50e+00 1.50e+00 1.50e+00 1.50e+00
```

### 2.3.4 rvector (TR\*,int,int)

Constructor

```
rvector::rvector (TR* pD, int nSize, int nIncr = 1);
```

creates rvector object of size equal to nSize. Unlike others, this constructor *does not allocate memory*. It just shares memory with array pointed to by pD using distance between elements equal to nIncr. Constructor throws `cvmexception` in case of non-positive size passed. See also `rvector`, `rvector (const TR*,int,int)`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
double a[] = {7., 7., 7., 7., 7., 7., 7.,};
```

```
rvector v1 (a, 4, 2);
```

```
std::cout << v1;
v1(2) = 1.23;
std::cout << v1 << std::endl;
```

```
for (int i = 0; i < 3; i++) {
    std::cout << a[i] << " ";
}
std::cout << std::endl;
```

```
rvector v2 (a, 5);
std::cout << v2;
```

prints

```
7.00e+000 7.00e+000 7.00e+000 7.00e+000
7.00e+000 1.23e+000 7.00e+000 7.00e+000

7.00e+000 7.00e+000 1.23e+000
7.00e+000 7.00e+000 1.23e+000 7.00e+000 7.00e+000
```



### 2.3.5 rvector (const TR\*,int,int)

Constructor

```
rvector::rvector (const TR* pD, int nSize, int nIncr = 1);
```

creates rvector object of size equal to nSize and copies nSize elements of array pD to it using increment nIncr (1 by default). Object created has increment set to 1. Constructor throws `cvmexception` in case of non-positive size passed. See also `rvector`, `rvector (TR*,int,int)`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
const double a[] = {7., 7., 7., 7., 7., 7., 7.,};
```

```
rvector v1 (a, 4, 2);
```

```
std::cout << v1;
v1(2) = 1.23;
std::cout << v1 << std::endl;
```

```
for (int i = 0; i < 3; i++) {
    std::cout << a[i] << " ";
}
std::cout << std::endl;
```

```
rvector v2 (a, 5);
std::cout << v2;
```

prints

```
7.00e+000 7.00e+000 7.00e+000 7.00e+000
7.00e+000 1.23e+000 7.00e+000 7.00e+000

7.00e+000 7.00e+000 7.00e+000
7.00e+000 7.00e+000 7.00e+000 7.00e+000 7.00e+000
```

### 2.3.6 rvector (const rvector&)

Copy constructor

```
rvector::rvector (const rvector& v);
```

creates rvector object of size equal to size of vector v and sets every element of created vector to be equal to appropriate element of v. Constructor throws `cvmexception` in case of memory allocation failure. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7.,};
const rvector v (a, 4, 2);
rvector vc (v);
```

```
vc(1) = 77.;
std::cout << vc;
std::cout << v;
```

prints

```
7.70e+001 3.00e+000 5.00e+000 7.00e+000
1.00e+000 3.00e+000 5.00e+000 7.00e+000
```

### 2.3.7 operator = (const rvector&)

Operator

```
rvector& rvector::operator = (const rvector& v) throw (cvmexception);
```

sets every element of calling vector to be equal to appropriate element of vector v and returns reference to the vector changed. Operator throws `cvmexception` in case of different vector sizes. See also `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    double a[] = {1., 2., 3., 4., 5.};
    const rvector v (a, 5);
    rvector vc(5);

    vc = v;
    std::cout << vc;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
1.00e+000 2.00e+000 3.00e+000 4.00e+000 5.00e+000
```

### 2.3.8 assign(const TR\*, int)

Function

```
rvector& rvector::assign (const TR* p, int nIncr = 1);
```

sets every element of calling vector to be equal to every nIncr-th element of array pointed to by parameter p and returns reference to the vector changed. See also [rvector](#). Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
const double a[] = {1., 2., 3., 4., 5., 6., 7.,};
rvector v (5);
rvector v2 (4);
```

```
v.assign(a);
v2.assign(a, 2);
std::cout << v;
std::cout << v2;
```

prints

```
1.00e+000 2.00e+000 3.00e+000 4.00e+000 5.00e+000
1.00e+000 3.00e+000 5.00e+000 7.00e+000
```

### 2.3.9 assign(int, const TR\*, int)

Function

```
rvector& rvector::assign (int n, const TR* p, int nIncr = 1);
```

sets every element of calling vector, starting from  $\ell$ -based  $n$ -th one, to be equal to every  $nIncr$ -th element of array pointed to by parameter  $p$  and returns reference to the vector changed. See also `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
const double a[] = {1., 2., 3., 4., 5.};
rvector v (5);
rvector v2 (5);
```

```
v.assign(2, a);
v2.assign(3, a, 2);
std::cout << v;
std::cout << v2;
```

prints

```
0.00e+000 1.00e+000 2.00e+000 3.00e+000 4.00e+000
0.00e+000 0.00e+000 1.00e+000 3.00e+000 5.00e+000
```

### 2.3.10 assign(int, const TR\*, int, int)

Function

```
rvector& rvector::assign (int n, const TR* p, int nSize, tint nIncr);
```

sets every element of calling vector, starting from *l*-based *n*-th one, up to *nSize* total, to be equal to every *nIncr*-th element of array pointed to by parameter *p* and returns reference to the vector changed. If *n* + *nSize* goes beyond vector boundaries assignment stops at the last element. See also [rvector](#). Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
const double a[] = {1., 2., 3., 4., 5., 6., 7.};
rvector v (5);
```

```
v.assign(2, a, 5, 2);
std::cout << v;
v.assign(2, a, 3, 1);
std::cout << v;
```

prints

```
0.00e+000 1.00e+000 3.00e+000 5.00e+000 7.00e+000
0.00e+000 1.00e+000 2.00e+000 3.00e+000 7.00e+000
```

### 2.3.11 assign (int, const rvector&)

Function

```
rvector& rvector::assign (int n, const rvector& v) throw (cvmexception);
```

sets every element of calling vector's sub-vector beginning with *l*-based index *n* to vector *v* and returns reference to the vector changed. Function throws `cvmexception` if *n* is not positive or `v.size()+n-1` is greater than calling vector's size. See also `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
rvector v1(5);
rvector v2(2);
v1.set(1.);
v2.set(2.);
v1.assign(3, v2);
std::cout << v1;
```

prints

```
1.00e+00 1.00e+00 2.00e+00 2.00e+00 1.00e+00
```

### 2.3.12 set(TR)

Function

```
rvector& rvector::set (TR x);
```

sets every element of calling vector to be equal to parameter `x` and returns reference to the vector changed. See also [rvector](#). Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);
```

```
rvector v(5);  
v.set(3.);  
std::cout << v;
```

prints

```
3.00e+000 3.00e+000 3.00e+000 3.00e+000 3.00e+000
```



### 2.3.13 `resize`

Function

```
rvector& rvector::resize (int nNewSize) throw (cvmexception);
```

changes size of calling vector to be equal to `nNewSize` and returns reference to the vector changed. In case of increasing of its size, the vector is filled up with zeroes. Function throws `cvmexception` in case of negative size passed or memory allocation failure. See also `basic_array::resize` and `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    double a[] = {1., 2., 3., 4.};
    rvector v (a, 3);
    std::cout << v;
    v.resize(2);
    std::cout << v;
    v.resize(4);
    std::cout << v;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
1.00e+00 2.00e+00 3.00e+00
1.00e+00 2.00e+00
1.00e+00 2.00e+00 0.00e+00 0.00e+00
```

### 2.3.14 operator ==

Operator

```
bool rvector::operator == (const rvector& v) const;
```

compares calling vector with vector `v` and returns `true` if they have same sizes and their appropriate elements differ by not more than the **smallest normalized positive number**. Returns `false` otherwise. See also `rvector`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4.};  
rvector v1 (a, 4);  
rvector v2 (4);
```

```
v2 (1) = 1.; v2 (2) = 2.;
```

```
v2 (3) = 3.; v2 (4) = 4.;
```

```
cout << (v1 == v2) << endl;
```

prints

```
1
```

### 2.3.15 operator !=

Operator

```
bool rvector::operator != (const rvector& v) const;
```

compares calling vector with vector `v` and returns `true` if they have different sizes or at least one of their appropriate elements differs by more than the **smallest normalized positive number**. Returns `false` otherwise. See also `rvector`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4.};  
rvector v1 (a, 4);  
rvector v2 (4);
```

```
v2 (1) = 1.; v2 (2) = 2.;  
v2 (3) = 3.; v2 (4) = 4.;
```

```
cout << (v1 != v2) << endl;
```

prints

```
0
```

### 2.3.16 operator <<

Operator

```
rvector& rvector::operator << (const rvector& v) throw (cvmexception);
```

destroys calling vector, creates a new one as a copy of v and returns reference to the vector changed. Operator throws `cvmexception` in case of memory allocation failure. See also `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    rvector v (5);
    rvector vc (3);
    v(1) = 1.;
    v(2) = 2.;
    std::cout << v << vc << std::endl;

    vc << v;
    std::cout << vc;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
1.00e+000 2.00e+000 0.00e+000 0.00e+000 0.00e+000
0.00e+000 0.00e+000 0.00e+000

1.00e+000 2.00e+000 0.00e+000 0.00e+000 0.00e+000
```

### 2.3.17 operator +

Operator

```
rvector rvector::operator + (const rvector& v) const throw (cvmexception);
```

creates object of type rvector as sum of calling vector and vector v. Operator throws **cvmexception** in case of different sizes of the operands or memory allocation failure. See also **rvector**, **rvector::sum**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    double a[] = {1., 2., 3., 4.};
    const double b[] = {3., 5., 7., 9.};
    const rvector va (a, 4);
    rvector vb (4);
    vb.assign(b);

    std::cout << va + vb;
    std::cout << va + va;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
4.00e+000 7.00e+000 1.00e+001 1.30e+001
2.00e+000 4.00e+000 6.00e+000 8.00e+000
```

### 2.3.18 operator -

Operator

```
rvector rvector::operator - (const rvector& v) const throw (cvmexception);
```

creates object of type rvector as difference of calling vector and vector v. It throws **cvmexception** in case of different sizes of the operands or memory allocation failure. See also **rvector::diff**, **rvector**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    double a[] = {1., 2., 3., 4.};
    const double b[] = {3., 5., 7., 9.};
    const rvector va (a, 4);
    rvector vb (4);
    vb.assign(b);

    std::cout << va - vb;
    std::cout << va - va;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

-2.00e+000 -3.00e+000 -4.00e+000 -5.00e+000
0.00e+000 0.00e+000 0.00e+000 0.00e+000
```

### 2.3.19 sum

Function

```
rvector& rvector::sum (const rvector& v1, const rvector& v2)
throw (cvmexception);
```

assigns result of addition of vectors v1 and v2 to calling vector and returns reference to the vector changed. It throws `cvmexception` in case of different sizes of the operands. See also `rvector::operator +`, `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    double a[] = {1., 2., 3., 4.};
    double b[] = {2., 3., 4., 5.};
    rvector va (a, 4);
    rvector vb (b, 4);
    rvector v (4);

    std::cout << v.sum(va, vb);
    std::cout << v.sum(v, va);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

3.00e+00 5.00e+00 7.00e+00 9.00e+00
4.00e+00 7.00e+00 1.00e+01 1.30e+01
```

### 2.3.20 diff

Function

```
rvector& rvector::diff (const rvector& v1, const rvector& v2)
throw (cvmexception);
```

assigns result of subtraction of vectors v1 and v2 to calling vector and returns reference to the vector changed. It throws `cvmexception` in case of different sizes of the operands. See also `rvector::operator -`, `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    double a[] = {1., 2., 3., 4.};
    double b[] = {2., 3., 4., 5.};
    rvector va (a, 4);
    rvector vb (b, 4);
    rvector v (4);

    std::cout << v.diff(va, vb);
    std::cout << v.diff(v, va);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
-1.00e+00 -1.00e+00 -1.00e+00 -1.00e+00
-2.00e+00 -3.00e+00 -4.00e+00 -5.00e+00
```



### 2.3.21 operator +=

Operator

```
rvector& rvector::operator += (const rvector& v) throw (cvmexception);
```

adds vector *v* to calling vector and returns reference to the vector changed. It throws `cvmexception` in case of different sizes of the operands. See also `rvector::operator +` , `rvector::sum`, `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    rvector v1 (4);
    rvector v2 (4);
    v1.set(1.);
    v2.set(2.);

    v1 += v2;
    std::cout << v1;

    // well, you can do this too, but temporary object would be created
    v2 += v2;
    std::cout << v2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

3.00e+00 3.00e+00 3.00e+00 3.00e+00
4.00e+00 4.00e+00 4.00e+00 4.00e+00
```

### 2.3.22 operator -=

Operator

```
rvector& rvector::operator -= (const rvector& v) throw (cvmexception);
```

subtracts vector *v* from calling vector and returns reference to the vector changed. It throws **cvmexception** in case of different sizes of the operands. See also **rvector::operator -** , **rvector::diff**, **rvector**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    rvector v1 (4);
    rvector v2 (4);
    v1.set(1.);
    v2.set(2.);

    v1 -= v2;
    std::cout << v1;

    // well, you can do this too, but temporary object would be created
    v2 -= v2;
    std::cout << v2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

-1.00e+00 -1.00e+00 -1.00e+00 -1.00e+00
0.00e+00 0.00e+00 0.00e+00 0.00e+00
```

### 2.3.23 operator - ()

Operator

```
rvector rvector::operator - () const throw (cvmexception);
```

creates object of type rvector as calling vector multiplied by  $-1$ . It throws `cvmexception` in case of memory allocation failure. See also `rvector`. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);
```

```
double a[] = {1., 2., 3., 4.};  
const rvector v (a, 4);
```

```
std::cout << - v;
```

prints

```
-1.00e+00 -2.00e+00 -3.00e+00 -4.00e+00
```

### 2.3.24 operator \* (TR)

Operator

```
rvector rvector::operator * (TR d) const throw (cvmexception);
```

creates object of type rvector as product of calling vector and number d. It throws `cvmexception` in case of memory allocation failure. See also `rvector::operator *`, `rvector`. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);
```

```
double a[] = {1., 2., 3., 4.};  
const rvector v (a, 4);
```

```
std::cout << v * 5.;
```

prints

```
5.00e+00 1.00e+01 1.50e+01 2.00e+01
```

### 2.3.25 operator / (TR)

Operator

```
rvector rvector::operator / (TR d) const throw (cvmexception);
```

creates object of type rvector as quotient of calling vector and number d. It throws **cvmexception** if d has absolute value equal or less than the **smallest normalized positive number**. It also throws exception in case of memory allocation failure. See also **rvector::operator /=, rvector**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    double a[] = {1., 2., 3., 4.};
    const rvector v (a, 4);

    std::cout << v / 2.;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
5.00e-01 1.00e+00 1.50e+00 2.00e+00
```

### 2.3.26 operator \*=

Operator

```
rvector& rvector::operator *= (TR d);
```

multiplies calling vector by number d and returns reference to the vector changed. See also `rvector::operator *`, `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
rvector v (4);
v.set(2.);
v *= 2.
std::cout << v;
```

prints

```
4.00e+00 4.00e+00 4.00e+00 4.00e+00
```

### 2.3.27 operator /=

Operator

```
rvector& rvector::operator /= (TR d) throw (cvmexception);
```

divides calling vector by number d and returns reference to the vector changed. It throws **cvmexception** if d has absolute value equal or less than the **smallest normalized positive number**. See also **rvector::operator /** , **rvector**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    rvector v (4);
    v.set(3.);
    v /= 2.;
    std::cout << v;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
1.50e+00 1.50e+00 1.50e+00 1.50e+00
```

### 2.3.28 normalize

Function

```
rvector& rvector::normalize ();
```

normalizes calling vector so its **Euclidean norm** becomes equal to 1 if it was greater than the **smallest normalized positive number** before the call (otherwise it does nothing). See also **rvector**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
rvector v(4);
v(1) = 1.;
v(2) = 2.;
v(3) = 3.;
v(4) = 4.;
std::cout << v.normalize();
std::cout << v.norm() << std::endl;
```

prints

```
1.83e-01 3.65e-01 5.48e-01 7.30e-01
1.00e+00
```



### 2.3.29 operator \* (const rvector&)

Operator

TR rvector::operator \* (const rvector& v) const throw (cvmexception);

returns scalar product of calling vector and v. It throws `cvmexception` if the operands have different sizes. See also `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    double a[] = {1., 2., 3., 4.};
    rvector v1(4);
    rvector v2(4);
    v1.assign(a);
    v2.assign(a);

    std::cout << v1 * v2 << std::endl;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
3.00e+01
```

### 2.3.30 operator \* (const rmatrix&)

Operator

```
rvector rvector::operator * (const rmatrix& m) const  
throw (cvmexception);
```

creates object of type rvector as product of calling vector and matrix m. Use `mult` function to avoid new object creation. Operator throws `cvmexception` if calling vector's size differs from number of rows in matrix m. See also `rvector`, `rmatrix`. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);  
  
try {  
    rvector v(2);  
    rmatrix m(2, 3);  
    v.set(2.);  
    m.set(1.);  
  
    std::cout << v * m;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}  
  
prints  
4.00e+00 4.00e+00 4.00e+00
```

### 2.3.31 mult (const rvector&, const rmatrix&)

Function

```
rvector& rvector::mult (const rvector& v, const rmatrix& m)
throw (cvmexception);
```

sets calling vector to be equal to product of vector v by matrix m and returns reference to the object changed. See also `rvector::mult (const rmatrix&, const rvector&)`, `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    rvector v2(2), v3(3);
    rmatrix m(2, 3);
    v2.set(2.);
    m.set(1.);

    std::cout << v3.mult(v2, m);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
4.00e+00 4.00e+00 4.00e+00
```

### 2.3.32 mult (const rmatrix&, const rvector&)

Function

```
rvector& rvector::mult (const rmatrix& m, const rvector& v)
throw (cvmexception);
```

sets calling vector to be equal to product of matrix m by vector v and returns reference to the vector changed. See also `rvector::mult (const rvector&, const rmatrix&)`, `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    rvector v2(2), v3(3);
    rmatrix m(2, 3);
    v3.set(2.);
    m.set(1.);

    std::cout << v2.mult(m, v3);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

6.00e+00 6.00e+00
```

### 2.3.33 rank1update

Function

```
rmatrix rvector::rank1update (const rvector& v) const;
```

creates object of type `rmatrix` as rank-1 update of calling vector and vector `v`. The rank-1 update of vector-column `x` of size `m` and vector-row `y` of size `n` is defined as  $m \times n$  matrix

$$\begin{pmatrix} x_1 y_1 & x_1 y_2 & \cdots & x_1 y_n \\ x_2 y_1 & x_2 y_2 & \cdots & x_2 y_n \\ \dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots\dots \\ x_m y_1 & x_m y_2 & \cdots & x_m y_n \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} (y_1 \ y_2 \ \cdots \ y_n).$$

See also `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
rvector v1(3);
rvector v2(2);
v1.set(2.);
v2.set(2.);
std::cout << v1.rank1update (v2);
```

prints

```
4.00e+00 4.00e+00
4.00e+00 4.00e+00
4.00e+00 4.00e+00
```

### 2.3.34 solve

Functions

```

rvector& rvector::solve (const srmatrix& mA,
                        const rvector& vB, TR& dErr)
                        throw (cvmexception);
rvector& rvector::solve (const srmatrix& mA,
                        const rvector& vB)
                        throw (cvmexception);

```

set calling vector to be equal to solution  $x$  of linear equation  $A * x = b$  where parameter `mA` is square matrix  $A$  and parameter `vB` is vector  $b$ . Every function returns reference to the vector changed. The first version also sets output parameter `dErr` to be equal to the norm of computation error. These functions throw `cvmexception` in case of inappropriate sizes of the objects or when matrix  $A$  is close to cingular. See also `operator % (srmatrix)`, `rvector::solve_tran`, `rvector`, `srmatrix`. Example:

```

using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (12);

try {
    double m[] = {1., -1., 1., 2., -2., 1., 3., -2., 1.};
    double b[] = {1., 2., 3.};
    srmatrix ma(m, 3);
    rvector vb(b, 3);
    rvector vx(3);
    double dErr = 0.;

    std::cout << vx.solve (ma, vb, dErr);
    std::cout << dErr << std::endl;

    std::cout << ma * vx - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

+8.000000000000e+00 -8.000000000000e+00 +3.000000000000e+00
+6.661338147751e-15
+0.000000000000e+00 +0.000000000000e+00 +0.000000000000e+00

```

### 2.3.35 solve\_tran

Functions

```

rvector& rvector::solve_tran (const srmatrix& mA,
                             const rvector& vB, TR& dErr)
                             throw (cvmexception);
rvector& rvector::solve_tran (const srmatrix& mA,
                             const rvector& vB)
                             throw (cvmexception);

```

set calling vector to be equal to solution  $x$  of linear equation  $A^T * x = b$  (which is equivalent to  $x * A = b$ ) where parameter `mA` is square matrix  $A$  and parameter `vB` is vector  $b$ . Every function returns reference to the vector changed. The first version also sets output parameter `dErr` to be equal to the norm of computation error. These functions throw `cvmexception` in case of inappropriate sizes of the objects or when matrix  $A$  is close to singular. See also `operator / (srmatrix)`, `rvector::solve`, `rvector`, `srmatrix`. Example:

```

using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (12);

try {
    double m[] = {1., -1., 1., 2., -2., 1., 3., -2., 1.};
    double b[] = {1., 2., 3.};
    srmatrix ma(m, 3);
    rvector vb(b, 3);
    rvector vx(3);
    double dErr = 0.;

    std::cout << vx.solve_tran (ma, vb, dErr);
    std::cout << dErr << std::endl;

    std::cout << vx * ma - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
+1.000000000000e+000 +0.000000000000e+000 +0.000000000000e+000
+7.105427357601e-015
+0.000000000000e+000 +0.000000000000e+000 +0.000000000000e+000

```

**2.3.36 operator / (const srmatrix&)**

Operator

```
rvector operator / (const srmatrix& mA) const throw (cvmexception);
```

returns solution  $x$  of linear equation  $A^T * x = b$  (which is equivalent to  $x * A = b$ ) where parameter `mA` is square matrix  $A$  and calling vector is  $b$ . This operator throws `cvmexception` in case of inappropriate sizes of the objects or when matrix  $A$  is close to singular. See also `rvector::solve_tran`, `rvector`, `srmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (12);

try {
    double m[] = {1., -1., 1., 2., -2., 1., 3., -2., 1.};
    double b[] = {1., 2., 3.};
    srmatrix ma(m, 3);
    rvector vb(b, 3);
    rvector vx(3);

    vx = vb / ma;

    std::cout << vx * ma - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
+0.000000000000e+000 +0.000000000000e+000 +0.000000000000e+000
```



**2.3.37 operator % (const srmatrix&)**

Operator

```
rvector operator % (const srmatrix& mA) const throw (cvmexception);
```

returns solution  $x$  of linear equation  $A * x = b$  where parameter `mA` is square matrix  $A$  and calling vector is  $b$ . This operator throws `cvmexception` in case of inappropriate sizes of the objects or when matrix  $A$  is close to singular. See also `rvector::solve`, `rvector`, `srmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (12);

try {
    double m[] = {1., -1., 1., 2., -2., 1., 3., -2., 1.};
    double b[] = {1., 2., 3.};
    srmatrix ma(m, 3);
    rvector vb(b, 3);
    rvector vx(3);

    vx = vb % ma;

    std::cout << ma * vx - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
+0.000000000000e+000 +0.000000000000e+000 +0.000000000000e+000
```

### 2.3.38 solve\_lu

Functions

```
rvector&
rvector::solve_lu (const srmatrix& mA, const srmatrix& mLU,
                  const int* pPivots, const rvector& vB, TR& dErr)
                  throw (cvmexception);
```

```
rvector&
rvector::solve_lu (const srmatrix& mA, const srmatrix& mLU,
                  const int* pPivots, const rvector& vB)
                  throw (cvmexception);
```

set calling vector to be equal to solution  $x$  of linear equation  $A * x = b$  where parameter `mA` is square matrix  $A$ , parameter `mLU` is [LU factorization](#) of matrix  $A$ , parameter `pPivots` is an array of pivot numbers created while factorizing matrix  $A$  and parameter `vB` is vector  $b$ . Every function returns reference to the vector changed. The first version also sets output parameter `dErr` to be equal to the norm of computation error. These functions are useful when you need to solve few linear equations of kind  $Ax = b$  with the same matrix  $A$  and different vectors  $b$ . In such case you save on matrix  $A$  factorization since it's needed to be performed just one time. These functions throw [cvmexception](#) in case of inappropriate sizes of the objects or when matrix  $A$  is close to singular. See also [rvector](#), [srmatrix](#). Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (12);

try {
    double m[] = {1., -1., 1., 2., -2., 1., 3., -2., 1.};
    double b1[] = {1., 2., 3.};
    double b2[] = {0., -1., -2.};
    srmatrix ma(m, 3);
    srmatrix mLU(3);
    rvector vb1(b1, 3);
    rvector vb2(b2, 3);
    rvector vx1(3);
    rvector vx2(3);
    iarray nPivots(3);
    double dErr = 0.;

    mLU.low_up(ma, nPivots);
    std::cout << vx1.solve_lu (ma, mLU, nPivots, vb1, dErr);
```

```
std::cout << dErr << std::endl;
std::cout << vx2.solve_lu (ma, mLU, nPivots, vb2);
std::cout << ma * vx1 - vb1 << ma * vx2 - vb2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
+8.000000000000000e+000 -8.000000000000000e+000 +3.000000000000000e+000
+6.661338147751e-015
-5.000000000000000e+000 +4.000000000000000e+000 -1.000000000000000e+000
+0.000000000000000e+000 +0.000000000000000e+000 +0.000000000000000e+000
+0.000000000000000e+000 +0.000000000000000e+000 +0.000000000000000e+000
```

### 2.3.39 gels

Function

```
rvector& rvector::gels (bool transpose,
                      const rmatrix& mA, const rvector& mB,
                      TR& dErr) throw (cvmexception);
```

solves overdetermined or underdetermined linear system

$$A * x = b$$

for  $m \times n$  matrix  $A$  (or transposed one) where  $b$  is a vector of length  $k$  and  $k = m$  in non-transposed case and  $k = n$  otherwise. The algorithm uses QR or LQ factorization of  $A$ . It is assumed that  $A$  has full rank, infinity returned otherwise. Internally function uses ?GELS LAPACK routines. If  $m > n$  and `transpose=false` or  $m < n$  and `transpose=true`, then the system is overdetermined, thus the algorithm tries to find the least squares solution  $x$  of the problem

$$\|A * x - b\|_2 \rightarrow \min \quad \text{or} \quad \|A^H * x - b\|_2 \rightarrow \min$$

respectively. Real number `dErr` returns residual sum of squares. The system is underdetermined otherwise, and the algorithm finds its minimum norm solution. In this case `bErr` is set to zero. In both cases the solution computed satisfies  $x = \text{pinv}(A) * b$ , but this algorithm is faster than pseudo inversion. Function sets calling object to be the solution and returns reference to it. It throws `cvmexception` in case of inappropriate sizes of the operands. See also `rvector`, `rmatrix::gels`. Example:

```
using namespace cvm;
rmatrix a(7, 5);
rvector b(7), bt(5);
treal dErr;
a.randomize(-10., 10.);
b.randomize(-10., 10.);
bt.randomize(-10., 10.);

rvector x(5);
x.gels(false, a, b, dErr);
cvm::rvector xt(7);
xt.gels(true, a, bt, dErr);
std::cout << (a.pinv()*b - x).norm() << std::endl;
std::cout << (~a.pinv()*bt - xt).norm() << std::endl;

prints

+5.1347815e-016
+1.1752774e-015
```

**2.3.40 gelsy**

Function

```
rvector& rvector::gelsy (const rmatrix& mA, const rvector& mB, int& rank,
                       TR tol = cvmMachSp()) throw (cvmexception);
```

computes minimum-norm solution to linear least squares problem

$$\|A * x - b\|_2 \rightarrow \min$$

using complete orthogonal factorization of  $m \times n$  matrix  $A$ . Here  $b$  is a vector of length  $m$ . Matrix  $A$  may be rank-deficient, function returns its effective rank in rank output parameter using `tol` tolerance. Internally function uses ?GELSY LAPACK routines, see more details about the algorithm in those routines' documentation. Matrix  $A$  is passed as argument `mA`. Function sets calling object to be the solution and returns reference to it. It throws `cvmexception` in case of inappropriate sizes of the operands. See also `rvector`, `rvector::gelss`, `rvector::gelsd`, `rmatrix::gelsy`. Example:

```
using namespace cvm;
```

```
rmatrix a(4, 5);
rvector b(4);
a.randomize(-10., 10.);
b.randomize(-10., 10.);
rvector x(5);
tint rank;

x.gelsy(a, b, rank);
std::cout << (a*x - b).norm() << std::endl;
std::cout << rank << " " << a.rank() << std::endl;
```

```
prints
```

```
+2.1326688e-015
+4 +4
```

**2.3.41 gelss**

Function

```
rvector& rvector::gelss (const rmatrix& mA, const rvector& mB, int& rank,
                        rvector& sv, TR tol = cvmMachSp())
                        throw (cvmexception);
```

computes minimum-norm solution to linear least squares problem

$$\|A * x - b\|_2 \rightarrow \min$$

using singular value decomposition of  $m \times n$  matrix  $A$ . Here  $b$  is a vector of length  $m$ . Matrix  $A$  may be rank-deficient, function returns its effective rank in `rank` output parameter using `tol` tolerance. This function also computes singular values of  $A$  in decreasing order and returns them in `sv` output parameter having  $\min(m, n)$  size. Internally function uses ?GELSS LAPACK routines, see more details about the algorithm in that routine's documentation. Matrix  $A$  is passed as argument `mA`. Function sets calling object to be the solution and returns reference to it. It throws `cvmexception` in case of inappropriate sizes of the operands. See also `rvector`, `rvector::gelsy`, `rvector::gelsd`, `rmatrix::gelss`. Example:

```
using namespace cvm;

rmatrix a(4, 5);
rvector b(4);
a.randomize(-10., 10.);
b.randomize(-10., 10.);
rvector sv(4);
rvector x(5);
tint rank;

x.gelss(a, b, sv, rank);
std::cout << (a*x - b).norm() << std::endl;
std::cout << (sv - a.svd()).norm() << std::endl;
std::cout << rank << " " << a.rank() << std::endl;

prints

+7.7937340e-015
+1.2338969e-014
+4 +4
```

**2.3.42 gelsd**

Function

```
rvector& rvector::gelsd (const rmatrix& mA, const rvector& mB, int& rank,
                        rvector& sv, TR tol = cvmMachSp())
                        throw (cvmexception);
```

computes minimum-norm solution to linear least squares problem

$$\|A * x - b\|_2 \rightarrow \min$$

using singular value decomposition of  $m \times n$  matrix  $A$  and divide and conquer method. Here  $b$  is a vector of length  $m$ . Matrix  $A$  may be rank-deficient, function returns its effective rank in `rank` output parameter using `tol` tolerance. This function also computes singular values of  $A$  in decreasing order and returns them in `sv` output parameter having  $\min(m, n)$  size. Internally function uses ?GELSD LAPACK routines, see more details about the algorithm in that routine's documentation. Matrix  $A$  is passed as argument `mA`. Function sets calling object to be the solution and returns reference to it. It throws `cvmexception` in case of inappropriate sizes of the operands. See also `rvector`, `rvector::gelsy`, `rvector::gelss`, `rmatrix::gelsd`. Example:

```
using namespace cvm;

rmatrix a(4, 5);
rvector b(4);
a.randomize(-10., 10.);
b.randomize(-10., 10.);
rvector sv(4);
rvector x(5);
tint rank;

x.gelsd(a, b, sv, rank);
std::cout << (a*x - b).norm() << std::endl;
std::cout << (sv - a.svd()).norm() << std::endl;
std::cout << rank << " " << a.rank() << std::endl;

prints

+7.8441794e-015
+5.1021966e-015
+4 +4
```

**2.3.43 svd**

Functions

```

rvector&
rvector::svd (const rmatrix& mArg) throw (cvmexception);
rvector&
rvector::svd (const cmatrix& mArg) throw (cvmexception);

rvector&
rvector::svd (const rmatrix& mArg,
              srmatrix& mU, srmatrix& mVH) throw (cvmexception);
rvector&
rvector::svd (const cmatrix& mArg,
              scmatrix& mU, scmatrix& mVH) throw (cvmexception);

```

set calling vector to be equal to singular values

$$\sigma_1 \geq \sigma_2 \geq \dots \geq \sigma_{\min(m,n)} \geq 0$$

of  $m \times n$  matrix  $A$  (parameter `mArg`). These values are the main diagonal of matrix  $\Sigma$  of the singular value decomposition

$$A = U\Sigma V^H$$

where  $U$  and  $V$  are orthogonal for real  $A$  and unitary for complex  $A$ .  $V^H$  is transposed  $V$  for real one and hermitian conjugated  $V$  for complex one. First  $\min(m, n)$  columns of matrices  $U$  and  $V$  are left and right singular vectors of  $A$  respectively. Singular values and singular vectors satisfy

$$Av_i = \sigma_i u_i \quad \text{and} \quad A^H u_i = \sigma_i v_i$$

where  $u_i$  and  $v_i$  are  $i$ -th columns of  $U$  and  $V$  respectively. Third and fourth versions of functions set output parameter `mU` to be equal to matrix  $U$  of size  $m \times m$  and `mVH` to be equal to matrix  $V^H$  of size  $n \times n$ . All functions return reference to the object they change and throw **cvmexception** in case of inappropriate calling object size (it must be equal to  $\min(m, n)$ ), matrix `mU` size (must be  $m \times m$ ), matrix `mVH` size (must be  $n \times n$ ) or in case of convergence error. See also **rvector**, **rmatrix**, **cmatrix**. Example:

```

using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (10);

try {
    double m[] = {1., -1., 1., 2., -2., 1.,
                  3., -2., 1., 0., -2., 1.};
    rmatrix mA(m,4,3);
    rmatrix mSigma(4,3);

```



```

rvector v(3);
srmatrix mU(4), mVH(3);

v.svd(mA, mU, mVH);
mSigma.diag(0) = v;

std::cout << mU << std::endl;
std::cout << mVH << std::endl;
std::cout << mSigma << std::endl;

std::cout << (mA * ~mVH - mU * mSigma).norm() << std::endl;
std::cout << (~mA * mU - ~(mSigma * mVH)).norm() << std::endl;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
prints
-4.8425643615e-01 +1.9516809011e-01 +1.1506232201e-02 -8.5280286542e-01
+2.1685987119e-01 -3.4107922671e-01 -8.8948423927e-01 -2.1320071636e-01
+6.6237057295e-01 +7.1553688692e-01 -6.1787070600e-02 -2.1320071636e-01
-5.2889765022e-01 +5.7756501033e-01 -4.5262319054e-01 +4.2640143271e-01

-2.2124855498e-01 +8.5354150454e-01 -4.7171599183e-01
+9.5937301747e-01 +1.0365951763e-01 -2.6240830353e-01
-1.7507852602e-01 -5.1060905244e-01 -8.4179920723e-01

+4.9561500411e+00 +0.0000000000e+00 +0.0000000000e+00
+0.0000000000e+00 +2.5088408581e+00 +0.0000000000e+00
+0.0000000000e+00 +0.0000000000e+00 +3.7721919242e-01
+0.0000000000e+00 +0.0000000000e+00 +0.0000000000e+00

+1.3710111285e-15
+2.4829995848e-15

```

**2.3.44 eig**

Functions

```
rvector&
rvector::eig (const srsmatrix& mArg) throw (cvmexception);
rvector&
rvector::eig (const schmatrix& mArg) throw (cvmexception);

rvector&
rvector::eig (const srsmatrix& mArg,
              srmatrix& mEigVect) throw (cvmexception);
rvector&
rvector::eig (const schmatrix& mArg,
              scmatrix& mEigVect) throw (cvmexception);
```

solve symmetric eigenvalue problem and set calling vector to be equal to eigenvalues of symmetric or hermitian matrix `mArg`. The symmetric eigenvalue problem is defined as follows: given symmetric or hermitian matrix  $A$ , find the eigenvalues  $\lambda$  and the corresponding eigenvectors  $z$  that satisfy the equation

$$Az = \lambda z.$$

All  $n$  eigenvalues are real not only for real symmetric but also for complex Hermitian matrices  $A$ , and there exists an orthogonal system of  $n$  eigenvectors. If  $A$  is symmetric or Hermitian positive-definite matrix, all eigenvalues are positive. See [3] for further details. The third and fourth versions of functions set output parameter `mEigVect` to be equal to square matrix containing eigenvectors as columns. All functions return reference to the vector they change and throw `cvmexception` in case of inappropriate calling object sizes or in case of convergence error. See also `rvector`, `cvector::eig`, `srsmatrix`, `schmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (3);

try {
    srsmatrix m(3);
    srmatrix me(3);
    rvector v(3);
    m.randomize(1., 3.);

    v.eig (m, me);
    std::cout << v;
```

```

std::cout << m * me(1) - me(1) * v(1);
std::cout << m * me(2) - me(2) * v(2);
std::cout << m * me(3) - me(3) * v(3);
std::cout << me(1) * me(2) << std::endl; // orthogonality check

schmatrix mc(3);
scmatrix mce(3);
mc.randomize_real(1., 3.);
mc.randomize_imag(1., 3.);

v.eig (mc, mce);
std::cout << v;

std::cout << mc * mce(1) - mce(1) * v(1);
std::cout << mc * mce(2) - mce(2) * v(2);
std::cout << mc * mce(3) - mce(3) * v(3);
std::cout << mce(1) % mce(2) << std::endl; // orthogonality check
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

-9.360e-01 +3.535e-01 +6.376e+00
-4.441e-16 -5.551e-16 -6.106e-16
+3.331e-16 +1.145e-16 +1.110e-16
-4.441e-16 +0.000e+00 -4.441e-16
+2.060e-17
-3.274e+00 +9.710e-01 +8.209e+00
(-4.441e-16,-1.221e-15) (-1.443e-15,-4.441e-16) (-8.882e-16,+4.683e-16)
(-5.551e-16,-2.776e-16) (+0.000e+00,-4.025e-16) (+6.661e-16,-2.461e-17)
(-5.551e-16,+0.000e+00) (+4.441e-16,-4.441e-16) (+0.000e+00,+3.896e-16)
(+1.608e-16,-2.261e-17)

```

**2.3.45 gemv**

Function

```
rvector& rvector::gemv (bool bLeft, const rmatrix& m, TR dAlpha,
                      const rvector& v, TR dBeta) throw (cvmexception);
```

calls one of ?GEMV routines of the [BLAS library](#) performing matrix-vector operation defined as

$$c = \alpha M \cdot v + \beta c \quad \text{or} \quad c = \alpha v \cdot M + \beta c,$$

where  $\alpha$  and  $\beta$  are real numbers (parameters dAlpha and dBeta), M is matrix (parameter m) and v and c are vectors (parameter v and calling vector respectively). First operation is performed if bLeft passed is false and second one otherwise. Function returns reference to the vector changed and throws [cvmexception](#) in case of inappropriate sizes of the operands. See also [rvector](#), [rmatrix](#). Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (7);

try {
    double alpha = 1.3;
    double beta = -0.7;
    rmatrix m(4,3);
    rvector c(4);
    rvector v(3);
    m.randomize(-1., 2.); v.randomize(-1., 3.); c.randomize(0., 2.);

    std::cout << m * v * alpha + c * beta;
    std::cout << c.gemv(false, m, alpha, v, beta);
    std::cout << c * m * alpha + v * beta;
    std::cout << v.gemv(true, m, alpha, c, beta);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

-3.5397829e-02 +3.1990410e-02 +3.2633344e-01 -5.4669713e-01
-3.5397829e-02 +3.1990410e-02 +3.2633344e-01 -5.4669713e-01
-4.7697026e-01 -2.2544922e-01 -5.5204984e-01
-4.7697026e-01 -2.2544922e-01 -5.5204984e-01
```

**2.3.46 gbmv**

Function

```
rvector& rvector::gbmv (bool bLeft, const srbmatrix& m, TR dAlpha,
                      const rvector& v, TR dBeta) throw (cvmexception);
```

calls one of ?GBMV routines of the [BLAS library](#) performing matrix-vector operation defined as

$$c = \alpha M \cdot v + \beta c \quad \text{or} \quad c = \alpha v \cdot M + \beta c,$$

where  $\alpha$  and  $\beta$  are real numbers (parameters dAlpha and dBeta), M is band matrix (parameter m) and v and c are vectors (parameter v and calling vector respectively). First operation is performed if bLeft passed is false and second one otherwise. Function returns reference to the vector changed and throws `cvmexception` in case of inappropriate sizes of the operands. See also [rvector](#), [srbmatrix](#). Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (7);

try {
    double alpha = 1.3;
    double beta = -0.7;
    srbmatrix m(3, 1, 0);
    rvector c(3);
    rvector v(3);
    m.randomize(-1., 2.); v.randomize(-1., 3.); c.randomize(0., 2.);

    std::cout << m * v * alpha + c * beta;
    std::cout << c.gbmv(false, m, alpha, v, beta);
    std::cout << c * m * alpha + v * beta;
    std::cout << v.gbmv(true, m, alpha, c, beta);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
}
```

prints

```
+1.4551599e+00 -5.1882508e-01 -5.2088503e-02
+1.4551599e+00 -5.1882508e-01 -5.2088503e-02
+7.3471591e-01 -2.6952064e-01 -2.0478054e-01
+7.3471591e-01 -2.6952064e-01 -2.0478054e-01
```

### 2.3.47 `randomize`

Function

```
rvector& rvector::randomize (TR dFrom, TR dTo);
```

fills calling vector with pseudo-random numbers distributed between dFrom and dTo. It returns reference to the vector changed. See also `rvector`. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (7);
```

```
rvector v(4);  
v.randomize(-2.,3.);  
std::cout << v;
```

prints

```
-1.1160314e+000 2.5649586e+000 8.9345988e-001 -1.1631825e+000
```

## 2.4 cvector

This is end-user class encapsulating vector of complex numbers.

```

template <typename TR, typename TC>
class cvector : public Array<TR,TC> {
public:
    cvector ();
    explicit cvector (int nSize);
    cvector (int nSize, TC c);
    cvector (TC* pD, int nSize, int nIncr = 1);
    cvector (const TC* pD, int nSize, int nIncr = 1);
    cvector (const cvector& v);
    cvector (const TR* pRe, const TR* pIm, int nSize,
            tint nIncrRe = 1, tint nIncrIm = 1);
    cvector (const rvector& vRe, const rvector& vIm);
    cvector (const TR* pA, int nSize,
            bool bRealPart = true, int nIncr = 1);
    explicit cvector (const rvector& v, bool bRealPart = true);
    rvector real ();
    rvector imag ();
    cvector& operator = (const cvector& v) throw (cvmexception);
    cvector& assign (const TC* p, int nIncr = 1);
    cvector& assign (int n, const TC* p, int nIncr = 1);
    cvector& assign (int n, const TC* p, int nSize, tint nIncr);
    cvector& assign (int n, const cvector& v) throw (cvmexception);
    cvector& set (TC x);
    cvector& assign_real (const rvector& vRe) throw (cvmexception);
    cvector& assign_imag (const rvector& vIm) throw (cvmexception);
    cvector& set_real (TR x);
    cvector& set_imag (TR x);
    cvector& resize (int nNewSize) throw (cvmexception);
    bool operator == (const cvector& v) const;
    bool operator != (const cvector& v) const;
    cvector& operator << (const cvector& v) throw (cvmexception);
    cvector operator + (const cvector& v) const throw (cvmexception);
    cvector operator - (const cvector& v) const throw (cvmexception);
    cvector& sum (const cvector& v1,
                const cvector& v2) const throw (cvmexception);
    cvector& diff (const cvector& v1,
                const cvector& v2) const throw (cvmexception);
    cvector& operator += (const cvector& v) throw (cvmexception);
    cvector& operator -= (const cvector& v) throw (cvmexception);

```

```

cvector operator - () const throw (cvmexception);
cvector operator * (TR d) const;
cvector operator / (TR d) const throw (cvmexception);
cvector operator * (TC c) const;
cvector operator / (TC c) const throw (cvmexception);
cvector& operator *= (TR d);
cvector& operator /= (TR d) throw (cvmexception);
cvector& operator *= (TC c);
cvector& operator /= (TC c) throw (cvmexception);
cvector& normalize ();
cvector operator ~() const throw (cvmexception);
cvector& conj (const cvector& v) throw (cvmexception);
cvector& conj ();
TC operator * (const cvector& v) const throw (cvmexception);
TC operator % (const cvector& v) const throw (cvmexception);
cvector operator * (const cvector& v) const throw (cvmexception);
cvector& mult (const cvector& v, const cmatrix& m)
    throw (cvmexception);
cvector& mult (const cmatrix& m, const cvector& v)
    throw (cvmexception);
cmatrix ranklupdate_u (const cvector& v) const;
cmatrix ranklupdate_c (const cvector& v) const;
cvector& solve (const scmatrix& mA,
    const cvector& vB, TR& dErr) throw (cvmexception);
cvector& solve (const scmatrix& mA,
    const cvector& vB) throw (cvmexception);
rvector& solve_tran (const scmatrix& mA,
    const cvector& vB, TR& dErr)
    throw (cvmexception);
rvector& solve_tran (const scmatrix& mA,
    const cvector& vB) throw (cvmexception);
rvector& solve_conj (const scmatrix& mA,
    const cvector& vB, TR& dErr)
    throw (cvmexception);
rvector& solve_conj (const scmatrix& mA,
    const cvector& vB) throw (cvmexception);
rvector operator / (const scmatrix& mA) const throw (cvmexception);
rvector operator % (const scmatrix& mA) const throw (cvmexception);
cvector& solve_lu (const scmatrix& mA, const scmatrix& mLU,
    const int* pPivots, const cvector& vB, TR& dErr)
    throw (cvmexception);
cvector& solve_lu (const scmatrix& mA, const scmatrix& mLU,
    const int* pPivots, const cvector& vB)

```



```
        throw (cvmexception);
cvector& gels (bool conjugate, const cmatrix& mA, const cvector& mB,
              TC& cErr) throw (cvmexception);
cvector& gelsy (const cmatrix& mA, const cvector& mB,
              int& rank, TR tol = cvmMachSp())
              throw (cvmexception);
rvector& gelss (const cmatrix& mA, const cvector& mB,
              rvector& sv, int& rank, TR tol = cvmMachSp())
              throw (cvmexception);
rvector& gelsd (const cmatrix& mA, const cvector& mB,
              rvector& sv, int& rank, TR tol = cvmMachSp())
              throw (cvmexception);
cvector& eig (const srmatrix& mArg) throw (cvmexception);
cvector& eig (const scmatrix& mArg) throw (cvmexception);
cvector& eig (const srmatrix& mArg, scmatrix& mEigVect,
              bool bRightVect = true) throw (cvmexception);
cvector& eig (const scmatrix& mArg, scmatrix& mEigVect,
              bool bRightVect = true) throw (cvmexception);
cvector& gemv (bool bLeft, const cmatrix& m, TC cAlpha,
              const cvector& v, TC dBeta) throw (cvmexception);
cvector& gbmvm (bool bLeft, const scbmatrix& m, TC dAlpha,
              const cvector& v, TC dBeta) throw (cvmexception);
cvector& randomize_real (TR dFrom, TR dTo);
cvector& randomize_imag (TR dFrom, TR dTo);
};
```

### 2.4.1 `cvector` ()

Constructor

```
cvector::cvector ();
```

creates empty `cvector` object. See also `cvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
cvector v;
std::cout << v.size() << std::endl;
```

```
v.resize (3);
v(1) = std::complex<double>(1.5, -1.);
std::cout << v;
```

prints

```
0
(1.50e+00,-1.00e+00) (0.00e+00,0.00e+00) (0.00e+00,0.00e+00)
```

### 2.4.2 *cvector* (int)

Constructor

```
explicit cvector::cvector (int nSize);
```

creates *cvector* object of size equal to nSize. It throws *cvmexception* in case of non-positive size passed or memory allocation failure. See also *cvector*. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);
```

```
cvector v(3);  
std::cout << v.size() << std::endl;  
std::cout << v;
```

prints

```
3  
(0.00e+00,0.00e+00) (0.00e+00,0.00e+00) (0.00e+00,0.00e+00)
```

### 2.4.3 `cvector` (int, TC)

Constructor

```
cvector::cvector (int nSize, TC c);
```

creates `cvector` object of size equal to `nSize` and fills it with value of `c`. It throws `cvmexception` in case of non-positive size passed or memory allocation failure. See also `cvector`. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);
```

```
cvector v (3, std::complex<double>(1.5, -1.));  
std::cout << v;
```

prints

```
(1.50e+00,-1.00e+00) (1.50e+00,-1.00e+00) (1.50e+00,-1.00e+00)
```

#### 2.4.4 cvector (TC\*,int,int)

Constructor

```
cvector::cvector (TC* pD, int nSize, int nIncr = 1);
```

creates cvector object of size equal to nSize. Unlike others, this constructor *does not allocate memory*. It just shares memory with an array pointed to by pD with distance between elements equal to nIncr. It throws `cvmexception` in case of non-positive size passed. See also `cvector`, `cvector (const TC*,int,int)`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
cvector v1 ((std::complex<double>*) a, 2, 2);

std::cout << v1;
v1(2) = std::complex<double> (9.99, 9.99);
std::cout << v1 << std::endl;

for (int i = 0; i < 6; i++) {
    std::cout << a[i] << " ";
}
std::cout << std::endl;

cvector v2 ((std::complex<double>*) a, 3);
std::cout << v2;

prints

(1.00e+000,2.00e+000) (5.00e+000,6.00e+000)
(1.00e+000,2.00e+000) (9.99e+000,9.99e+000)

1.00e+000 2.00e+000 3.00e+000 4.00e+000 9.99e+000 9.99e+000
(1.00e+000,2.00e+000) (3.00e+000,4.00e+000) (9.99e+000,9.99e+000)
```

### 2.4.5 cvector (const TC\*,int,int)

Constructor

```
cvector::cvector (const TC* pD, int nSize, int nIncr = 1);
```

creates cvector object of size equal to nSize and copies nSize elements of array pD to it using increment nIncr (1 by default). The object created has increment set to 1. It throws `cvmexception` in case of non-positive size passed. See also `cvvector`, `cvvector (TC*,int,int)`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

const double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
cvvector v1 ((const std::complex<double>*) a, 2, 2);

std::cout << v1;
v1(2) = std::complex<double> (9.99, 9.99);
std::cout << v1 << std::endl;

for (int i = 0; i < 6; i++) {
    std::cout << a[i] << " ";
}
std::cout << std::endl;

cvvector v2 ((std::complex<double>*) a, 3);
std::cout << v2;

prints

(1.00e+000,2.00e+000) (5.00e+000,6.00e+000)
(1.00e+000,2.00e+000) (9.99e+000,9.99e+000)

1.00e+000 2.00e+000 3.00e+000 4.00e+000 5.00e+000 6.00e+000
(1.00e+000,2.00e+000) (3.00e+000,4.00e+000) (5.00e+000,6.00e+000)
```

### 2.4.6 cvector (const cvector&)

Copy constructor

```
cvector::cvector (const cvector& v);
```

creates cvector object of size equal to size of vector v and sets every element of created vector to be equal to appropriate element of v. It throws `cvmexception` in case of memory allocation failure. See also `cvector`. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};  
const cvector v ((std::complex<double>*) a, 3, 2);  
cvector vc (v);
```

```
vc(1) = std::complex<double>(7.77,8.88);  
std::cout << vc;  
std::cout << v;
```

prints

```
(7.77e+00,8.88e+00) (5.00e+00,6.00e+00) (9.00e+00,1.00e+01)  
(1.00e+00,2.00e+00) (5.00e+00,6.00e+00) (9.00e+00,1.00e+01)
```

### 2.4.7 cvector (const TR\*,const TR\*,int,int,int)

Constructor

```
cvector::cvector (const TR* pRe, const TR* pIm, int nSize,  
                tint nIncrRe = 1, tint nIncrIm = 1);
```

creates cvector object of size equal to nSize and copies every nIncrRe-th element of array pointed to by pRe and every nIncrIm-th element of array pointed to by pIm to real and imaginary part of the object created. It throws `cvmexception` in case of non-positive size passed or memory allocation failure. See also `cvector`. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);  
  
double re[] = {1., 2., 3., 4., 5.};  
double im[] = {5., 4., 3., 2., 1.};  
cvector v (re, im, 3, 2);  
  
std::cout << v;  
re[0] = 7.77;  
std::cout << v;  
  
const double rec[] = {1., 2., 3.};  
const double imc[] = {5., 4., 3.};  
const cvector vc (rec, imc, 3);  
std::cout << vc;  
  
prints  
  
(1.00e+00,5.00e+00) (3.00e+00,4.00e+00) (5.00e+00,3.00e+00)  
(1.00e+00,5.00e+00) (3.00e+00,4.00e+00) (5.00e+00,3.00e+00)  
(1.00e+00,5.00e+00) (2.00e+00,4.00e+00) (3.00e+00,3.00e+00)
```



### 2.4.8 cvector (const rvector&, const rvector&)

Constructor

```
cvector::cvector (const rvector& vRe, const rvector& vIm);
```

creates cvector object of size equal to `vRe.size()` and `vIm.size()` and copies vectors `vRe` and `vIm` to real and imaginary part of the object created. It throws `cvmexception` in case of non-equal sizes of the parameters passed or memory allocation failure. See also `cvector` and `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

rvector vr(3), vi(3);
vr[1] = 1.;
vr[2] = 2.;
vr[3] = 3.;
vi[1] = 5.;
vi[2] = 4.;
vi[3] = 3.;

const cvector vc(vr, vi);
std::cout << vc;

prints

(1.00e+00,5.00e+00) (2.00e+00,4.00e+00) (3.00e+00,3.00e+00)
```

### 2.4.9 `cvector (const TR*,int,bool,int)`

Constructor

```
cvector::cvector (const TR* pA, int nSize,  
                 bool bRealPart = true, int nIncr = 1);
```

creates `cvector` object of size equal to `nSize` and copies every `nIncr`-th element of array pointed to by `pA` to real (if `bRealPart` is true) or imaginary (if `bRealPart` is false) part of the object created. Constructor throws `cvmexception` in case of non-positive size passed or memory allocation failure. See also `cvector`. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);  
  
const double a[] = {1., 2., 3., 4., 5.};  
cvector v1 (a, 3, false, 2);  
cvector v2 (a, 2);  
  
std::cout << v1 << v2;  
  
prints  
  
(0.00e+00,1.00e+00) (0.00e+00,3.00e+00) (0.00e+00,5.00e+00)  
(1.00e+00,0.00e+00) (2.00e+00,0.00e+00)
```

### 2.4.10 cvector (const rvector&,bool)

Constructor

```
explicit cvector::cvector (const rvector& v, bool bRealPart = true);
```

creates cvector object of size equal to `v.size()` and copies every element of vector `v` to real (if `bRealPart` is true) or imaginary (if `bRealPart` is false) part of the object created. Constructor throws `cvmexception` in case of memory allocation failure. See also `cvector` and `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

rvector vr (3);
vr(1) = 1.;
vr(2) = 2.;
vr(3) = 3.;

cvector v1 (vr);
cvector v2 (vr, false);
std::cout << v1 << v2;

prints

(1.00e+00,0.00e+00) (2.00e+00,0.00e+00) (3.00e+00,0.00e+00)
(0.00e+00,1.00e+00) (0.00e+00,2.00e+00) (0.00e+00,3.00e+00)
```

### 2.4.11 **real**

Function

```
rvector cvector::real ();
```

creates **rvector** object of size equal to size of calling vector sharing memory with its real part. In other words, the vector returned is *l-value*. See also **cvector**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
cvector vc(3);
vc.set(std::complex<double>(1.,1.));
std::cout << vc << vc.real();
vc.real()(1) = 7.77;
std::cout << vc;
```

prints

```
(1.00e+00,1.00e+00) (1.00e+00,1.00e+00) (1.00e+00,1.00e+00)
1.00e+00 1.00e+00 1.00e+00
(7.77e+00,1.00e+00) (1.00e+00,1.00e+00) (1.00e+00,1.00e+00)
```

### 2.4.12 `imag`

function

```
rvector cvector::imag ();
```

creates `rvector` object of size equal to size of calling vector sharing memory with its imaginary part. In other words, the vector returned is *l-value*. See also `cvector`. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);
```

```
cvector vc(3);  
vc.set(std::complex<double>(1.,1.));  
std::cout << vc << vc.imag();  
vc.imag()(1) = 7.77;  
std::cout << vc;
```

prints

```
(1.00e+00,1.00e+00) (1.00e+00,1.00e+00) (1.00e+00,1.00e+00)  
1.00e+00 1.00e+00 1.00e+00  
(1.00e+00,7.77e+00) (1.00e+00,1.00e+00) (1.00e+00,1.00e+00)
```

### 2.4.13 operator = (const cvector&)

Operator

```
cvector& cvector::operator = (const cvector& v) throw (cvmexception);
```

sets every element of calling vector to be equal to appropriate element of vector v and returns reference to the vector changed. It throws `cvmexception` in case of different vector sizes. See also `cvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    cvector v(3);
    cvector vc(3);
    v(1) = std::complex<double>(1.,2.);

    vc = v;
    std::cout << vc;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

(1.00e+00,2.00e+00) (0.00e+00,0.00e+00) (0.00e+00,0.00e+00)
```

### 2.4.14 assign(const TC\*, int)

Function

```
cvector& cvector::assign (const TC* p, int nIncr = 1);
```

sets every element of calling vector to be equal to every nIncr-th element of array pointed to by parameter p and returns reference to the vector changed. See also [cvector](#). Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
const double a[] = {1., 2., 3., 4., 5., 6., 7.};
cvector v1(3);
cvector v2(2);
```

```
v1.assign((const std::complex<double>*) a);
v2.assign((const std::complex<double>*) a, 2);
std::cout << v1;
std::cout << v2;
```

prints

```
(1.00e+00,2.00e+00) (3.00e+00,4.00e+00) (5.00e+00,6.00e+00)
(1.00e+00,2.00e+00) (5.00e+00,6.00e+00)
```

### 2.4.15 assign(int, const TC\*, int)

Function

```
cvector& cvector::assign (int n, const TC* p, int nIncr = 1);
```

sets every element of calling vector, starting from  $\ell$ -based  $n$ -th one, to be equal to every  $nIncr$ -th element of array pointed to by parameter  $p$  and returns reference to the vector changed. See also [cvector](#). Example:

```
using namespace cvm;
```

```
const double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};  
cvector v1(4);  
cvector v2(4);
```

```
v1.assign(3, (const std::complex<double>*) a);  
v2.assign(2, (const std::complex<double>*) a, 2);  
std::cout << v1;  
std::cout << v2;
```

prints

```
(0,0) (0,0) (1,2) (3,4)  
(0,0) (1,2) (5,6) (9,10)
```



### 2.4.16 assign(int, const TR\*, int, int)

Function

```
cvector& cvector::assign (int n, const TC* p, int nSize, tint nIncr);
```

sets every element of calling vector, starting from *l*-based *n*-th one, up to *nSize* total, to be equal to every *nIncr*-th element of array pointed to by parameter *p* and returns reference to the vector changed. If *n* + *nSize* goes beyond vector boundaries assignment stops at the last element. See also [cvector](#). Example:

```
using namespace cvm;  
const double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10.};  
cvector v1(4);
```

```
v1.assign(2, (const std::complex<double>*) a, 2, 3);  
std::cout << v1;
```

prints

```
(0,0) (1,2) (7,8) (0,0)
```

### 2.4.17 assign (int, const cvector&)

Function

```
cvector& cvector::assign (int n, const cvector& v) throw (cvmexception);
```

sets every element of calling vector's sub-vector beginning with *l*-based index *n* to vector *v* and returns reference to the vector changed. Function throws `cvmexception` if *n* is not positive or `v.size()+n-1` is greater than calling vector's size. See also `cvector`. Example:

```
using namespace cvm;
```

```
cvector v1(5);  
cvector v2(2);  
v1.set(std::complex<double>(1.,1.));  
v2.set(std::complex<double>(2.,2.));  
v1.assign(3, v2);  
std::cout << v1;
```

prints

```
(1,1) (1,1) (2,2) (2,2) (1,1)
```

### 2.4.18 set(TC)

Function

```
cvector& cvector::set (TC x);
```

sets every element of calling vector to be equal to parameter x and returns reference to the vector changed. See also [cvector](#). Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);
```

```
cvector v(3);  
v.set(std::complex<double>(3.,1.));  
std::cout << v;
```

prints

```
(3.00e+00,1.00e+00) (3.00e+00,1.00e+00) (3.00e+00,1.00e+00)
```

### 2.4.19 assign\_real

Function

```
cvector& cvector::assign_real (const rvector& vRe) throw (cvmexception);
```

sets real part of every element of calling vector to be equal to appropriate element of vector vRe and returns reference to the vector changed. Function throws `cvmexception` in case of different sizes of the operands. See also `cvector` and `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

rvector v(3);
cvector vc(3);
v(1) = 1.; v(2) = 2.; v(3) = 3.;

vc.assign_real(v);
std::cout << vc;

prints

(1.00e+00,0.00e+00) (2.00e+00,0.00e+00) (3.00e+00,0.00e+00)
```

### 2.4.20 assign\_imag

Function

```
cvector& cvector::assign_imag (const rvector& vIm) throw (cvmexception);
```

sets imaginary part of every element of calling vector to be equal to appropriate element of vector vIm and returns reference to the vector changed. Function throws **cvmexception** in case of different sizes of the operands. See also **cvector** and **rvector**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

rvector v(3);
cvector vc(3);
v(1) = 1.; v(2) = 2.; v(3) = 3.;

vc.assign_imag(v);
std::cout << vc;

prints

(0.00e+00,1.00e+00) (0.00e+00,2.00e+00) (0.00e+00,3.00e+00)
```

### 2.4.21 `set_real`

Function

```
cvector& cvector::set_real (TR x);
```

sets real part of every element of calling vector to be equal to parameter `x` and returns reference to the vector changed. See also `cvector`. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);
```

```
cvector v(3);  
v.set_real(1.);  
std::cout << v;
```

prints

```
(1.00e+00,0.00e+00) (1.00e+00,0.00e+00) (1.00e+00,0.00e+00)
```

### 2.4.22 set\_imag

Function

```
cvector& cvector::set_imag (TR x);
```

sets imaginary part of every element of calling vector to be equal to parameter x and returns reference to the vector changed. See also [cvector](#). Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);
```

```
cvector v(3);  
v.set_imag(1.);  
std::cout << v;
```

prints

```
(0.00e+00,1.00e+00) (0.00e+00,1.00e+00) (0.00e+00,1.00e+00)
```

### 2.4.23 `resize`

Function

```
cvector& cvector::resize (int nNewSize) throw (cvmexception);
```

changes size of calling vector to be equal to `nNewSize` and returns reference to the vector changed. In case of increasing of its size, the vector is filled up with zeroes. Function throws `cvmexception` in case of negative size passed or memory allocation failure. See also `basic_array::resize` and `cvector`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4.};
    rvector v (a, 3);
    std::cout << v;
    v.resize(2);
    std::cout << v;
    v.resize(4);
    std::cout << v;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(1,2) (3,4) (5,6)
(1,2) (3,4)
(1,2) (3,4) (0,0) (0,0)
```



### 2.4.24 operator ==

Operator

```
bool cvector::operator == (const cvector& v) const;
```

compares calling vector with vector *v* and returns true if they have the same sizes and their appropriate elements differ by not more than the **smallest normalized positive number**. Returns false otherwise. See also **cvector**. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4.};  
cvector v1 ((std::complex<double>*)a, 2);  
cvector v2 (2);
```

```
v2(1) = std::complex<double>(1.,2.);  
v2(2) = std::complex<double>(3.,4.);
```

```
std::cout << (v1 == v2) << std::endl;
```

prints

```
1
```

### 2.4.25 operator !=

Operator

```
bool cvector::operator != (const cvector& v) const;
```

compares calling vector with vector *v* and returns true if they have different sizes or some of their appropriate elements differ by more than the **smallest normalized positive number**. Returns false otherwise. See also **cvector**. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4.};  
cvector v1 ((std::complex<double>*)a, 2);  
cvector v2 (2);
```

```
std::cout << (v1 != v2) << std::endl;
```

prints

1

### 2.4.26 operator <<

Operator

```
cvector& cvector::operator << (const cvector& v) throw (cvmexception);
```

destroys calling vector, creates a new one as a copy of v and returns reference to the vector changed. See also [cvector](#). Operator throws [cvmexception](#) in case of memory allocation failure. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    cvector v(2);
    cvector vc(3);
    v(1) = std::complex<double> (1.,2.);
    v(2) = std::complex<double> (3.,4.);
    std::cout << v << vc << std::endl;

    vc << v;
    std::cout << vc;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(1.00e+000,2.00e+000) (3.00e+000,4.00e+000)
(0.00e+000,0.00e+000) (0.00e+000,0.00e+000) (0.00e+000,0.00e+000)

(1.00e+000,2.00e+000) (3.00e+000,4.00e+000)
```

### 2.4.27 operator +

Operator

```
cvector cvector::operator + (const cvector& v) const  
throw (cvmexception);
```

creates object of type cvector as sum of calling vector and vector v. Operator throws **cvmexception** in case of different sizes of the operands or memory allocation failure. See also **cvector**, **cvector::sum**. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);  
  
try {  
    cvector va(3);  
    cvector vb(3);  
    va.set(std::complex<double>(1.,1.));  
    vb.set(std::complex<double>(2.,2.));  
  
    std::cout << va + vb;  
    std::cout << va + va;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
(3.00e+000,3.00e+000) (3.00e+000,3.00e+000) (3.00e+000,3.00e+000)  
(2.00e+000,2.00e+000) (2.00e+000,2.00e+000) (2.00e+000,2.00e+000)
```

### 2.4.28 operator -

Operator

```
cvector cvector::operator - (const cvector& v) const  
throw (cvmexception);
```

creates object of type cvector as difference of calling vector and vector v. It throws **cvmexception** in case of different sizes of the operands or memory allocation failure. See also **cvector::diff**, **cvector**. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);  
  
try {  
    cvector va(3);  
    cvector vb(3);  
    va.set(std::complex<double> (1.,1.));  
    vb.set(std::complex<double> (2.,2.));  
  
    std::cout << va - vb;  
    std::cout << va - va;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}  
  
prints  
  
(-1.00e+000,-1.00e+000) (-1.00e+000,-1.00e+000) (-1.00e+000,-1.00e+000)  
(0.00e+000,0.00e+000) (0.00e+000,0.00e+000) (0.00e+000,0.00e+000)
```

### 2.4.29 sum

Function

```
cvector& cvector::sum (const cvector& v1, const cvector& v2)
throw (cvmexception);
```

assigns result of addition of vectors v1 and v2 to calling vector and returns reference to the vector changed. It throws `cvmexception` in case of different sizes of the operands. See also `cvector::operator +`, `cvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    cvector va(3);
    cvector vb(3);
    cvector v(3);
    va.set(std::complex<double> (1.,1.));
    vb.set(std::complex<double> (2.,2.));

    std::cout << v.sum(va, vb);
    std::cout << v.sum(v, va);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(3.00e+000,3.00e+000) (3.00e+000,3.00e+000) (3.00e+000,3.00e+000)
(4.00e+000,4.00e+000) (4.00e+000,4.00e+000) (4.00e+000,4.00e+000)
```

### 2.4.30 diff

Function

```
cvector& cvector::diff (const cvector& v1, const cvector& v2)
throw (cvmexception);
```

assigns result of subtraction of vectors v1 and v2 to calling vector and returns reference to the vector changed. It throws `cvmexception` in case of different sizes of the operands. See also `cvector::operator -`, `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    cvector va(3);
    cvector vb(3);
    cvector v(3);
    va.set(std::complex<double> (1.,1.));
    vb.set(std::complex<double> (2.,2.));

    std::cout << v.diff(va, vb);
    std::cout << v.diff(v, va);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(-1.00e+000,-1.00e+000) (-1.00e+000,-1.00e+000) (-1.00e+000,-1.00e+000)
(-2.00e+000,-2.00e+000) (-2.00e+000,-2.00e+000) (-2.00e+000,-2.00e+000)
```

**2.4.31 operator +=**

Operator

```
cvector& cvector::operator += (const cvector& v) throw (cvmexception);
```

adds vector *v* to calling vector and returns reference to the vector changed. It throws **cvmexception** in case of different sizes of the operands. See also **cvector::operator +** , **cvector::sum**, **cvector**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    cvector v1(3);
    cvector v2(3);
    v1.set(std::complex<double> (1.,1.));
    v2.set(std::complex<double> (2.,2.));

    v1 += v2;
    std::cout << v1;

    // well, you can do this too, but temporary object would be created
    v2 += v2;
    std::cout << v2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

(3.00e+000,3.00e+000) (3.00e+000,3.00e+000) (3.00e+000,3.00e+000)
(4.00e+000,4.00e+000) (4.00e+000,4.00e+000) (4.00e+000,4.00e+000)
```



**2.4.32 operator -=**

Operator

```
cvector& cvector::operator -= (const cvector& v) throw (cvmexception);
```

subtracts vector v from calling vector and returns reference to the vector changed. It throws **cvmexception** in case of different sizes of the operands. See also **cvector::operator -** , **cvector::diff**, **cvector**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    cvector v1(3);
    cvector v2(3);
    v1.set(std::complex<double> (1.,1.));
    v2.set(std::complex<double> (2.,2.));

    v1 -= v2;
    std::cout << v1;

    // well, you can do this too, but temporary object would be created
    v2 -= v2;
    std::cout << v2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

(-1.00e+000,-1.00e+000) (-1.00e+000,-1.00e+000) (-1.00e+000,-1.00e+000)
(0.00e+000,0.00e+000) (0.00e+000,0.00e+000) (0.00e+000,0.00e+000)
```

### 2.4.33 operator - ()

Operator

```
cvector cvector::operator - () const throw (cvmexception);
```

creates object of type `cvector` as calling vector multiplied by  $-1$ . It throws `cvmexception` in case of memory allocation failure. See also `cvector`. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);
```

```
double a[] = {1., 2., 3., 4.};  
const cvector v ((std::complex<double>*) a, 2);
```

```
std::cout << - v;
```

prints

```
(-1.00e+000,-2.00e+000) (-3.00e+000,-4.00e+000)
```

**2.4.34 operator \* (TR)**

Operator

```
cvector cvector::operator * (TR d) const;
```

creates object of type `cvector` as product of calling vector and number `d`. See also `cvector`.

Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);
```

```
double a[] = {1., 2., 3., 4.};  
const cvector v ((std::complex<double>*) a, 2);
```

```
std::cout << v * 5.;
```

prints

```
(5.00e+000,1.00e+001) (1.50e+001,2.00e+001)
```

### 2.4.35 operator / (TR)

Operator

```
cvector cvector::operator / (TR d) const throw (cvmexception);
```

creates object of type `cvector` as quotient of calling vector and number `d`. Operator throws `cvmexception` if `d` has absolute value equal or less than the **smallest normalized positive number**. See also `cvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
double a[] = {1., 2., 3., 4.};
const cvector v ((std::complex<double>*) a, 2);
```

```
std::cout << v / 4.;
```

prints

```
(2.50e-001,5.00e-001) (7.50e-001,1.00e+000)
```

### 2.4.36 operator \* (TC)

Operator

```
cvector cvector::operator * (TC c) const;
```

creates object of type cvector as product of calling vector and complex number c. See also [cvector](#). Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);
```

```
double a[] = {1., 2., 3., 4.};  
const cvector v ((std::complex<double>*) a, 2);
```

```
std::cout << v * std::complex<double>(1.,1.);
```

prints

```
(-1.00e+000,3.00e+000) (-1.00e+000,7.00e+000)
```

### 2.4.37 operator / (TC)

Operator

```
cvector cvector::operator / (TC c) const  
throw (cvmexception);
```

creates object of type `cvector` as quotient of calling vector and complex number `c`. Operator throws `cvmexception` if `c` has absolute value equal or less than the **smallest normalized positive number**. See also `cvector`. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);  
  
double a[] = {1., 2., 3., 4.};  
const cvector v ((std::complex<double>*) a, 2);  
  
std::cout << v / std::complex<double>(1.,1.);  
  
prints  
  
(1.50e+000,5.00e-001) (3.50e+000,5.00e-001)
```

### 2.4.38 operator \*= (TR)

Operator

```
cvector& cvector::operator *= (TR d);
```

multiplies calling vector by real number d and returns reference to the vector changed. See also `cvector::operator *`, `cvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
double a[] = {1., 2., 3., 4.};
cvector v ((std::complex<double>*) a, 2);
```

```
std::cout << (v *= 2.);
```

prints

```
(2.00e+000,4.00e+000) (6.00e+000,8.00e+000)
```

**2.4.39 operator /= (TR)**

Operator

```
cvector& cvector::operator /= (TR d) throw (cvmexception);
```

divides calling vector by real number *d* and returns reference to the vector changed. It throws **cvmexception** if *d* has absolute value equal or less than the **smallest normalized positive number**. See also **cvector::operator /** , **cvector**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
double a[] = {1., 2., 3., 4.};
cvector v ((std::complex<double>*) a, 2);
```

```
std::cout << (v /= 2.);
```

prints

```
(5.00e-001,1.00e+000) (1.50e+000,2.00e+000)
```



### 2.4.40 operator \*= (TC)

Operator

```
cvector& cvector::operator *= (TC c);
```

multiplies calling vector by complex number *c* and returns reference to the vector changed.

See also `cvector::operator *`, `cvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
double a[] = {1., 2., 3., 4.};
cvector v ((std::complex<double>*) a, 2);
```

```
v *= std::complex<double>(1.,1.);
std::cout << v;
```

prints

```
(-1.00e+000,3.00e+000) (-1.00e+000,7.00e+000)
```

### 2.4.41 operator /= (TC)

Operator

```
cvector& cvector::operator /= (TC c) throw (cvmexception);
```

divides calling vector by complex number `c` and returns reference to the vector changed. It throws `cvmexception` if `c` has absolute value equal or less than the **smallest normalized positive number**. See also `cvector::operator /`, `cvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
double a[] = {1., 2., 3., 4.};
cvector v ((std::complex<double>*) a, 2);
```

```
v /= std::complex<double>(1.,1.);
std::cout << v;
```

prints

```
(1.50e+000,5.00e-001) (3.50e+000,5.00e-001)
```

### 2.4.42 normalize

Function

```
cvector& cvector::normalize ();
```

normalizes calling vector so its **Euclidean norm** becomes equal to 1 if it was greater than the **smallest normalized positive number** before the call (otherwise it does nothing). See also **cvector**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
double a[] = {1., 2., 3., 4.};
cvector v ((std::complex<double>*) a, 2);
```

```
std::cout << v.normalize();
std::cout << v.norm() << std::endl;
```

prints

```
(1.83e-01,3.65e-01) (5.48e-01,7.30e-01)
1.00e+00
```

### 2.4.43 conjugation

Operator and functions

```
cvector  cvector::operator ~ () const throw (cvmexception);  
cvector& cvector::conj (const cvector& v) throw (cvmexception);  
cvector& cvector::conj ();
```

implement complex conjugation. First operator creates object of type `cvector` as complex conjugated calling vector (it throws `cvmexception` in case of memory allocation failure). Second function sets calling vector to be equal to vector `v` conjugated (it throws `cvmexception` in case of different sizes of the operands), third one makes it to be equal to conjugated itself. See also `cvector`. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);  
double a[] = {1., 2., 3., 4.};  
const cvector v ((std::complex<double>*) a, 2);  
cvector vc(2);  
  
std::cout << ~v;  
std::cout << vc.conj(v);  
std::cout << vc.conj();
```

prints

```
(1.00e+00,-2.00e+00) (3.00e+00,-4.00e+00)  
(1.00e+00,-2.00e+00) (3.00e+00,-4.00e+00)  
(1.00e+00,2.00e+00) (3.00e+00,4.00e+00)
```

#### 2.4.44 operator \* (const cvector&)

Operator

```
TC cvector::operator * (const cvector& v) const throw (cvmexception);
```

returns scalar product of calling vector and vector v. Operator throws `cvmexception` if the operands have different sizes. See also `cvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

double a[] = {1., 2., 3., 4.};
double b[] = {1., -1., 1., 2.};
const cvector v1((std::complex<double>*) a, 2);
const cvector v2((std::complex<double>*) b, 2);

std::cout << v1 * v2 << std::endl;

prints

(-2.00e+00,1.10e+01)
```

### 2.4.45 operator %

Operator

TC `cvector::operator % (const cvector& v) const throw (cvmexception);`

returns scalar product of complex conjugated calling vector and vector `v`. Operator throws `cvmexception` if the operands have different sizes. See also `cvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
double a[] = {1., 2., 3., 4.};
double b[] = {1., -1., 1., 2.};
const cvector v1((std::complex<double>*) a, 2);
const cvector v2((std::complex<double>*) b, 2);
```

```
std::cout << v1 % v2 << std::endl;
std::cout << ~v1 * v2 << std::endl;
```

prints

```
(1.00e+01,-1.00e+00)
(1.00e+01,-1.00e+00)
```

**2.4.46 operator \* (const cmatrix&)**

Operator

```
cvector cvector::operator * (const cmatrix& m) const
throw (cvmexception);
```

creates object of type cvector as product of calling vector and matrix m. Operator throws **cvmexception** if the calling vector's size is differ from number of rows of matrix m'. See also **cvector::mult (const cvector&, const cmatrix&), cvector, cmatrix**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

double a[] = {1., 2., 3., 3., 2., 1.};
double b[] = {1., -1., 1., 2., -2., 1.,
              3., -2., 1., 2., -1., 3.};
const cvector v((std::complex<double>*) a, 3);
const cmatrix m((std::complex<double>*) b, 3, 2);

std::cout << v << m << std::endl << v * m;

prints

(1.00e+00,2.00e+00) (3.00e+00,3.00e+00) (2.00e+00,1.00e+00)
(1.00e+00,-1.00e+00) (3.00e+00,-2.00e+00)
(1.00e+00,2.00e+00) (1.00e+00,2.00e+00)
(-2.00e+00,1.00e+00) (-1.00e+00,3.00e+00)

(-5.00e+00,1.00e+01) (-1.00e+00,1.80e+01)
```

**2.4.47 mult (const cvector&, const cmatrix&)**

Function

```
cvector& cvector::mult (const cvector& v, const cmatrix& m)
throw (cvmexception);
```

sets calling vector to be equal to product of vector v and matrix m and returns reference to the object changed. Function throws `cvmexception` if case of inappropriate sizes of the operands. See also `cvector::mult (const cmatrix&, const cvector&)`, `cvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    double a[] = {1., 2., 3., 1., 2., 3.};
    double b[] = {1., -1., 1., -1., 1., -1.,
                  2., -1., 2., -1., 2., -1.};
    const cvector v ((std::complex<double>*) a, 3);
    const cmatrix m ((std::complex<double>*) b, 3, 2);
    const scmatrix sm ((std::complex<double>*) b, 2);
    cvector vm (2);

    std::cout << vm.mult(v, m) << std::endl;
    std::cout << sm << std::endl;
    std::cout << vm.mult(vm, sm);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(1.20e+01,0.00e+00) (1.80e+01,6.00e+00)

(1.00e+00,-1.00e+00) (1.00e+00,-1.00e+00)
(1.00e+00,-1.00e+00) (2.00e+00,-1.00e+00)

(3.60e+01,-2.40e+01) (5.40e+01,-1.80e+01)
```



**2.4.48 mult (const cmatrix&, const cvector&)**

Function

```
cvector& cvector::mult (const cmatrix& m, const cvector& v)
throw (cvmexception);
```

sets calling vector to be equal to product of matrix m and vector v and returns reference to the object changed. Function throws `cvmexception` if case of inappropriate sizes of the operands. See also `cvector::mult (const cvector&, const cmatrix&)`, `cvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    double a[] = {1., 2., 3., 1., 2., 3.};
    double b[] = {1., -1., 1., -1., 1., -1.,
                  2., -1., 2., -1., 2., -1.};
    const cvector v ((std::complex<double>*) a, 3);
    const cmatrix m ((std::complex<double>*) b, 2, 3);
    const scmatrix sm ((std::complex<double>*) b, 2);
    cvector vm (2);

    std::cout << vm.mult(m, v) << std::endl;
    std::cout << sm << std::endl;
    std::cout << vm.mult(vm, sm);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(1.40e+01,3.00e+00) (1.70e+01,4.00e+00)

(1.00e+00,-1.00e+00) (1.00e+00,-1.00e+00)
(1.00e+00,-1.00e+00) (2.00e+00,-1.00e+00)

(3.80e+01,-2.40e+01) (5.50e+01,-2.00e+01)
```

### 2.4.49 ranklupdate\_u

Function

```
cmatrix cvector::ranklupdate_u (const cvector& v) const;
```

creates object of type **cmatrix** as rank-1 update (unconjugated) of calling vector and vector  $v$ . The rank-1 update (unconjugated) of vector-column  $x$  of size  $m$  and vector-row  $y$  of size  $n$  is defined as  $m \times n$  matrix

$$\begin{pmatrix} x_1y_1 & x_1y_2 & \cdots & x_1y_n \\ x_2y_1 & x_2y_2 & \cdots & x_2y_n \\ \dots\dots\dots \\ x_my_1 & x_my_2 & \cdots & x_my_n \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} (y_1 \ y_2 \ \cdots \ y_n)$$

See also **cvector**, **cmatrix**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
double a[] = {1., 2., 3., -2., -1., 1.};
double b[] = {4., 5., 3., 2.};
cvector v1((std::complex<double>*) a, 3);
cvector v2((std::complex<double>*) b, 2);
```

```
std::cout << v1.ranklupdate_u (v2);
```

prints

```
(-6.00e+00,1.30e+01) (-1.00e+00,8.00e+00)
(2.20e+01,7.00e+00) (1.30e+01,0.00e+00)
(-9.00e+00,-1.00e+00) (-5.00e+00,1.00e+00)
```

**2.4.50 ranklupdate\_c**

Function

```
cmatrix cvector::ranklupdate_c (const cvector& v) const;
```

creates object of type `cmatrix` as rank-1 update (conjugated) of calling vector and complex conjugated vector `v`. The rank-1 update (conjugated) operation of vector-column `x` of size `m` and complex conjugated vector-column `y` of size `n` is defined as `m × n` matrix

$$\begin{pmatrix} x_1 y_1^* & x_1 y_2^* & \cdots & x_1 y_n^* \\ x_2 y_1^* & x_2 y_2^* & \cdots & x_2 y_n^* \\ \cdots & \cdots & \cdots & \cdots \\ x_m y_1^* & x_m y_2^* & \cdots & x_m y_n^* \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} \text{conj} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} (y_1^* \ y_2^* \ \cdots \ y_n^*),$$

where  $y_i^*$  is *i*-th complex conjugated element of `y`. See also `cvector`, `cmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
double a[] = {1., 2., 3., -2., -1., 1.};
double b[] = {4., 5., 3., 2.};
cvector v1((std::complex<double>*) a, 3);
cvector v2((std::complex<double>*) b, 2);

std::cout << v1.ranklupdate_c (v2) << std::endl;
std::cout << v1.ranklupdate_u (~v2);
```

prints

```
(1.40e+01,3.00e+00) (7.00e+00,4.00e+00)
(2.00e+00,-2.30e+01) (5.00e+00,-1.20e+01)
(1.00e+00,9.00e+00) (-1.00e+00,5.00e+00)

(1.40e+01,3.00e+00) (7.00e+00,4.00e+00)
(2.00e+00,-2.30e+01) (5.00e+00,-1.20e+01)
(1.00e+00,9.00e+00) (-1.00e+00,5.00e+00)
```

**2.4.51 solve**

Functions

```

cvector&
cvector::solve (const scmatrix& mA,
                const cvector& vB, TR& dErr) throw (cvmexception);

cvector&
cvector::solve (const scmatrix& mA,
                const cvector& vB) throw (cvmexception);

```

set calling vector to be equal to solution  $x$  of linear equation  $A * x = b$  where parameter `mA` is square complex matrix  $A$  and parameter `vB` is complex vector  $b$ . Every function returns reference to the vector changed. The first version also sets output parameter `dErr` to be equal to norm of computation error. These functions throw `cvmexception` in case of inappropriate sizes of the objects or when the matrix  $A$  is close to singular. See also `operator % (scmatrix), cvector, scmatrix`. Example:

```

using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (7);

try {
    double m[] = {1., -1., 1., 2., -2., 1., 3., -3.};
    double b[] = {1., 2., 5., -3.};
    scmatrix ma((std::complex<double>*) m, 2);
    cvector vb((std::complex<double>*) b, 2);
    cvector vx(2);
    double dErr = 0.;

    std::cout << vx.solve (ma, vb, dErr);
    std::cout << dErr << std::endl;
    std::cout << ma * vx - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

```

prints

```

(3.5200000e+00,6.4000000e-01) (2.2400000e+00,-1.3200000e+00)
3.2788531e-15
(-7.7715612e-16,4.4408921e-16) (0.0000000e+00,0.0000000e+00)

```

### 2.4.52 solve\_tran

Functions

```
cvector& cvector::solve_tran (const scmatrix& mA,
                             const cvector& vB, TR& dErr)
                             throw (cvmexception);
cvector& cvector::solve_tran (const scmatrix& mA,
                             const cvector& vB)
                             throw (cvmexception);
```

set calling vector to be equal to solution  $x$  of linear equation  $A^T * b = b$  (which is equivalent to  $x * A = b$ ) where parameter `mA` is square complex matrix  $A$  and parameter `vB` is complex vector  $b$ . Every function returns reference to the vector changed. The first version also sets output parameter `dErr` to be equal to the norm of computation error. These functions throw `cvmexception` in case of inappropriate sizes of the objects or when the matrix  $A$  is close to singular. See also `operator / (scmatrix)`, `cvector::solve`, `cvector::solve_conj`, `cvector`, `scmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (7);

try {
    double m[] = {1., -1., 1., 2., -2., 1., 3., -3.};
    double b[] = {1., 2., 5., -3.};
    scmatrix ma((std::complex<double>*) m, 2);
    cvector vb((std::complex<double>*) b, 2);
    cvector vx(2);
    double dErr = 0.;

    std::cout << vx.solve_tran (ma, vb, dErr);
    std::cout << dErr << std::endl;
    std::cout << vx * ma - vb;
    std::cout << !ma * vx - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
(+1.60000000e-001,-8.80000000e-001) (+1.56000000e+000,-8.00000000e-002)
+3.7480513e-015
(+0.00000000e+000,+0.00000000e+000) (+0.00000000e+000,+4.4408921e-016)
(+0.00000000e+000,+0.00000000e+000) (+0.00000000e+000,+4.4408921e-016)
```

### 2.4.53 solve\_conj

Functions

```
cvector& cvector::solve_conj (const scmatrix& mA,
                             const cvector& vB, TR& dErr)
                             throw (cvmexception);
cvector& cvector::solve_conj (const scmatrix& mA,
                             const cvector& vB)
                             throw (cvmexception);
```

set calling vector to be equal to solution  $x$  of linear equation  $A^H * x = b$  (here  $A^H$  is conjugated  $A$ ) where parameter  $mA$  is square complex matrix  $A$  and parameter  $vB$  is complex vector  $b$ . Every function returns reference to the vector changed. The first version also sets output parameter  $dErr$  to be equal to the norm of computation error. These functions throw `cvmexception` in case of inappropriate sizes of the objects or when the matrix  $A$  is close to singular. See also `cvector::solve`, `cvector::solve_tran`, `cvector`, `scmatrix`, `scmatrix::operator ~`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (7);

try {
    double m[] = {1., -1., 1., 2., -2., 1., 3., -3.};
    double b[] = {1., 2., 5., -3.};
    scmatrix ma((std::complex<double>*) m, 2);
    cvector vb((std::complex<double>*) b, 2);
    cvector vx(2);
    double dErr = 0.;

    std::cout << vx.solve_conj (ma, vb, dErr);
    std::cout << dErr << std::endl;
    std::cout << ~ma * vx - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

(+2.3200000e+000,+3.7600000e+000) (+2.1200000e+000,+1.6000000e-001)
+2.1932508e-015
(+0.0000000e+000,-4.4408921e-016) (-8.8817842e-016,-8.8817842e-016)
```

**2.4.54 operator / (const scmatrix&)**

Operator

```
cvector operator / (const scmatrix& mA) const throw (cvmexception);
```

returns solution  $x$  of linear equation  $A^T * x = b$  (which is equivalent to  $x * A = b$ ) where parameter `mA` is square complex matrix  $A$  and calling vector is  $b$ . This operator throws exception of type `cvmexception` in case of inappropriate sizes of the objects or when the matrix  $A$  is close to singular. See also `cvector::solve_tran`, `cvector`, `scmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (7);

try {
    double m[] = {1., -1., 1., 2., -2., 1., 3., -3.};
    double b[] = {1., 2., 5., -3.};
    scmatrix ma((std::complex<double>*) m, 2);
    cvector vb((std::complex<double>*) b, 2);
    cvector vx(2);

    vx = vb / ma;

    std::cout << vx * ma - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

(+0.00000000e+000,+0.00000000e+000) (+0.00000000e+000,+4.4408921e-016)
```

**2.4.55 operator % (const scmatrix&)**

Operator

```
cvector operator % (const scmatrix& mA) const throw (cvmexception);
```

returns solution  $x$  of linear equation  $A * x = b$  where parameter `mA` is square complex matrix  $A$  and calling vector is  $b$ . This operator throws exception of type `cvmexception` in case of inappropriate sizes of the objects or when the matrix  $A$  is close to singular. See also `cvector::solve`, `cvector`, `scmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (7);

try {
    double m[] = {1., -1., 1., 2., -2., 1., 3., -3.};
    double b[] = {1., 2., 5., -3.};
    scmatrix ma((std::complex<double>*) m, 2);
    cvector vb((std::complex<double>*) b, 2);
    cvector vx(2);

    vx = vb % ma;

    std::cout << ma * vx - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
(-6.6613381e-016,+4.4408921e-016) (+0.0000000e+000,+0.0000000e+000)
```



### 2.4.56 solve\_lu

Functions

```
cvector&
cvector::solve_lu (const scmatrix& mA, const scmatrix& mLU,
                  const int* pPivots, const cvector& vB, TR& dErr)
                  throw (cvmexception);
```

```
cvector&
cvector::solve_lu (const scmatrix& mA, const scmatrix& mLU,
                  const int* pPivots, const cvector& vB)
                  throw (cvmexception);
```

set calling vector to be equal to solution  $x$  of linear equation  $A * x = b$  where parameter mA is square matrix A, parameter mLU is [LU factorization](#) of the matrix A, parameter pPivots is array of pivot numbers created while factorizing the matrix A and parameter vB is vector b. Every function returns reference to the vector changed. The first version also sets output parameter dErr to be equal to norm of computation error. These functions are useful when you need to solve few linear equations of kind  $Ax = b$  with the same matrix A and different vectors b. In such case you save on matrix A factorization since it's needed to be performed just one time. These functions throw [cvmexception](#) in case of inappropriate sizes of the objects or when the matrix A is close to singular. See also [cvector](#), [scmatrix](#). Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (7);

try {
    double m[] = {1., -1., 1., 2., -2., 1., 3., -3.};
    double b1[] = {1., 2., 5., -3.};
    double b2[] = {3., -1., 1., 7.};
    scmatrix ma((std::complex<double>*) m, 2);
    scmatrix mLU(2);
    cvector vb1((std::complex<double>*) b1, 2);
    cvector vb2((std::complex<double>*) b2, 2);
    cvector vx1(2);
    cvector vx2(2);
    iarray nPivots(2);
    double dErr = 0.;

    mLU.low_up(ma, nPivots);
    std::cout << vx1.solve_lu (ma, mLU, nPivots, vb1, dErr);
    std::cout << dErr << std::endl;
```

```
    std::cout << vx2.solve_lu (ma, mLU, nPivots, vb2);
    std::cout << ma * vx1 - vb1 << ma * vx2 - vb2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

(3.5200000e+00,6.4000000e-01) (2.2400000e+00,-1.3200000e+00)
3.2788531e-15
(2.2800000e+00,1.9600000e+00) (3.6000000e-01,5.2000000e-01)
(-7.7715612e-16,4.4408921e-16) (0.0000000e+00,0.0000000e+00)
(-8.8817842e-16,0.0000000e+00) (-2.2204460e-16,0.0000000e+00)
```

**2.4.57 gels**

Function

```
cvector& cvector::gels (bool conjugate,
                      const cmatrix& mA, const cvector& mB,
                      TC& cErr) throw (cvmexception);
```

This function solves overdetermined or underdetermined linear system

$$A * x = b$$

for  $m \times n$  matrix  $A$  (or its conjugate) where  $b$  is a vector of length  $k$  and  $k = m$  in non-transposed case and  $k = n$  otherwise. The algorithm uses QR or LQ factorization of  $A$ . It is assumed that  $A$  has full rank, infinity returned otherwise. Internally function uses ?GELS LAPACK routines. If  $m > n$  and `conjugate=false` or  $m < n$  and `conjugate=true`, then the system is overdetermined, thus the algorithm tries to find the least squares solution  $x$  of the problem

$$\|A * x - b\|_2 \rightarrow \min \quad \text{or} \quad \|A^H * x - b\|_2 \rightarrow \min$$

respectively. Complex number `cErr` returns residuals. The system is underdetermined otherwise, and the algorithm finds its minimum norm solution. In this case `cErr` is set to zero. In both cases the solution computed satisfies  $x = \text{pinv}(A) * b$ , but this algorithm is faster than pseudo inversion. Function sets calling object to be the solution and returns reference to it. It throws `cvmexception` in case of inappropriate sizes of the operands. See also `cvector`, `cmatrix::gels`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (7);
cvm::cmatrix a(7, 5);
cvm::cvector b(7), bt(5);
tcomplex cErr, cErrc;
a.randomize_real(-10., 10.);
a.randomize_imag(-10., 10.);
b.randomize_real(-10., 10.);
b.randomize_imag(-10., 10.);
bt.randomize_real(-10., 10.);
bt.randomize_imag(-10., 10.);

cvm::cvector x(5);
x.gels(false, a, b, cErr);
cvm::cvector xt(7);
xt.gels(true, a, bt, cErrc);
```

```
std::cout << (a.pinv()*b - x).norm() << std::endl;
std::cout << cErr << std::endl;
std::cout << (~a.pinv()*bt - xt).norm() << std::endl;
std::cout << cErrc << std::endl;
```

prints

```
+1.0444134e-015
(-3.2469414e+001,+5.8375852e+000)
+1.9786436e-015
(+0.0000000e+000,+0.0000000e+000)
```

### 2.4.58 gelsy

Function

```
cvector& cvector::gelsy (const cmatrix& mA, const cvector& mB, int& rank,
                       TR tol = cvmMachSp()) throw (cvmexception);
```

These functions compute minimum-norm solution to linear least squares problem

$$\|A * x - b\|_2 \rightarrow \min$$

using complete orthogonal factorization of  $m \times n$  matrix  $A$ . Here  $b$  is a vector of length  $m$ . Matrix  $A$  may be rank-deficient, function returns its effective rank in rank output parameter using tol tolerance. Internally function uses ?GELSY LAPACK routines, see more details about the algorithm in those routines' documentation. Matrix  $A$  is passed as argument mA. Function sets calling object to be the solution and returns reference to it. It throws `cvmexception` in case of inappropriate sizes of the operands. See also `cvector`, `cvector::gelss`, `cvector::gelsd`, `cmatrix::gelsy`. Example:

```
using namespace cvm;
```

```
cmatrix a(4, 5);
cvector b(4);
a.randomize_real(-10., 10.);
a.randomize_imag(-10., 10.);
b.randomize_real(-10., 10.);
b.randomize_imag(-10., 10.);
cvector x(5);
tint rank;

x.gelsy(a, b, rank);
std::cout << (a*x - b).norm() << std::endl;
std::cout << rank << " " << a.rank() << std::endl;
```

prints

```
+6.0116296e-015
+4 +4
```

**2.4.59 gelss**

Function

```
cvector& cvector::gelss (const cmatrix& mA, const cvector& mB, int& rank,
                        rvector& sv, TR tol = cvmMachSp())
                        throw (cvmexception);
```

computes minimum-norm solution to linear least squares problem

$$\|A * x - b\|_2 \rightarrow \min$$

using the singular value decomposition of  $m \times n$  matrix  $A$ . Here  $b$  is a vector of length  $m$ . Matrix  $A$  may be rank-deficient, function returns its effective rank in `rank` output parameter using `tol` tolerance. This function also computes singular values of  $A$  in decreasing order and returns them in `sv` output parameter having  $\min(m, n)$  size. Internally function uses ?GELSS LAPACK routines, see more details about the algorithm in that routine's documentation. Matrix  $A$  is passed as argument `mA`. Function sets calling object to be the solution and returns reference to it. It throws `cvmexception` in case of inappropriate sizes of the operands. See also `cvector`, `cvector::gelsy`, `cvector::gelsd`, `cmatrix::gelss`. Example:

```
using namespace cvm;

cvm::cmatrix a(4, 5);
cvm::cvector b(4);
a.randomize_real(-10., 10.);
a.randomize_imag(-10., 10.);
b.randomize_real(-10., 10.);
b.randomize_imag(-10., 10.);
cvm::rvector sv(4);
cvm::cvector x(5);
tint rank;

x.gelss(a, b, sv, rank);
std::cout << (a*x - b).norm() << std::endl;
std::cout << (sv - a.svd()).norm() << std::endl;
std::cout << rank << " " << a.rank() << std::endl;
```

prints

```
+9.4127010e-015
+4.4408921e-015
+4 +4
```

**2.4.60 gelsd**

Function

```
cvector& cvector::gelsd (const cmatrix& mA, const cvector& mB, int& rank,
                        rvector& sv, TR tol = cvmMachSp())
                        throw (cvmexception);
```

computes minimum-norm solution to linear least squares problem

$$\|A * x - b\|_2 \rightarrow \min$$

using the singular value decomposition of  $m \times n$  matrix  $A$  and divide and conquer method. Here  $b$  is a vector of length  $m$ . Matrix  $A$  may be rank-deficient, function returns its effective rank in `rank` output parameter using `tol` tolerance. This function also computes singular values of  $A$  in decreasing order and returns them in `sv` output parameter having  $\min(m, n)$  size. Internally function uses ?GELSD LAPACK routines, see more details about the algorithm in that routine's documentation. Matrix  $A$  is passed as argument `mA`. Function sets calling object to be the solution and returns reference to it. It throws `cvmexception` in case of inappropriate sizes of the operands. See also `cvector`, `cvector::gelsy`, `cvector::gelss`, `cmatrix::gelsd`. Example:

```
using namespace cvm;
```

```
cvm::cmatrix a(4, 5);
cvm::cvector b(4);
a.randomize_real(-10., 10.);
a.randomize_imag(-10., 10.);
b.randomize_real(-10., 10.);
b.randomize_imag(-10., 10.);
cvm::rvector sv(4);
cvm::cvector x(5);
tint rank;

x.gelsd(a, b, sv, rank);
std::cout << (a*x - b).norm() << std::endl;
std::cout << (sv - a.svd()).norm() << std::endl;
std::cout << rank << " " << a.rank() << std::endl;
```

```
prints
```

```
+1.3491837e-014
+5.3290705e-015
+4 +4
```

**2.4.61 eig**

Functions

```

cvector&
cvector::eig (const srmatrix& mArg) throw (cvmexception);
cvector&
cvector::eig (const scmatrix& mArg) throw (cvmexception);

cvector&
cvector::eig (const srmatrix& mArg, scmatrix& mEigVect,
              bool bRightVect = true) throw (cvmexception);
cvector&
cvector::eig (const scmatrix& mArg, scmatrix& mEigVect,
              bool bRightVect = true) throw (cvmexception);

```

solve nonsymmetric eigenvalue problem and set calling vector to be equal to eigenvalues of a square matrix `mArg`. The nonsymmetric eigenvalue problem is defined as follows: given a nonsymmetric (or non-Hermitian) matrix  $A$ , find the eigenvalues  $\lambda$  and the corresponding eigenvectors  $z$  that satisfy the equation

$$Az = \lambda z.$$

Some eigenvalues may be complex even for real matrix  $A$ . Moreover, if real nonsymmetric matrix has a complex eigenvalue  $a + bi$  corresponding to an eigenvector  $z$ , then  $a - bi$  is also an eigenvalue. The eigenvalue  $a - bi$  corresponds to the eigenvector whose elements are complex conjugate to the elements of  $z$ . The third and fourth versions of functions set output parameter `mEigVect` to be equal to square matrix containing eigenvectors as columns. They also compute "left" eigenvectors if parameter `bRightVect` is set to `false`. Left eigenvectors satisfy

$$z^H A = \lambda z^H.$$

All functions return reference to the vector they change and throw `cvmexception` in case of inappropriate calling object sizes or in case of convergence error. See also `cvector`, `rvector::eig`, `scmatrix`. Example:

```

using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (2);

try {
    srmatrix m(3);
    scmatrix me(3);
    cvector vl(3);

```



```

m(1,1) = 0.1; m(1,2) = 0.2; m(1,3) = 0.1;
m(2,1) = 0.11; m(2,2) = 2.9; m(2,3) = -8.4;
m(3,1) = 0.; m(3,2) = 2.91; m(3,3) = 8.2;
scmatrix mc(m);

std::cout << vl.eig (m, me) << std::endl;
std::cout << mc * me(1) - me(1) * vl(1);
std::cout << mc * me(2) - me(2) * vl(2);
std::cout << mc * me(3) - me(3) * vl(3) << std::endl;

std::cout << vl.eig (m, me, false) << std::endl;
std::cout << ~(me(1)) * mc - ~(me(1)) * vl(1);
std::cout << ~(me(2)) * mc - ~(me(2)) * vl(2);
std::cout << ~(me(3)) * mc - ~(me(3)) * vl(3);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
prints
(+9.69e-002,+0.00e+000) (+5.55e+000,+4.17e+000) (+5.55e+000,-4.17e+000)

(+2.78e-017,+0.00e+000) (+7.24e-017,+0.00e+000) (+4.23e-017,+0.00e+000)
(-2.22e-016,-1.94e-016) (-7.11e-015,-4.88e-015) (+0.00e+000,+1.78e-015)
(-2.22e-016,+1.94e-016) (-7.11e-015,+4.88e-015) (+0.00e+000,-1.78e-015)

(+9.69e-002,+0.00e+000) (+5.55e+000,+4.17e+000) (+5.55e+000,-4.17e+000)

(+0.00e+000,+0.00e+000) (+4.19e-017,+0.00e+000) (+1.13e-017,+0.00e+000)
(+0.00e+000,-4.16e-017) (+2.22e-016,-1.78e-015) (-8.88e-016,+2.44e-015)
(+0.00e+000,+4.16e-017) (+2.22e-016,+1.78e-015) (-8.88e-016,-2.44e-015)

```

**2.4.62 gemv**

Function

```
cvector& cvector::gemv (bool bLeft, const cmatrix& m, TC dAlpha,
                      const cvector& v, TC dBeta) throw (cvmexception);
```

calls one of ?GEMV routines of the [BLAS library](#) performing matrix-vector operation defined as

$$c = \alpha M \cdot v + \beta c \quad \text{or} \quad c = \alpha v \cdot M + \beta c,$$

where  $\alpha$  and  $\beta$  are complex numbers (parameters dAlpha and dBeta), M is complex matrix (parameter m) and v and c are complex vectors (parameter v and calling vector respectively). First operation is performed if bLeft passed is false and second one otherwise. Function returns reference to the vector changed and throws [cvmexception](#) in case of inappropriate calling object sizes. See also [cvector](#), [cmatrix](#). Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    std::complex<double> alpha = std::complex<double>(1.3,-0.7);
    std::complex<double> beta  = std::complex<double>(0.15,-1.09);
    cmatrix m(3,2);
    cvector c(3);
    cvector v(2);
    m.randomize_real(-1., 2.); m.randomize_imag(0., 1.);
    v.randomize_real(-1., 3.); v.randomize_imag(2., 4.);
    c.randomize_real(0., 2.); c.randomize_imag(3., 7.);
    std::cout << m * v * alpha + c * beta;
    std::cout << c.gemv(false, m, alpha, v, beta);
    std::cout << c * m * alpha + v * beta;
    std::cout << v.gemv(true, m, alpha, c, beta);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

(2.71e-01,2.44e+00) (2.20e+01,7.16e+00) (-7.89e-01,2.45e+00)
(2.71e-01,2.44e+00) (2.20e+01,7.16e+00) (-7.89e-01,2.45e+00)
(5.92e+01,-1.47e+01) (3.54e+01,-3.14e+00)
(5.92e+01,-1.47e+01) (3.54e+01,-3.14e+00)
```

**2.4.63 gbmv**

Function

```
cvector& cvector::gbmv (bool bLeft, const scbmatrix& m, TC dAlpha,
                       const cvector& v, TC dBeta) throw (cvmexception);
```

calls one of ?GBMV routines of the [BLAS library](#) performing matrix-vector operation defined as

$$c = \alpha M \cdot v + \beta c \quad \text{or} \quad c = \alpha v \cdot M + \beta c,$$

where  $\alpha$  and  $\beta$  are complex numbers (parameters dAlpha and dBeta), M is complex band matrix (parameter m) and v and c are complex vectors (parameter v and calling vector respectively). First operation is performed if bLeft passed is false and second one otherwise. Function returns reference to the vector changed and throws [cvmexception](#) in case of inappropriate calling object sizes. See also [cvector](#), [scbmatrix](#). Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    std::complex<double> alpha = std::complex<double>(1.3,-0.7);
    std::complex<double> beta  = std::complex<double>(0.15,-1.09);
    scbmatrix m(3,1,0);
    cvector c(3);
    cvector v(3);
    m.randomize_real(-1., 2.); m.randomize_imag(0., 1.);
    v.randomize_real(-1., 3.); v.randomize_imag(2., 4.);
    c.randomize_real(0., 2.); c.randomize_imag(3., 7.);
    std::cout << m * v * alpha + c * beta;
    std::cout << c.gbmv(false, m, alpha, v, beta);
    std::cout << c * m * alpha + v * beta;
    std::cout << v.gbmv(true, m, alpha, c, beta);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(3.73e+00,7.96e-01) (6.89e+00,1.07e+01) (2.16e+00,3.28e+00)
(3.73e+00,7.96e-01) (6.89e+00,1.07e+01) (2.16e+00,3.28e+00)
(3.11e+01,2.51e+01) (-4.93e+00,1.34e+01) (1.70e+00,3.93e+00)
(3.11e+01,2.51e+01) (-4.93e+00,1.34e+01) (1.70e+00,3.93e+00)
```

### 2.4.64 `randomize_real`

Function

```
cvector& cvector::randomize_real (TR dFrom, TR dTo);
```

fills real part of calling vector with pseudo-random numbers distributed between `dFrom` and `dTo`. Function returns reference to the vector changed. See also `cvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
cvector v(3);
v.randomize_real(-2.,3.);
std::cout << v;
```

prints

```
(-4.93e-01,0.00e+00) (1.37e+00,0.00e+00) (-1.49e-01,0.00e+00)
```

### 2.4.65 `randomize_imag`

Function

```
cvector& cvector::randomize_imag (TR dFrom, TR dTo);
```

fills imaginary part of calling vector with pseudo-random numbers distributed between `dFrom` and `dTo`. Function returns reference to the vector changed. See also `cvector`.  
Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

cvector v(3);
v.randomize_imag(-2.,3.);
std::cout << v;

prints

(0.00e+00,-4.37e-01) (0.00e+00,-1.59e+00) (0.00e+00,2.42e+00)
```

## 2.5 Matrix

This base class contains member functions common for all matrices. This class is not designed to be instantiated.

```
template <typename TR, typename TC>
class Matrix : public Array<TR,TC> {
public:
    int msize () const;
    int nsize () const;
    int ld () const;
    int rowofmax () const;
    int rowofmin () const;
    int colofmax () const;
    int colofmin () const;
    virtual TR norm1 () const;
    <typename TR, typename TC>
    friend std::ostream& operator >> <> (std::istream& is,
                                           const Array<TR,TC>& mIn);

    <typename TR, typename TC>
    friend std::ostream& operator << <> (std::ostream& os,
                                           const Array<TR,TC>& mOut);
};
```

### 2.5.1 msize

Function

```
int Matrix<TR,TC>::msize () const;
```

returns number of rows of calling matrix. Function is *inherited* in all matrix classes of the library: `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srbmatrix`, `scbmatrix`, `srsmatrix` and `schmatrix`. See also `Matrix`. Example:

```
using namespace cvm;
```

```
rmatrix m (100, 200);  
std::cout << m.msize() << std::endl;
```

prints

```
100
```

### 2.5.2 nsize

Function

```
int Matrix<TR,TC>::nsize () const;
```

returns number of columns of calling matrix. Function is *inherited* in all matrix classes of the library: `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srbmatrix`, `scbmatrix`, `srsmatrix` and `schmatrix`. See also `Matrix`. Example:

```
using namespace cvm;
```

```
rmatrix m (100, 200);
```

```
std::cout << m.nsize() << std::endl;
```

prints

```
200
```



### 2.5.3 ld

Function

```
int Matrix<TR,TC>::ld () const;
```

returns leading dimension of calling matrix. Leading dimension is equal to number of rows for every matrix except submatrices. For submatrices it's equal to number of rows of parent matrix. Function is *inherited* in all matrix classes of the library: `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srbmatrix`, `scbmatrix`, `srsmatrix` and `schmatrix`. See also `Matrix`. Example:

```
using namespace cvm;
```

```
rmatrix m (100, 200);  
srmatrix ms (m, 30, 40, 5); // 5x5 submatrix  
std::cout << ms.ld() << std::endl;
```

prints

```
100
```

### 2.5.4 rowofmax

Function

```
int Matrix<TR,TC>::rowofmax () const;
```

returns *l*-based number of calling matrix row where the element with the maximum absolute value is located. Function is *inherited*<sup>5</sup> in all matrix classes of the library: `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srbmatrix`, `scbmatrix`, `srsmatrix` and `schmatrix`. See also `Matrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., -3., 1., -1.};  
rmatrix m (a, 2, 3);
```

```
std::cout << m << std::endl << m.rowofmax() << std::endl;
```

prints

```
1 2 1  
0 -3 -1
```

```
2
```

---

<sup>5</sup>Calls *virtual function* inside

### 2.5.5 rowofmin

Function

```
int Matrix<TR,TC>::rowofmin () const;
```

returns *l*-based number of calling matrix row where the element with the minimum absolute value is located. Function is *inherited*<sup>6</sup> in all matrix classes of the library: *rmatrix*, *cmatrix*, *srmatrix*, *scmatrix*, *srbmatrix*, *scbmatrix*, *srsmatrix* and *schmatrix*. See also *Matrix*. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., -3., 1., -1.};  
rmatrix m (a, 2, 3);
```

```
std::cout << m << std::endl << m.rowofmin() << std::endl;
```

prints

```
1 2 1  
0 -3 -1
```

```
2
```

---

<sup>6</sup>Calls *virtual function* inside

### 2.5.6 colofmax

Function

```
int Matrix<TR,TC>::colofmax () const;
```

returns *l*-based number of calling matrix column where the element with the maximum absolute value is located. Function is *inherited*<sup>7</sup> in all matrix classes of the library: `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srbmatrix`, `scbmatrix`, `srsmatrix` and `schmatrix`. See also `Matrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., -3., 1., -1.};  
rmatrix m (a, 2, 3);
```

```
std::cout << m << std::endl << m.colofmax() << std::endl;
```

prints

```
1 2 1  
0 -3 -1
```

```
2
```

---

<sup>7</sup>Calls *virtual function* inside

### 2.5.7 colofmin

Function

```
int Matrix<TR,TC>::colofmin () const;
```

returns *l*-based number of calling matrix column where the element with the minimum absolute value is located. Function is *inherited*<sup>8</sup> in all matrix classes of the library: `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srbmatrix`, `scbmatrix`, `srsmatrix` and `schmatrix`. See also `Matrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., -3., 1., -1.};  
rmatrix m (a, 2, 3);
```

```
std::cout << m << std::endl << m.colofmin() << std::endl;
```

prints

```
1 2 1  
0 -3 -1
```

```
1
```

---

<sup>8</sup>Calls *virtual function* inside

**2.5.8 norm1**

Virtual function

```
virtual TR Matrix<TR,TC>::norm1 () const;
```

returns 1-norm of calling matrix that is defined as

$$\|A\|_1 = \max_{j=1,\dots,n} \sum_{i=1}^m |a_{ij}|,$$

where  $A$  is  $m \times n$  matrix. Function is *inherited* in the following classes of the library: `rmatrix`, `cmatrix`, `srmatrix`, `scmatrix`, `srsmatrix` and `schmatrix`. It's *redefined* in `srbmatrix` and `scbmatrix`. See also `Array::norminf` and `Matrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., -3., 1., 0.};
rmatrix m (a, 2, 3);
```

```
std::cout << m << std::endl << m.norm1()
          << std::endl << m.norminf() << std::endl;
```

prints

```
1 2 1
0 -3 0
```

```
5
4
```

**2.5.9 operator >> <> (std::istream& is, Matrix<TR,TC>& mIn)**

Friend template operator

```
template <typename TR, typename TC>
friend std::istream& operator >> <> (std::istream& is,
                                       Matrix<TR,TC>& mIn);
```

fills calling Matrix (row by row) referenced by parameter mIn with numbers from is stream. See also [Array::operator >>](#) , [Matrix](#). Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    std::ofstream os;
    os.open ("in.txt");
    os << 1.2 << " " << 2.3 << std::endl << 3.4;
    os.close ();

    std::ifstream is("in.txt");
    rmatrix m(3,2);
    is >> m;

    std::cout << m;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
}
```

prints

```
1.20e+000 2.30e+000
3.40e+000 0.00e+000
0.00e+000 0.00e+000
```

**2.5.10 operator << >> (std::ostream& os, const Matrix<TR,TC>& mOut)**

Friend template operator

```
template <typename TR, typename TC>
friend std::ostream& operator << >> (std::ostream& os,
                                       const Matrix<TR,TC>& mOut);
```

writes matrix (row by row) referenced by parameter mOut into os stream. See also [Array::operator << , Matrix](#). Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
srmatrix m(3);
m(1,1) = 1.;
m(2,3) = 3.;
```

```
std::cout << m;
```

prints

```
1.00e+00 0.00e+00 0.00e+00
0.00e+00 0.00e+00 3.00e+00
0.00e+00 0.00e+00 0.00e+00
```



## 2.6 rmatrix

This is end-user class encapsulating matrix of real numbers.

```

template <typename TR>
class rmatrix : public Matrix <TR,TR> {
public:
    rmatrix ();
    rmatrix (int nM, int nN);
    rmatrix (TR* pD, int nM, int nN);
    rmatrix (const TR* pD, int nM, int nN);
    rmatrix (const rmatrix& m);
    explicit rmatrix (const rvector& v, bool bBeColumn = true);
    rmatrix (rmatrix& m, int nRow, int nCol, int nHeight, int nWidth);
    TR& operator () (int im, int in) throw (cvmexception);
    TR operator () (int im, int in) const throw (cvmexception);
    rvector operator () (int i) throw (cvmexception);
    const rvector operator () (int i) const throw (cvmexception);
    rvector operator [] (int i) throw (cvmexception);
    const rvector operator [] (int i) const throw (cvmexception);
    rvector diag (int i) throw (cvmexception);
    const rvector diag (int i) const throw (cvmexception);
    rmatrix& operator = (const rmatrix& m) throw (cvmexception);
    rmatrix& assign (const rvector& v) throw (cvmexception);
    rmatrix& assign (const TR* pD);
    rmatrix& assign (int nRow, int nCol, const rmatrix& m)
        throw (cvmexception);
    rmatrix& set (TR x);
    rmatrix& resize (int nNewM, int nNewN) throw (cvmexception);
    bool operator == (const rmatrix& m) const;
    bool operator != (const rmatrix& m) const;
    rmatrix& operator << (const rmatrix& m) throw (cvmexception);
    rmatrix operator + (const rmatrix& m) const
        throw (cvmexception);
    rmatrix operator - (const rmatrix& m) const
        throw (cvmexception);
    rmatrix& sum (const rmatrix& m1,
        const rmatrix& m2) throw (cvmexception);
    rmatrix& diff (const rmatrix& m1,
        const rmatrix& m2) throw (cvmexception);
    rmatrix& operator += (const rmatrix& m) throw (cvmexception);
    rmatrix& operator -= (const rmatrix& m) throw (cvmexception);
    rmatrix operator - () const;

```

```

rmatrix operator * (TR d) const;
rmatrix operator / (TR d) const
    throw (cvmexception);
rmatrix& operator *= (TR d);
rmatrix& operator /= (TR d) throw (cvmexception);
rmatrix& normalize ();
rmatrix operator ~ () const throw (cvmexception);
rmatrix& transpose (const rmatrix& m) throw (cvmexception);
rmatrix& transpose () throw (cvmexception);
rvector operator * (const rvector& v) const
    throw (cvmexception);
rmatrix operator * (const rmatrix& m) const
    throw (cvmexception);
rmatrix& mult (const rmatrix& m1, const rmatrix& m2)
    throw (cvmexception);
rmatrix& ranklupdate (const rvector& vCol,
    const rvector& vRow)
    throw (cvmexception);
rmatrix& swap_rows (int n1, int n2) throw (cvmexception);
rmatrix& swap_cols (int n1, int n2) throw (cvmexception);
rmatrix& solve (const srmatrix& mA,
    const rmatrix& mB, TR& dErr)
    throw (cvmexception);
rmatrix& solve (const srmatrix& mA,
    const rmatrix& mB) throw (cvmexception);
rmatrix& solve_tran (const srmatrix& mA,
    const rmatrix& mB, TR& dErr)
    throw (cvmexception);
rmatrix& solve_tran (const srmatrix& mA,
    const rmatrix& mB) throw (cvmexception);
rmatrix& solve_lu (const srmatrix& mA, const srmatrix& mLU,
    const int* pPivots, const rmatrix& mB, TR& dErr)
    throw (cvmexception);
rmatrix& solve_lu (const srmatrix& mA, const srmatrix& mLU,
    const int* pPivots, const rmatrix& mB)
    throw (cvmexception);
rvector svd () const throw (cvmexception);
rvector svd (srmatrix& mU, srmatrix& mVH) const
    throw (cvmexception);
rmatrix pinv (TR threshold = cvmMachSp()) const
    throw (cvmexception);
rmatrix& pinv (const rmatrix& mA, TR threshold = cvmMachSp())
    throw (cvmexception);

```

```

rmatrix gels (bool transpose, const rmatrix& mB,
              rvector& vErr) const throw (cvmexception);
rmatrix& gels (bool transpose, const rmatrix& mA, const rmatrix& mB,
              rvector& vErr) throw (cvmexception);
rvector gels (bool transpose, const rvector& mB,
              TR& dErr) const throw (cvmexception);
rmatrix gelsy (const rmatrix& mB, int& rank,
              TR tol = cvmMachSp()) const throw (cvmexception);
rmatrix& gelsy (const rmatrix& mA, const rmatrix& mB, int& rank,
              TR tol = cvmMachSp()) throw (cvmexception);
rvector gelsy (const rvector& mB, int& rank,
              TR tol = cvmMachSp()) const throw (cvmexception);
rmatrix gelss (const rmatrix& mB, rvector& sv,
              int& rank, TR tol = cvmMachSp()) const
              throw (cvmexception);
rmatrix& gelss (const rmatrix& mA, const rmatrix& mB, rvector& sv,
              int& rank, TR tol = cvmMachSp())
              throw (cvmexception);
rvector gelss (const rvector& mB, rvector& sv,
              int& rank, TR tol = cvmMachSp()) const
              throw (cvmexception);
rmatrix gelsd (const rmatrix& mB, rvector& sv,
              int& rank, TR tol = cvmMachSp()) const
              throw (cvmexception);
rmatrix& gelsd (const rmatrix& mA, const rmatrix& mB, rvector& sv,
              int& rank, TR tol = cvmMachSp())
              throw (cvmexception);
rvector gelsd (const rvector& mB, rvector& sv,
              int& rank, TR tol = cvmMachSp()) const
              throw (cvmexception);
int rank (TR eps = cvmMachSp ()) const throw (cvmexception);
rmatrix& ger (TR dAlpha, const rvector& vCol,
             const rvector& vRow) throw (cvmexception);
rmatrix& gemm (const rmatrix& m1, bool bTrans1,
             const rmatrix& m2, bool bTrans2,
             TR dAlpha, TR dBeta) throw (cvmexception);
rmatrix& symm (bool bLeft, const srmatrix& ms,
             const rmatrix& m, TR dAlpha, TR dBeta)
             throw (cvmexception);
void qr(rmatrix& mQ, srmatrix& mR) const throw (cvmexception);
void qr(srmatrix& mQ, rmatrix& mR) const throw (cvmexception);
void lq(srmatrix& mL, rmatrix& mQ) const throw (cvmexception);
void lq(rmatrix& mL, srmatrix& mQ) const throw (cvmexception);

```

```
void rq(srmatrix& mR, rmatrix& mQ) const throw (cvmexception);  
void rq(rmatrix& mR, srmatrix& mQ) const throw (cvmexception);  
void ql(rmatrix& mQ, srmatrix& mL) const throw (cvmexception);  
void ql(srmatrix& mQ, rmatrix& mL) const throw (cvmexception);  
rmatrix& vanish ();  
rmatrix& randomize (TR dFrom, TR dTo);  
};
```

### 2.6.1 `rmatrix ()`

Default constructor

```
rmatrix::rmatrix ();
```

creates empty `rmatrix` object. See also `rmatrix`. Example:

```
using namespace cvm;
```

```
rmatrix m;  
std::cout << m.msize() << std::endl << m.nsize() << std::endl;  
std::cout << m.size() << std::endl;
```

```
m.resize (2, 3);  
std::cout << m;
```

prints

```
0  
0  
0  
0 0 0  
0 0 0
```

### 2.6.2 `rmatrix (int,int)`

Constructor

```
rmatrix::rmatrix (int nM, int nN);
```

creates  $m \times n$  `rmatrix` object where  $m$  is passed in `nM` parameter (number of rows) and  $n$  is passed in `nN` (number of columns). Constructor sets all elements to zero. It throws `cvmexception` in case of non-positive sizes passed or memory allocation failure. See also `rmatrix`. Example:

```
using namespace cvm;
```

```
rmatrix m (3, 4);  
std::cout << m.msize() << std::endl << m.nsize()  
          << std::endl << m.size() << std::endl << m;
```

prints

```
3  
4  
12  
0 0 0 0  
0 0 0 0  
0 0 0 0
```

### 2.6.3 *rmatrix* (TR\*,int,int)

Constructor

```
rmatrix::rmatrix (TR* pD, int nM, int nN);
```

creates  $m \times n$  *rmatrix* object where  $m$  is passed in  $nM$  parameter (number of rows) and  $n$  is passed in  $nN$  (number of columns). Unlike others, this constructor *does not allocate memory*. It just shares memory with array pointed to by  $pD$ . It throws `cvmexception` in case of non-positive size passed. See also `rmatrix`, `rmatrix (const TR*,int,int)`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
double a[] = {1., 2., 3., 4., 5., 6.};
rmatrix m (a, 2, 3);

m(1,1) = 7.77;
std::cout << m << std::endl;
std::cout << a[0] << " " << a[1] << " " << a[2] << std::endl;
```

prints

```
7.77e+000 3.00e+000 5.00e+000
2.00e+000 4.00e+000 6.00e+000

7.77e+000 2.00e+000 3.00e+000
```

### 2.6.4 `rmatrix (const TR*,int,int)`

Constructor

```
rmatrix::rmatrix (const TR* pD, int nM, int nN);
```

creates  $m \times n$  `rmatrix` object where  $m$  is passed in `nM` parameter (number of rows) and  $n$  is passed in `nN` (number of columns) and copies  $m * n$  elements of array `pD` to it by **columns**. It throws **`cvmexception`** in case of non-positive size passed. See also **`rmatrix`**, **`rmatrix (TR*,int,int)`**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
const double a[] = {1., 2., 3., 4., 5., 6.};
rmatrix m (a, 2, 3);

m(1,1) = 7.77;
std::cout << m << std::endl;
std::cout << a[0] << " " << a[1] << " " << a[2] << std::endl;
```

prints

```
7.77e+000 3.00e+000 5.00e+000
2.00e+000 4.00e+000 6.00e+000

1.00e+000 2.00e+000 3.00e+000
```



### 2.6.5 `rmatrix (const rmatrix&)`

Copy constructor

```
rmatrix::rmatrix (const rmatrix& m);
```

creates `rmatrix` object as a copy of `m`. It throws `cvmexception` in case of memory allocation failure. See also `rmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
double a[] = {1., 2., 3., 4., 5., 6.};
rmatrix m (a, 2, 3);
rmatrix mc(m);
```

```
m(1,1) = 7.77;
std::cout << m << std::endl << mc;
```

prints

```
7.77e+000 3.00e+000 5.00e+000
2.00e+000 4.00e+000 6.00e+000

1.00e+000 3.00e+000 5.00e+000
2.00e+000 4.00e+000 6.00e+000
```

### 2.6.6 `rmatrix (const rvector&,bool)`

Constructor

```
explicit rmatrix::rmatrix (const rvector& v, bool bBeColumn = true);
```

creates `rmatrix` object containing `v.size()` rows and 1 column if `bBeColumn` is true or 1 row and `v.size()` columns otherwise. After that it copies vector `v`'s elements to the matrix created. Constructor throws `cvmexception` in case of memory allocation failure. See also `rmatrix`, `rvector`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
rvector v(3);
v(1) = 1.;
v(2) = 2.;
v(3) = 3.;

rmatrix mc (v);
rmatrix mr (v, false);
```

```
std::cout << mc << std::endl << mr;
```

prints

```
1.00e+000
2.00e+000
3.00e+000
```

```
1.00e+000 2.00e+000 3.00e+000
```

### 2.6.7 **submatrix**

Submatrix constructor

```
rmatrix::rmatrix (rmatrix& m, int nRow, int nCol,  
                 int nHeight, int nWidth);
```

creates *rmatrix* object as *submatrix* of *m*. It means that the matrix object created shares memory with some part of *m*. This part is defined by its upper left corner (parameters *nRow* and *nCol*, both are *l*-based) and its height and width (parameters *nHeight* and *nWidth*). See also *rmatrix*. Example:

```
using namespace cvm;  
rmatrix m(4,5);  
rmatrix subm(m, 2, 2, 2, 2);  
subm.set(1.);
```

```
std::cout << m;
```

prints

```
0 0 0 0 0  
0 1 1 0 0  
0 1 1 0 0  
0 0 0 0 0
```

### 2.6.8 operator (,)

Indexing operators

```
TR& rmatrix::operator () (int im, int in) throw (cvmexception);
TR rmatrix::operator () (int im, int in) const throw (cvmexception);
```

provide access to a particular element of calling matrix. The first version is applicable to non-constant object. This version returns *l-value* in order to make possible write access to an element. Both operators are *l-based*. Operators throw `cvmexception` if `im` is outside of `[1,msize()]` range or `in` is outside of `[1,nsize()]` range. Operators are *inherited* in the classes `srmatrix` and `srbmatrix`. Operators are *redefined* in the class `srsmatrix`. See also `rmatrix`, `Matrix::msize()`, `Matrix::nsize()`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    double a[] = {1., 2., 3., 4., 5., 6.};
    const rmatrix m (a, 2, 3);
    rmatrix ms(m);

    std::cout << m(1,1) << " " << m(2,3) << std::endl << std::endl;

    ms(2,2) = 7.77;
    std::cout << ms;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

1.00e+00 6.00e+00

1.00e+00 3.00e+00 5.00e+00
2.00e+00 7.77e+00 6.00e+00
```

### 2.6.9 operator ()

Indexing operators

```
rvector rmatrix::operator () (int i) throw (cvmexception);
const rvector rmatrix::operator () (int i) const throw (cvmexception);
```

provide access to *i*-th column of calling matrix. The first version is applicable to non-constant object and *returns l-value*, i.e. vector returned shares memory with *i*-th column of the matrix in order to make possible write access to it. The second version creates *copy* of a column and therefore is *not l-value*. Both operators are *l-based*. Operators throw **cvmexception** if the parameter *i* is outside of [1, nsize()] range. Operators are *inherited* in the class **rmatrix**. Operators are *redefined* in the classes **srbmatrix** and **srsrmatrix**. See also **rmatrix**, **Matrix::nsize()**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    double a[] = {1., 2., 3., 4., 5., 6.};
    const rmatrix m (a, 2, 3);
    srmatrix ms(2);

    std::cout << m(2) << std::endl;

    ms(2) = m(3);
    std::cout << ms;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
3.00e+00 4.00e+00

0.00e+00 5.00e+00
0.00e+00 6.00e+00
```

**2.6.10 operator []**

Indexing operators

```
rvector rmatrix::operator [] (int i) throw (cvmexception);
const rvector rmatrix::operator [] (int i) const throw (cvmexception);
```

provide access to *i*-th row of matrix. The first version is applicable to non-constant object and *returns l-value*, i.e. vector returned shares memory with *i*-th row of the matrix in order to make possible write access to it. The second version creates *copy* of a row and therefore is *not l-value*. Both operators are *ℓ-based*. Operators throw **cvmexception** if *i* is outside of `[1,msize())` range. Operators are *inherited* in the class **srmatrix**. Operators are *redefined* in the classes **srbmatrix** and **srsmatrix**. See also **rmatrix**, **Matrix::msize()**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    double a[] = {1., 2., 3., 4., 5., 6.};
    const rmatrix m (a, 2, 3);
    srmatrix ms(3);

    std::cout << m[1] << std::endl;

    ms[1] = m[2];
    std::cout << ms;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
1.00e+00 3.00e+00 5.00e+00

2.00e+00 4.00e+00 6.00e+00
0.00e+00 0.00e+00 0.00e+00
0.00e+00 0.00e+00 0.00e+00
```

**2.6.11 diag**

Functions

```
rvector rmatrix::diag (int i) throw (cvmexception);
const rvector rmatrix::diag (int i) const throw (cvmexception);
```

provide access to *i*-th diagonal of calling matrix, where  $i = 0$  for main diagonal,  $i < 0$  for lower diagonals and  $i > 0$  for upper ones. The first version is applicable to non-constant object and *returns l-value*, i.e. vector returned shares memory with *i*-th diagonal of the matrix in order to make possible write access to it. The second version creates *copy* of a diagonal and therefore is *not l-value*. Functions throw **cvmexception** if parameter *i* is outside of  $[-m\text{size}()+1, n\text{size}()-1]$  range. Functions are *inherited* in the classes **smatrix** and **srbmatrix**. Functions are *redefined* in the class **srsmatrix**. See also **rmatrix**. Example:

```
using namespace cvm;
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
    rmatrix m(2,3);
    const smatrix ms(a,3);

    m.diag(-1).set(1.);
    m.diag(0).set(2.);
    m.diag(1).set(3.);
    m.diag(2).set(4.);
    std::cout << m << std::endl;

    std::cout << ms << std::endl;
    std::cout << ms.diag(0) << ms.diag(1);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
2 3 4
1 2 3
```

```
1 4 7
2 5 8
3 6 9
```

```
1 5 9
4 8
```

### 2.6.12 `operator = (const rmatrix&)`

Operator

```
rmatrix& rmatrix::operator = (const rmatrix& m) throw (cvmexception);
```

sets every element of calling matrix to be equal to appropriate element of matrix `m` and returns reference to the matrix changed. Operator throws `cvmexception` in case of different matrix sizes. Operator is *redefined* in the classes `srmatrix`, `srbmatrix` and `srsmatrix`. See also `rmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    double a[] = {1., 2., 3., 4., 5., 6.};
    const rmatrix m1(a, 3, 2);
    rmatrix m2(3, 2);

    m2 = m1;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

1.00e+000 4.00e+000
2.00e+000 5.00e+000
3.00e+000 6.00e+000
```



### 2.6.13 `assign (const TR*)`

Function

```
rmatrix& rmatrix::assign (const rvector& v) throw (cvmexception);  
rmatrix& rmatrix::assign (const TR* pD);
```

sets every element of calling matrix to be equal to appropriate element of vector `v` or array pointed to by `pD` and returns reference to the matrix changed. In first version it's assumed that vector passed is long enough to fill calling matrix. Function throws `cvmexception` otherwise. Function is *redefined* in the classes `srmatrix`, `srbmatrix` and `srsmatrix`. See also `rmatrix`. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);  
const double a[] = {1., 2., 3., 4., 5., 6.};  
rmatrix m(2, 3);  
  
m.assign(a);  
std::cout << m;
```

prints

```
1.00e+000 3.00e+000 5.00e+000  
2.00e+000 4.00e+000 6.00e+000
```

**2.6.14 assign (int, int, const rmatrix&)**

Function

```
rmatrix& rmatrix::assign (int nRow, int nCol, const rmatrix& m)
throw (cvmexception);
```

sets sub-matrix of calling matrix beginning with *l*-based row nRow and column nCol to matrix m and returns reference to the matrix changed. Function throws *cvmexception* if nRow or nCol are not positive or matrix m doesn't fit. Function is *redefined* in the classes *srmatrix*, *srbmatrix* and *srsmatrix*. See also *rmatrix*. Example:

```
using namespace cvm;
```

```
rmatrix m1(4,5);
rmatrix m2(2,2);
m1.set(1.);
m2.set(2.);
m1.assign(2,3,m2);
std::cout << m1;
```

prints

```
1 1 1 1 1
1 1 2 2 1
1 1 2 2 1
1 1 1 1 1
```

### 2.6.15 `set (TR)`

Function

```
rmatrix& rmatrix::set (TR x);
```

sets every element of calling matrix to be equal to parameter `x` and returns reference to the matrix changed. Use `vanish` to set every element of calling matrix to be equal to zero. Function is *redefined* in the classes `srmatrix`, `srbmatrix` and `srsmatrix`. See also `rmatrix`.  
Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);  
rmatrix m(2, 3);
```

```
m.set(3.);  
std::cout << m;
```

prints

```
3.00e+000 3.00e+000 3.00e+000  
3.00e+000 3.00e+000 3.00e+000
```

**2.6.16 *resize***

Function

```
rmatrix& rmatrix::resize (int nNewM, int nNewN)
throw (cvmexception);
```

changes size of calling matrix to *nNewM* by *nNewN* and returns reference to the matrix changed. In case of increasing of its size, the matrix is filled up with zeroes. Function throws ***cvmexception*** in case of negative size passed or memory allocation failure. Function is *redefined* in the classes ***srmatrix***, ***srbmatrix*** and ***srsmatrix***. See also ***rmatrix***. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    double a[] = {1., 2., 3., 4., 5., 6.};
    rmatrix m(a, 2, 3);
    std::cout << m << std::endl;
    m.resize (2, 2);
    std::cout << m << std::endl;
    m.resize (3, 3);
    std::cout << m;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
1.00e+000 3.00e+000 5.00e+000
2.00e+000 4.00e+000 6.00e+000
```

```
1.00e+000 3.00e+000
2.00e+000 4.00e+000
```

```
1.00e+000 3.00e+000 0.00e+000
2.00e+000 4.00e+000 0.00e+000
0.00e+000 0.00e+000 0.00e+000
```

### 2.6.17 operator ==

Operator

```
bool rmatrix::operator == (const rmatrix& m) const;
```

compares calling matrix with matrix *m* and returns true if they have the same sizes and their appropriate elements differ by not more than the **smallest normalized positive number**. Returns false otherwise. Operator is *redefined* in the classes **srmatrix**, **srbmatrix** and **srsmatrix**. See also **rmatrix**. Example:

```
using namespace cvm;
double a[] = {1., 2., 3., 4.};
rmatrix m1(a, 2, 2);
rmatrix m2(2, 2);

m2(1,1) = 1.; m2(1,2) = 3.;
m2(2,1) = 2.; m2(2,2) = 4.;

std::cout << (m1 == m2) << std::endl;

prints

1
```

### 2.6.18 operator !=

Operator

```
bool rmatrix::operator != (const rmatrix& m) const;
```

compares calling matrix with matrix *m* and returns true if they have different sizes or at least one of their appropriate elements differs by more than the **smallest normalized positive number**. Returns false otherwise. Operator is *redefined* in the classes **srmatrix**, **srbmatrix** and **srsrmatrix**. See also **rmatrix**. Example:

```
using namespace cvm;
double a[] = {1., 2., 3., 4.};
rmatrix m1(a, 2, 2);
rmatrix m2(2, 2);

m2(1,1) = 1.; m2(1,2) = 3.;
m2(2,1) = 2.; m2(2,2) = 4.;

std::cout << (m1 != m2) << std::endl;

prints

0
```

**2.6.19 operator <<**

Operator

```
rmatrix& rmatrix::operator << (const rmatrix& m)
throw (cvmexception);
```

destroys calling matrix, creates new one as a copy of *m* and returns reference to the matrix changed. Operator throws **cvmexception** in case of memory allocation failure. Operator is *redefined* in the classes **srmatrix**, **srbmatrix** and **srsmatrix**. See also **rmatrix**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    rmatrix m(3,4);
    rmatrix mc(1,1);
    m(1,2) = 1.;
    m(3,4) = 2.;
    std::cout << m << mc << std::endl;

    mc << m;
    std::cout << mc;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
0.00e+000 1.00e+000 0.00e+000 0.00e+000
0.00e+000 0.00e+000 0.00e+000 0.00e+000
0.00e+000 0.00e+000 0.00e+000 2.00e+000
0.00e+000

0.00e+000 1.00e+000 0.00e+000 0.00e+000
0.00e+000 0.00e+000 0.00e+000 0.00e+000
0.00e+000 0.00e+000 0.00e+000 2.00e+000
```

### 2.6.20 operator +

Operator

```
rmatrix rmatrix::operator + (const rmatrix& m) const  
throw (cvmexception);
```

creates object of type `rmatrix` as sum of calling matrix and matrix `m`. Operator throws `cvmexception` in case of different sizes of the operands. Operator is *redefined* in the classes `srmatrix`, `srbmatrix` and `srsmatrix`. See also `rmatrix::sum`, `rmatrix`. Example:

```
using namespace cvm;  
try {  
    double a[] = {1., 2., 3., 4., 5., 6.};  
    const rmatrix ma(a,2,3);  
    rmatrix mb(2,3);  
    mb.set(1.);  
  
    std::cout << ma + mb << std::endl;  
    std::cout << ma + ma;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
2 4 6  
3 5 7
```

```
2 6 10  
4 8 12
```



### 2.6.21 operator -

Operator

```
rmatrix rmatrix::operator - (const rmatrix& m) const  
throw (cvmexception);
```

creates object of type `rmatrix` as difference of calling matrix and matrix `m`. It throws `cvmexception` in case of different sizes of the operands. Operator is *redefined* in the classes `srmatrix`, `srbmatrix` and `srsmatrix`. See also `rmatrix::diff`, `rmatrix`. Example:

```
using namespace cvm;  
try {  
    double a[] = {1., 2., 3., 4., 5., 6.};  
    const rmatrix ma(a,2,3);  
    rmatrix mb(2,3);  
    mb.set(1.);  
  
    std::cout << ma - mb << std::endl;  
    std::cout << ma - ma;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
0 2 4
```

```
1 3 5
```

```
0 0 0
```

```
0 0 0
```

### 2.6.22 `sum`

Function

```
rmatrix& rmatrix::sum (const rmatrix& m1, const rmatrix& m2)
throw (cvmexception);
```

assigns result of addition of matrices `m1` and `m2` to calling matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. Function is *redefined* in the classes `srmatrix`, `srbmatrix` and `srsmatrix`. See also `rmatrix::operator +`, `rmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6.};
    const rmatrix m1(a, 2, 3);
    rmatrix m2(2, 3);
    rmatrix m(2, 3);
    m2.set(1.);

    std::cout << m.sum(m1, m2) << std::endl;
    std::cout << m.sum(m, m2);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
2 4 6
3 5 7
```

```
3 5 7
4 6 8
```

**2.6.23 diff**

Function

```
rmatrix& rmatrix::diff (const rmatrix& m1, const rmatrix& m2)
throw (cvmexception);
```

assigns result of subtraction of matrices *m1* and *m2* to calling matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. Function is *redefined* in the classes `srmatrix`, `srbmatrix` and `srsmatrix`. See also `rmatrix::operator -`, `rmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6.};
    const rmatrix m1(a, 2, 3);
    rmatrix m2(2, 3);
    rmatrix m(2, 3);
    m2.set(1.);

    std::cout << m.diff(m1, m2) << std::endl;
    std::cout << m.diff(m, m2);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
0 2 4
1 3 5
```

```
-1 1 3
0 2 4
```

### 2.6.24 operator +=

Operator

```
rmatrix& rmatrix::operator += (const rmatrix& m) throw (cvmexception);
```

adds matrix *m* to calling matrix and returns reference to the matrix changed. It throws **cvmexception** in case of different sizes of the operands. Operator is *redefined* in the classes **srmatrix**, **srbmatrix** and **srsmatrix**. See also **rmatrix::operator +** , **rmatrix::sum**, **rmatrix**. Example:

```
using namespace cvm;
try {
    rmatrix m1(2, 3);
    rmatrix m2(2, 3);
    m1.set(1.);
    m2.set(2.);

    m1 += m2;
    std::cout << m1 << std::endl;

    // well, you can do this too, but temporary object would be created
    m2 += m2;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

3 3 3
3 3 3

4 4 4
4 4 4
```

**2.6.25 operator -=**

Operator

```
rmatrix& rmatrix::operator -= (const rmatrix& m) throw (cvmexception);
```

subtracts matrix *m* from calling matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. Operator is *redefined* in the classes `srmatrix`, `srbmatrix` and `srsmatrix`. See also `rmatrix::operator -`, `rmatrix::diff`, `rmatrix`. Example:

```
using namespace cvm;
try {
    rmatrix m1(2, 3);
    rmatrix m2(2, 3);
    m1.set(1.);
    m2.set(2.);

    m1 -= m2;
    std::cout << m1 << std::endl;

    // well, you can do this too, but temporary object would be created
    m2 -= m2;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

-1 -1 -1
-1 -1 -1

0 0 0
0 0 0
```

### 2.6.26 `operator - ()`

Operator

```
rmatrix rmatrix::operator - () const throw (cvmexception);
```

creates object of type `rmatrix` as calling matrix multiplied by  $-1$ . Operator is *redefined* in the classes `srmatrix`, `srbmatrix` and `srsmatrix`. See also `rmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6.};  
const rmatrix ma(a, 2, 3);
```

```
std::cout << - ma;
```

prints

```
-1 -3 -5  
-2 -4 -6
```

### 2.6.27 operator \* (TR)

Operator

```
rmatrix rmatrix::operator * (TR d) const;
```

creates object of type `rmatrix` as product of calling matrix and number `d`. Operator is *redefined* in the classes `srmatrix`, `srbmatrix` and `srsmatrix`. See also `rmatrix::operator *=`, `rmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6.};  
rmatrix m(a, 2, 3);
```

```
std::cout << m * 2.;
```

prints

```
2 6 10  
4 8 12
```

### 2.6.28 `operator /` (TR)

Operator

```
rmatrix rmatrix::operator / (TR d) const throw (cvmexception);
```

creates object of type `rmatrix` as quotient of calling matrix and number `d`. It throws `cvmexception` if `d` has absolute value equal or less than the **smallest normalized positive number**. Operator is *redefined* in the classes `srmatrix`, `srbmatrix` and `srsmatrix`. See also `rmatrix::operator /=`, `rmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    double a[] = {1., 2., 3., 4., 5., 6.};
    rmatrix m(a, 2, 3);

    std::cout << m / 2.;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

5.00e-01 1.50e+00 2.50e+00
1.00e+00 2.00e+00 3.00e+00
```



**2.6.29 operator \*= (TR)**

Operator

```
rmatrix& rmatrix::operator *= (TR d);
```

multiplies calling matrix by number *d* and returns reference to the matrix changed. Operator is *redefined* in the classes `srmatrix`, `srbmatrix` and `srsmatrix`. See also `rmatrix`, `rmatrix::operator *`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6.};  
rmatrix m(a, 2, 3);
```

```
m *= 2.;  
std::cout << m;
```

prints

```
2 6 10  
4 8 12
```

### 2.6.30 operator /= (TR)

Operator

```
rmatrix& rmatrix::operator /= (TR d) throw (cvmexception);
```

divides calling matrix by number d and returns reference to the matrix changed. It throws **cvmexception** if d has absolute value equal or less than the **smallest normalized positive number**. Operator is *redefined* in the classes **srmatrix**, **srbmatrix** and **srsmatrix**. See also **rmatrix::operator /** , **rmatrix**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    double a[] = {1., 2., 3., 4., 5., 6.};
    rmatrix m(a, 2, 3);

    m /= 2.;
    std::cout << m;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

5.00e-01 1.50e+00 2.50e+00
1.00e+00 2.00e+00 3.00e+00
```

### 2.6.31 `normalize`

Function

```
rmatrix& rmatrix::normalize ();
```

normalizes calling matrix so its **Euclidean norm** becomes equal to 1 if it was greater than the **smallest normalized positive number** before the call (otherwise the function does nothing). Function is *redefined* in the classes **srmatrix**, **srbmatrix** and **srsrmatrix**. See also **rmatrix**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
double a[] = {1., 2., 3., 4., 5., 6.};
rmatrix m(a, 2, 3);
```

```
m.normalize();
std::cout << m;
```

prints

```
1.05e-01 3.14e-01 5.24e-01
2.10e-01 4.19e-01 6.29e-01
```

**2.6.32 transposition**

Operator and functions

```
rmatrix rmatrix::operator ~ () const throw (cvmexception);
rmatrix& rmatrix::transpose (const rmatrix& m) throw (cvmexception);
rmatrix& rmatrix::transpose () throw (cvmexception);
```

implement matrix transposition. First operator creates object of type `rmatrix` as transposed calling matrix (it throws `cvmexception` in case of memory allocation failure). Second function sets calling matrix to be equal to matrix `m` transposed (it throws `cvmexception` in case of not appropriate sizes of the operands), third one makes it to be equal to transposed itself (it also throws `cvmexception` in case of memory allocation failure). Functions are *redefined* in the classes `srmatrix`, `srbmatrix` and `srsmatrix`. See also `rmatrix`. Example:

```
using namespace cvm;
try {
    double a[] = {1., 2., 3., 4., 5., 6.};
    rmatrix m(a,2,3);
    rmatrix mt(3,2);
    std::cout << m << std::endl << ~m << std::endl ;
    mt.transpose(m);
    std::cout << mt << std::endl;
    mt.transpose();
    std::cout << mt;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
1 3 5
2 4 6
```

```
1 2
3 4
5 6
```

```
1 2
3 4
5 6
```

```
1 3 5
2 4 6
```

### 2.6.33 `operator * (const rvector&)`

Operator

```
rvector rmatrix::operator * (const rvector& v) const  
throw (cvmexception);
```

creates object of type `rvector` as product of calling matrix and vector `v`. It throws `cvmexception` if the number of columns of the calling matrix differs from size of the vector `v`. Use `rvector::mult` to avoid new object creation. Function is *redefined* in the classes `srmatrix`, `srbmatrix` and `srsmatrix`. See also `rmatrix`, `rvector`. Example:

```
using namespace cvm;
```

```
try {  
    rmatrix m(2, 3);  
    rvector v(3);  
    m.set(1.);  
    v.set(1.);  
  
    std::cout << m * v;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
3 3
```

### 2.6.34 operator \* (const rmatrix&)

Operator

```
rmatrix rmatrix::operator * (const rmatrix& m) const  
throw (cvmexception);
```

creates object of type `rmatrix` as product of calling matrix and matrix `m`. It throws `cvmexception` if number of columns of calling matrix differs from number of rows of the matrix `m`. Use `rmatrix::mult` to avoid new object creation. Operator is *redefined* in the classes `srmatrix`, `srbmatrix` and `srsmatrix`. See also `rmatrix`. Example:

```
using namespace cvm;
```

```
try {  
    rmatrix m1(2, 3);  
    rmatrix m2(3, 2);  
    m1.set(1.);  
    m2.set(1.);  
  
    std::cout << m1 * m2;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
3 3  
3 3
```

### 2.6.35 `mult`

Function

```
rmatrix& rmatrix::mult (const rmatrix& m1, const rmatrix& m2)
throw (cvmexception);
```

sets calling matrix to be equal to product of matrix `m1` by matrix `m2` and returns reference to the matrix changed. Function throws `cvmexception` in case of inappropriate sizes of the operands. Function is *inherited* in the class `srmatrix` and *redefined* in the classes `srbmatrix` and `srsmatrix`. See also `rmatrix`. Example:

```
using namespace cvm;
```

```
try {
    rmatrix m1(2, 3);
    rmatrix m2(3, 2);
    rmatrix m(2, 2);
    m1.set(1.);
    m2.set(1.);

    std::cout << m.mult(m1, m2) << std::endl;
    std::cout << m1.mult(m, m1);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
3 3
3 3
```

```
6 6 6
6 6 6
```

### 2.6.36 `ranklupdate`

Function

```
rmatrix&  
rmatrix::ranklupdate (const rvector& vCol, const rvector& vRow)  
throw (cvmexception);
```

sets calling matrix to be equal to **rank-1 update** of vectors `vCol` and `vRow` and returns reference to the matrix changed. Function throws **cvmexception** if number of rows of calling matrix is not equal to `vCol.size()` or number of columns of calling matrix is not equal to `vRow.size()`. Function is *inherited* in the class **srmatrix** and *not applicable* to objects of the classes **srbmatrix** and **srsmatrix** (i.e. exception of type **cvmexception** would be thrown in case of using it for objects of those classes). See also **rvector::ranklupdate**, **rmatrix**. Example:

```
using namespace cvm;  
  
try {  
    rvector vc(3), vr(2);  
    rmatrix m(3, 2);  
    vc(1) = 1.;  
    vc(2) = 2.;  
    vc(3) = 3.;  
    vr(1) = 4.;  
    vr(2) = 5.;  
  
    std::cout << m.ranklupdate (vc, vr);  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}  
  
prints  
  
4 5  
8 10  
12 15
```



### 2.6.37 `swap_rows`

Function

```
rmatrix& rmatrix::swap_rows (int n1, int n2) throw (cvmexception);
```

swaps two rows of calling matrix and returns reference to the matrix changed. `n1` and `n2` are numbers of rows to be swapped, both are *ℓ-based*). Function throws `cvmexception` if one of the parameters is outside of the range `[1,msize()]`. Function is *redefined* in the class `srmatrix` and *not applicable* to objects of the classes `srbmatrix` and `srsmatrix` (i.e. exception of type `cvmexception` would be thrown in case of using it for objects of those classes). See also `rmatrix`. Example:

```
using namespace cvm;
try {
    double a[] = {1., 2., 3., 4., 5., 6.};
    rmatrix m (a, 3, 2);

    std::cout << m << std::endl;
    std::cout << m.swap_rows(2,3);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
1 4
2 5
3 6
```

```
1 4
3 6
2 5
```

### 2.6.38 `swap_cols`

Function

```
rmatrix& rmatrix::swap_cols (int n1, int n2) throw (cvmexception);
```

swaps two columns of calling matrix and returns reference to the matrix changed. `n1` and `n2` are numbers of columns to be swapped, both are *l-based*). Function throws `cvmexception` if one of the parameters is outside of the range `[1, nsize()]`. Function is *redefined* in the class `srmatrix` and *not applicable* to objects of the classes `srbmatrix` and `srsmatrix` (i.e. exception of type `cvmexception` would be thrown in case of using it for objects of those classes). See also `rmatrix`. Example:

```
using namespace cvm;
try {
    double a[] = {1., 2., 3., 4., 5., 6.};
    rmatrix m (a, 2, 3);

    std::cout << m << std::endl;
    std::cout << m.swap_cols(2,3);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
1 3 5
```

```
2 4 6
```

```
1 5 3
```

```
2 6 4
```

**2.6.39 solve**

Functions

```

rmatrix&
rmatrix::solve (const srmatrix& mA,
                const rmatrix& mB, TR& dErr) throw (cvmexception);

rmatrix&
rmatrix::solve (const srmatrix& mA,
                const rmatrix& vB) throw (cvmexception);

```

set calling matrix to be equal to solution  $X$  of matrix linear equation  $A * X = B$  where parameter `mA` is square matrix  $A$  and parameter `vB` is matrix  $B$ . Every function returns reference to the matrix changed. The first version also sets output parameter `dErr` to be equal to the norm of computation error. Functions throw `cvmexception` in case of inappropriate sizes of the operands or when the matrix  $A$  is close to cingular. Functions are *redefined* in the class `srmatrix` and *inherited* thereafter in the classes `srbmatrix` and `srsrmatrix`. See also `rvector::solve`, `rmatrix`, `srmatrix`. Example:

```

using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (2);
try {
    srmatrix ma(3);
    rmatrix mb(3,4);
    rmatrix mx(3,4);
    double dErr;
    ma.randomize(-10., 10.);
    mb.randomize(-10., 10.);

    mx.solve (ma, mb, dErr);

    std::cout << ma * mx - mb << dErr << std::endl;
}
catch (cvmexception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

```

prints

```

-1.78e-015 +1.80e-016 -8.88e-016 +0.00e+000
+0.00e+000 -1.33e-015 +8.88e-016 +0.00e+000
+0.00e+000 +0.00e+000 -1.78e-015 +0.00e+000
+2.02e-015

```

**2.6.40 solve\_tran**

Functions

```
rmatrix&
rmatrix::solve_tran (const srmatrix& mA,
                    const rmatrix& mB, TR& dErr) throw (cvmexception);
```

```
rmatrix&
rmatrix::solve_tran (const srmatrix& mA,
                    const rmatrix& vB) throw (cvmexception);
```

set calling matrix to be equal to solution  $X$  of matrix linear equation  $A^T * X = B$  (which is equivalent to  $X^T * A = B^T$ ) where parameter `mA` is square matrix  $A$  and parameter `vB` is matrix  $B$ . Every function returns reference to the matrix changed. The first version also sets output parameter `dErr` to be equal to the norm of computation error. Functions throw `cvmexception` in case of inappropriate sizes of the operands or when the matrix  $A$  is close to singular. Functions are *redefined* in the class `srmatrix` and *inherited* thereafter in the classes `srbmatrix` and `srsrmatrix`. See also `rmatrix::solve`, `rvector::solve_tran`, `rmatrix`, `srmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (3);
```

```
srmatrix ma(3);
rmatrix mb(3,4);
rmatrix mx(3,4);
double dErr;
ma.randomize(-10., 10.);
mb.randomize(-10., 10.);
```

```
mx.solve_tran (ma, mb, dErr);
```

```
std::cout << ~ma * mx - mb << dErr << std::endl;
std::cout << ~mx * ma - ~mb;
```

prints

```
-1.776e-015 -8.882e-016 -3.553e-015 +4.441e-016
+0.000e+000 +3.553e-015 +1.243e-014 -8.882e-016
+0.000e+000 +3.553e-015 -1.066e-014 +0.000e+000
+4.357e-015
-1.776e-015 +0.000e+000 +0.000e+000
-8.882e-016 +3.553e-015 +3.553e-015
-3.553e-015 +1.243e-014 -1.066e-014
+4.441e-016 -8.882e-016 +0.000e+000
```

**2.6.41 solve\_lu**

Functions

```
rmatrix&
rmatrix::solve_lu (const srmatrix& mA, const srmatrix& mLU,
                  const int* pPivots, const rmatrix& mB, TR& dErr)
                  throw (cvmexception);
```

```
rmatrix&
rmatrix::solve_lu (const srmatrix& mA, const srmatrix& mLU,
                  const int* pPivots, const rmatrix& mB)
                  throw (cvmexception);
```

set calling matrix to be equal to solution  $X$  of matrix linear equation  $A * X = B$  where parameter `mA` is square matrix  $A$ , parameter `mLU` is **LU factorization** of the matrix  $A$ , parameter `pPivots` is array of pivot numbers created while factorizing the matrix  $A$  and parameter `mB` is the matrix  $B$ . Every function returns reference to the matrix changed. The first version also sets output parameter `dErr` to be equal to norm of computation error. These functions are useful when you need to solve few linear equations of kind  $AX = B$  with the same matrix  $A$  and different matrices  $B$ . In such case you save on matrix  $A$  factorization since it's needed to be performed just one time. Functions throw **cvmexception** in case of inappropriate sizes of the operands or when the matrix  $A$  is close to singular. See also **rvector::solve**, **rmatrix**, **srmatrix**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    double a[] = {1., -1., 1., 2., -2., 1., 3., -2., 1.};
    srmatrix ma(a,3);
    srmatrix mLU(3);
    rmatrix mb1(3,2);
    rmatrix mb2(3,2);
    rmatrix mx1(3,2);
    rmatrix mx2(3,2);
    iarray nPivots(3);
    double dErr = 0.;
    mb1.randomize(-1.,3.);
    mb2.randomize(2.,5.);

    mLU.low_up(ma, nPivots);
    std::cout << mx1.solve_lu (ma, mLU, nPivots, mb1, dErr);
    std::cout << dErr << std::endl;
    std::cout << mx2.solve_lu (ma, mLU, nPivots, mb2) << std::endl;
```

```
    std::cout << ma * mx1 - mb1 << std::endl << ma * mx2 - mb2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
3.85e+00 5.90e-01
-4.23e+00 -3.67e+00
2.10e+00 2.55e+00
7.04e-15
9.49e+00 8.93e+00
-1.00e+01 -1.42e+01
4.21e+00 7.55e+00

0.00e+00 0.00e+00
0.00e+00 0.00e+00
4.44e-16 -1.11e-16

4.44e-16 0.00e+00
-4.44e-16 0.00e+00
8.88e-16 0.00e+00
```

**2.6.42 svd**

Functions

rvector

`rmatrix::svd () const throw (cvmexception);`

rvector

`rmatrix::svd (srmatrix& mU, srmatrix& mVH) const throw (cvmexception);`

create object of type rvector as a vector of **singular values** of calling matrix. The second version sets output parameter mU to be equal to matrix U of size  $m \times m$  and mVH to be equal to matrix  $V^H$  of size  $n \times n$ . Functions throw **cvmexception** in case of inappropriate sizes of the operands or in case of convergence error. Use **rvector::svd** in order to avoid new vector creation. Function is *redefined* in the classes **srmatrix**, **srbmatrix**, **srsrmatrix**. See also **rvector**, **rmatrix**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    double m[] = {1., -1., 1., 2., -2., 1.,
                  3., -2., 1., 0., -2., 1.};
    rmatrix mA(m,4,3);
    rmatrix mSigma(4,3);
    rvector v;
    srmatrix mU(4), mVH(3);

    v << mA.svd(mU, mVH);
    mSigma.diag(0) = v;

    std::cout << mU << std::endl;
    std::cout << mVH << std::endl;
    std::cout << mSigma << std::endl;

    std::cout << (mA * ~mVH - mU * mSigma).norm() << std::endl;
    std::cout << (~mA * mU - ~(mSigma * mVH)).norm() << std::endl;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

-4.84e-01 1.95e-01 1.15e-02 -8.53e-01
2.17e-01 -3.41e-01 -8.89e-01 -2.13e-01
```

```
6.62e-01 7.16e-01 -6.18e-02 -2.13e-01  
-5.29e-01 5.78e-01 -4.53e-01 4.26e-01
```

```
-2.21e-01 8.54e-01 -4.72e-01  
9.59e-01 1.04e-01 -2.62e-01  
-1.75e-01 -5.11e-01 -8.42e-01
```

```
4.96e+00 0.00e+00 0.00e+00  
0.00e+00 2.51e+00 0.00e+00  
0.00e+00 0.00e+00 3.77e-01  
0.00e+00 0.00e+00 0.00e+00
```

```
1.37e-15  
2.48e-15
```



### 2.6.43 pinv

Functions

```
rmatrix rmatrix::pinv (TR threshold = cvmMachSp()) const
    throw (cvmexception);
rmatrix& rmatrix::pinv (const basic_rmatrix& mA,
    TR threshold = cvmMachSp())
    throw (cvmexception);
```

implement matrix pseudo inversion [6], p. 33 (or Moore-Penrose generalized inversion [5], p. 421). Strictly defined,  $n \times m$  matrix  $A^+$  is *pseudo inversion* of  $m \times n$  matrix  $A$  if the following two equations are satisfied:

$$AA^+A = A,$$

$$A^+ = QA^H = A^HP$$

where  $Q$  and  $P$  are some matrices. To compute the pseudo inversion, we use [Singular Value Decomposition \(SVD\)](#)

$$A = U\Sigma V^H$$

of matrix  $A$ , thus

$$A^+ = V\Sigma^{-1}U^H,$$

where  $\Sigma^{-1}$  is diagonal  $n \times m$  matrix having inverted diagonal values of matrix  $\Sigma$  if they are greater than some threshold, and zeros otherwise.

First version creates object of type `rmatrix` as pseudo inverted calling matrix (it throws `cvmexception` in case of memory allocation failure). Second function sets calling matrix to be equal to matrix `mA` pseudo inverted (it throws `cvmexception` in case of not appropriate sizes of the operands). The threshold parameter sets minimum distinguishable from zero singular value to be used to compute the pseudo inversion. All values equal or less than the threshold are treated as zeros. Functions are *inherited* in the classes `srmatrix` and `srsrmatrix` and *redefined* in `srbmatrix`. See also `rmatrix`. Example:

```
using namespace cvm;

try {
    rmatrix mA(3,4);
    mA(1,1) = 1.; mA(1,2) = -1.; mA(1,3) = 2.; mA(1,4) = 0.;
    mA(2,1) = -1.; mA(2,2) = 2.; mA(2,3) = -3.; mA(2,4) = 1.;
    mA(3,1) = 0.; mA(3,2) = 1.; mA(3,3) = -1.; mA(3,4) = 1.;
    rmatrix mX = mA.pinv(1.e-13);
    std::cout << mX << (mA * mX * mA - mA).norm2() << std::endl;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
+3.3333333e-001 -2.1510571e-016 +3.3333333e-001  
+1.1111111e-001 +1.1111111e-001 +2.2222222e-001  
+2.2222222e-001 -1.1111111e-001 +1.1111111e-001  
+4.4444444e-001 +1.1111111e-001 +5.5555556e-001  
+2.5460202e-015
```

Band matrix example:

```
using namespace cvm;
```

```
try {  
    srbmatrix mA (40, 1, 0);  
    mA.diag(0).randomize(-1.,1.);  
    mA.diag(-1).randomize(5.,10.);  
  
    srmatrix mX (40);  
    mX.pinv(mA);  
    std::cout << (mA * mX * mA - mA).norm2() << std::endl;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
+8.1956952e-14
```

**2.6.44 gels**

Functions

```

rmatrix  rmatrix::gels (bool transpose, const rmatrix& mB,
                       rvector& vErr) const throw (cvmexception);
rmatrix& rmatrix::gels (bool transpose,
                       const rmatrix& mA, const rmatrix& mB,
                       rvector& vErr) throw (cvmexception);
rvector  rmatrix::gels (bool transpose, const rvector& mB,
                       TR& dErr) const throw (cvmexception);

```

solve overdetermined or underdetermined linear systems

$$A * x = b$$

for  $m \times n$  matrix  $A$  (or its transpose) where  $b$  is a vector of length  $k$  or systems

$$A * X = B$$

for multiple vectors  $b$  stored as columns of  $k \times l$  matrix  $B$  where  $k = m$  in non-transposed case and  $k = n$  otherwise. The algorithm uses QR or LQ factorization of  $A$ . It is assumed that  $A$  has full rank, infinity returned otherwise. Internally functions use ?GELS LAPACK routines. If  $m > n$  and `transpose=false` or  $m < n$  and `transpose=true`, then the system is overdetermined, thus the algorithm tries to find the least squares solution  $x$  of the problem

$$\|A * x - b\|_2 \rightarrow \min \quad \text{or} \quad \|A^H * x - b\|_2 \rightarrow \min$$

respectively. Output vector `vErr` of length  $l$  (or real number `dErr` for single vector  $b$ ) returns residual sum of squares. The system is underdetermined otherwise, and the algorithm finds its minimum norm solution. In this case `vErr` (or `bErr`) is set to zero. In both cases the solution computed satisfies  $x = \text{pinv}(A) * b$ , but this algorithm is faster than pseudo inversion. Matrix  $A$  is passed as argument `mA` or (in second case) it's a calling object. First and third versions return the solution as new matrix or vector object. Second version sets calling object to be the solution. Functions throw `cvmexception` in case of inappropriate sizes of the operands. Functions are *inherited* in all real matrix classes. See also `rmatrix`, `rvector::gels`, `rmatrix::pinv`. Example:

```
using namespace cvm;
```

```

rmatrix a(7, 5);
rmatrix b(7, 2);
rmatrix bt(5, 2);
rvector bv(5);
rvector vErr(2);
treval dErr;

```

```
a.randomize(-10., 10.);
b.randomize(-10., 10.);
bt.randomize(-10., 10.);
bv.randomize(-10., 10.);

rmatrix x = a.gels(false, b, vErr);
rvector vt = a.gels(true, bv, dErr);
rmatrix xt(7,2);
xt.gels(true, a, bt, vErr);

std::cout << (a.pinv()*b - x).norm() << std::endl;
std::cout << (~a.pinv()*bv - vt).norm() << std::endl;
std::cout << (~a.pinv()*bt - xt).norm() << std::endl;

prints

+2.6295387e-015
+5.8515615e-015
+6.6462869e-015
```

**2.6.45 gelsy**

Functions

```

rmatrix  rmatrix::gelsy (const rmatrix& mB, int& rank,
                        TR tol = cvmMachSp()) const throw (cvmexception);
rmatrix& rmatrix::gelsy (const rmatrix& mA, const rmatrix& mB, int& rank,
                        TR tol = cvmMachSp()) throw (cvmexception);
rvector  rmatrix::gelsy (const rvector& mB, int& rank,
                        TR tol = cvmMachSp()) const throw (cvmexception);

```

compute the minimum-norm solution to linear least squares problem

$$\|A * x - b\|_2 \rightarrow \min$$

using complete orthogonal factorization of  $m \times n$  matrix  $A$ . Here  $b$  is a vector of length  $m$ . Multiple vectors  $b$  can be stored as columns of  $m \times l$  matrix  $B$ . Matrix  $A$  may be rank-deficient, functions return its effective rank in `rank` output parameter using `tol` tolerance. Internally functions use ?GELSY LAPACK routines, see more details about the algorithm in those routines' documentation. Matrix  $A$  is passed as argument `mA` or (in second case) it's a calling object. First and third versions return the solution as new matrix or vector object. Second version sets calling object to be the solution. Functions throw `cvmexception` in case of inappropriate sizes of the operands. Functions are *inherited* in all real matrix classes. See also `rmatrix`, `rvector::gelsy`, `rmatrix::gelss`, `rmatrix::gelsd`. Example:

```
using namespace cvm;
```

```

rmatrix a(4, 5);
rmatrix b(4, 2);
rvector bv(4);
a.randomize(-10., 10.);
b.randomize(-10., 10.);
bv.randomize(-10., 10.);
tint rank;

```

```

rmatrix x = a.gelsy(b, rank);
rvector xv = a.gelsy(bv, rank);
rmatrix x2(5,2);
x2.gelsy(a, b, rank);

```

```

std::cout << (a*x - b).norm() << std::endl;
std::cout << (a*xv - bv).norm() << std::endl;
std::cout << (a*x2 - b).norm() << std::endl;
std::cout << rank << " " << a.rank() << std::endl;

```

```
prints
```

```
+3.3539762e-014  
+5.2545344e-015  
+3.3539762e-014  
+4 +4
```

**2.6.46 gelss**

Functions

```

rmatrix  rmatrix::gelss (const rmatrix& mB, int& rank, rvector& sv,
                        TR tol = cvmMachSp()) const throw (cvmexception);
rmatrix& rmatrix::gelss (const rmatrix& mA, const rmatrix& mB,
                        rvector& sv, int& rank, TR tol = cvmMachSp())
                        throw (cvmexception);
rvector  rmatrix::gelss (const rvector& mB, rvector& sv, int& rank,
                        TR tol = cvmMachSp()) const throw (cvmexception);

```

compute minimum-norm solution to linear least squares problem

$$\|A * x - b\|_2 \rightarrow \min$$

using singular value decomposition of  $m \times n$  matrix  $A$ . Here  $b$  is a vector of length  $m$ . Multiple vectors  $b$  can be stored as columns of  $m \times l$  matrix  $B$ . Matrix  $A$  may be rank-deficient, functions return its effective rank in `rank` output parameter using `tol` tolerance. These functions also compute singular values of  $A$  in decreasing order and return them in `sv` output parameter having  $\min(m, n)$  size. Internally functions use ?GELSS LAPACK routines, see more details about the algorithm in those routines' documentation. Matrix  $A$  is passed as argument `mA` or (in second case) it's a calling object. First and third versions return the solution as new matrix or vector object. Second version sets calling object to be the solution. Functions throw `cvmexception` in case of inappropriate sizes of the operands. Functions are *inherited* in all real matrix classes. See also `rmatrix`, `rvector::gelss`, `rmatrix::gelsy`, `rmatrix::gelsd`. Example:

```

using namespace cvm;

rmatrix a(4, 5);
rmatrix b(4, 2);
rvector bv(4);
rvector sv(4);
tint rank;
a.randomize(-10., 10.);
b.randomize(-10., 10.);
bv.randomize(-10., 10.);

rmatrix x = a.gelss(b, sv, rank);
rvector xv = a.gelss(bv, sv, rank);
rmatrix xt(5,2);
xt.gelss(a, b, sv, rank);

std::cout << (a*x - b).norm() << std::endl;

```

```
std::cout << (a*xv - bv).norm() << std::endl;  
std::cout << (a*xt - b).norm() << std::endl;  
std::cout << (sv - a.svd()).norm() << std::endl;  
std::cout << rank << " " << a.rank() << std::endl;
```

prints

```
+1.9169279e-014  
+8.9260826e-015  
+1.9169279e-014  
+8.1402897e-015  
+4 +4
```



**2.6.47 gelsd**

Functions

```

rmatrix  rmatrix::gelsd (const rmatrix& mB, int& rank, rvector& sv,
                        TR tol = cvmMachSp()) const throw (cvmexception);
rmatrix& rmatrix::gelsd (const rmatrix& mA, const rmatrix& mB,
                        rvector& sv, int& rank, TR tol = cvmMachSp())
                        throw (cvmexception);
rvector  rmatrix::gelsd (const rvector& mB, rvector& sv, int& rank,
                        TR tol = cvmMachSp()) const throw (cvmexception);

```

compute minimum-norm solution to linear least squares problem

$$\|A * x - b\|_2 \rightarrow \min$$

using singular value decomposition of  $m \times n$  matrix  $A$  and divide and conquer method. Here  $b$  is a vector of length  $m$ . Multiple vectors  $b$  can be stored as columns of  $m \times l$  matrix  $B$ . Matrix  $A$  may be rank-deficient, functions return its effective rank in `rank` output parameter using `tol` tolerance. These functions also compute singular values of  $A$  in decreasing order and return them in `sv` output parameter having  $\min(m, n)$  size. Internally functions use ?GELSD LAPACK routines, see more details about the algorithm in those routines' documentation. Matrix  $A$  is passed as argument `mA` or (in second case) it's a calling object. First and third versions return the solution as new matrix or vector object. Second version sets calling object to be the solution. Functions throw `cvmexception` in case of inappropriate sizes of the operands. Functions are *inherited* in all real matrix classes. See also `rmatrix`, `rvector::gelsd`, `rmatrix::gelsy`, `rmatrix::gelss`. Example:

```

using namespace cvm;

rmatrix a(4, 5);
rmatrix b(4, 2);
rvector bv(4);
rvector sv(4);
tint rank;
a.randomize(-10., 10.);
b.randomize(-10., 10.);
bv.randomize(-10., 10.);

rmatrix x = a.gelsd(b, sv, rank);
rvector xv = a.gelsd(bv, sv, rank);
rmatrix xt(5,2);
xt.gelsd(a, b, sv, rank);

std::cout << (a*x - b).norm() << std::endl;

```

```
std::cout << (a*xv - bv).norm() << std::endl;  
std::cout << (a*xt - b).norm() << std::endl;  
std::cout << (sv - a.svd()).norm() << std::endl;  
std::cout << rank << " " << a.rank() << std::endl;
```

prints

```
+1.0341178e-014  
+6.0443462e-015  
+1.0341178e-014  
+8.9260826e-015  
+4 +4
```

### 2.6.48 rank

Function

```
int rmatrix::rank (TR eps = cvmMachSp ()) const throw (cvmexception);
```

returns rank of calling matrix as number of **singular values** with **normalized** absolute value greater than or equal to parameter `eps` (this is the **largest relative spacing** by default). Function throws **cvmexception** in case of convergence error. Function is *inherited* in the classes **srmatrix**, **srbmatrix**, **srsmatrix**. See also **rmatrix**. Example:

```
using namespace cvm;
try {
    double a[] = {1., 2., 3., 4., 5., 6.,
                  7., 8., 9., 10., 11., 12.};
    rmatrix m(a,3,4);

    std::cout << m << m.rank() << std::endl;
    m(3,4) = 13.;
    std::cout << m.rank() << std::endl;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
1 4 7 10
2 5 8 11
3 6 9 12
2
3
```

**2.6.49 ger**

Function

```
rmatrix&
rmatrix::ger (TR dAlpha, const rvector& vCol, const rvector& vRow)
throw (cvmexception);
```

calls one of ?GER routines of the [BLAS library](#) performing **rank-1 update** matrix-vector operation defined as

$$M = \alpha \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} (y_1 \ y_2 \ \cdots \ y_n) + M,$$

where  $\alpha$  is real number (parameter dAlpha), M is calling matrix and x and y are real vectors (parameters vCol and vRow respectively). Function returns reference to the matrix changed and throws **cvmexception** in case of inappropriate sizes of the operands. Function is *inherited* in the class **srmatrix** and *not applicable* to objects of the classes **srbmatrix** and **srsmatrix** (i.e. **cvmexception** would be thrown in case of using it for objects of those classes). See also **rvector**, **rmatrix**. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (4);
try {
    double alpha = 1.3;
    rmatrix m(3,4);
    rvector vc(3);
    rvector vr(4);
    m.randomize(-1., 2.); vc.randomize(-1., 3.); vr.randomize(0., 2.);

    std::cout << m + vc.rank1update (vr) * alpha << std::endl;
    std::cout << m.ger(alpha, vc, vr);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
-1.7127e-01 2.9410e+00 1.3449e+00 3.6055e+00
1.9057e+00 2.6726e+00 1.7134e+00 2.2154e+00
1.7217e-01 1.3508e+00 8.8949e-01 2.2551e+00
```

```
-1.7127e-01 2.9410e+00 1.3449e+00 3.6055e+00  
1.9057e+00 2.6726e+00 1.7134e+00 2.2154e+00  
1.7217e-01 1.3508e+00 8.8949e-01 2.2551e+00
```

**2.6.50 gemm**

Function

```
rmatrix& rmatrix::gemm (const rmatrix& m1, bool bTrans1,
                       const rmatrix& m2, bool bTrans2,
                       TR dAlpha, TR dBeta) throw (cvmexception);
```

calls one of ?GEMM routines of the [BLAS library](#) performing matrix-matrix operation defined as

$$M = \alpha \mathcal{T}(M_1) \cdot \mathcal{T}(M_2) + \beta M,$$

where  $\alpha$  and  $\beta$  are real numbers (parameters `dAlpha` and `dBeta`),  $M$  is calling matrix and  $M_1$  and  $M_2$  are matrices (parameters `m1` and `m2` respectively). Function  $\mathcal{T}(M_i)$  transposes matrix  $M_i$  if appropriate boolean parameter `bTrans*` is equal to `true` and does nothing otherwise. Function returns reference to the matrix changed and throws `cvmexception` in case of inappropriate sizes of the operands. Function is *inherited* in the class `rmatrix` and *not applicable* to objects of the classes `srbmatrix` and `srsmatrix` (i.e. exception of type `cvmexception` would be thrown in case of using it for objects of those classes). See also `rmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (4);
try {
    double alpha = 1.3;
    double beta = -0.7;
    rmatrix m1(4,3); rmatrix m2(4,3);
    rmatrix m(3,3);
    m.randomize(-1., 2.); m1.randomize(-1., 3.); m2.randomize(0., 2.);
    std::cout << ~m1 * m2 * alpha + m * beta << std::endl;
    std::cout << m.gemm(m1, true, m2, false, alpha, beta);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
5.0504e+00 6.8736e+00 3.1171e+00
2.3915e+00 2.2544e+00 3.9205e+00
3.4607e+00 3.5351e+00 4.8622e+00

5.0504e+00 6.8736e+00 3.1171e+00
2.3915e+00 2.2544e+00 3.9205e+00
3.4607e+00 3.5351e+00 4.8622e+00
```

**2.6.51 symm**

Function

```
rmatrix& rmatrix::symm (bool bLeft, const srsmatrix& ms,
                       const rmatrix& m, TR dAlpha, TR dBeta)
                       throw (cvmexception);
```

calls one of ?SYMM routines of the [BLAS library](#) performing one of matrix-matrix operations defined as

$$M = \alpha M_s \cdot M_1 + \beta M \quad \text{or} \quad M = \alpha M_1 \cdot M_s + \beta M,$$

where  $\alpha$  and  $\beta$  are real numbers (parameters dAlpha and dBeta),  $M$  is calling matrix,  $M_s$  is symmetric matrix and  $M_1$  is real matrix (parameters ms and m respectively). First operation is performed if bLeft passed is true and second one otherwise. Function returns reference to the matrix changed and throws [cvmexception](#) in case of inappropriate sizes of the operands. Function is *inherited* in the classes [srmatrix](#) and [srsmatrix](#) and *not applicable* to objects of the class [srbmatrix](#) (i.e. [cvmexception](#) would be thrown in case of using it for objects of that class). See also [srsmatrix](#), [rmatrix](#). Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (4);
try {
    double alpha = 1.3;
    double beta = -0.7;
    rmatrix m1(3,4);
    rmatrix m2(4,3);
    srsmatrix ms(3);
    rmatrix m(3,4);
    m.randomize(-1., 2.); m1.randomize(-1., 3.); m2.randomize(0., 2.);
    ms.randomize(-3., 1.);

    std::cout << ms * m1 * alpha + m * beta << std::endl;
    std::cout << m.symm (true, ms, m1, alpha, beta) << std::endl;

    m.resize(4,3);
    std::cout << m2 * ms * alpha + m * beta << std::endl;
    std::cout << m.symm (false, ms, m2, alpha, beta);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
-3.3733e+00 -5.0566e+00 -6.3018e+00 -5.4907e+00  
-1.8629e+00 -1.5133e+00 -1.1372e+00 -2.5557e+00  
-3.5695e+00 -1.0012e+01 -1.4239e+00 -6.1786e-01
```

```
-3.3733e+00 -5.0566e+00 -6.3018e+00 -5.4907e+00  
-1.8629e+00 -1.5133e+00 -1.1372e+00 -2.5557e+00  
-3.5695e+00 -1.0012e+01 -1.4239e+00 -6.1786e-01
```

```
-6.4072e+00 7.0534e-01 1.5349e+00  
-4.8219e+00 -6.9891e+00 -5.1766e+00  
6.8503e-01 3.5828e+00 -3.2174e+00  
2.3469e-01 -9.3921e-01 -2.1961e+00
```

```
-6.4072e+00 7.0534e-01 1.5349e+00  
-4.8219e+00 -6.9891e+00 -5.1766e+00  
6.8503e-01 3.5828e+00 -3.2174e+00  
2.3469e-01 -9.3921e-01 -2.1961e+00
```



**2.6.52 qr**

Functions

```
void rmatrix::qr (rmatrix& mQ, srmatrix& mR) const throw (cvmexception);
void rmatrix::qr (srmatrix& mQ, rmatrix& mR) const throw (cvmexception);
```

compute QR factorization as

$$M = QR$$

where  $M$  is calling matrix, orthogonal matrix  $Q$  and upper triangular (trapezoidal) matrix  $R$  are  $mQ$  and  $mR$  respectively. First version implements so-called "economy" algorithm which for given  $m \times n$  matrix  $M$  computes  $m \times n$  matrix  $Q$  and  $n \times n$  matrix  $R$ . Second version is "full" mode one computing  $m \times m$  matrix  $Q$  and  $m \times n$  matrix  $R$ . Functions throw `cvmexception` in case if inappropriate sizes of the operands passed. Functions are *redefined* in the class `srmatrix`. See also `rmatrix`, `srmatrix`. Example:

```
using namespace cvm;

treal a[] = {1., 2., 3., 4., 5., 6.};
const cvm::rmatrix mh(a, 2, 3);
const cvm::rmatrix mv(a, 3, 2);
cvm::srmatrix s2(2), s3(3);
cvm::rmatrix h(2,3), v(3,2);

mh.qr(h,s3);
std::cout << (eye_real(2) - ~rmatrix(h,1,1,2,2)*rmatrix(h,1,1,2,2)).norm()
           << " " << (mh - h * s3).norm() << std::endl;
mh.qr(s2,h);
std::cout << (eye_real(2) - ~s2 * s2).norm()
           << " " << (mh - s2 * h).norm() << std::endl;
mv.qr(v,s2);
std::cout << (eye_real(2) - ~v * v).norm()
           << " " << (mv - v * s2).norm() << std::endl;
mv.qr(s3,v);
std::cout << (eye_real(3) - ~s3 * s3).norm()
           << " " << (mv - s3 * v).norm() << std::endl;

prints

+4.6933177e-16 +2.2342807e-15
+4.6933177e-16 +2.2342807e-15
+5.1302953e-16 +1.4432899e-15
+5.2889959e-16 +1.4432899e-15
```

**2.6.53 lq**

Functions

```
void rmatrix::lq (srmatrix& mL, rmatrix& mQ) const throw (cvmexception);
void rmatrix::lq (rmatrix& mL, srmatrix& mQ) const throw (cvmexception);
```

compute LQ factorization as

$$M = LQ$$

where  $M$  is calling matrix, lower triangular (trapezoidal) matrix  $L$  and orthogonal matrix  $Q$  are  $mL$  and  $mQ$  respectively. First version implements so-called "economy" algorithm which for given  $m \times n$  matrix  $M$  computes  $m \times m$  matrix  $L$  and  $m \times n$  matrix  $Q$ . Second version is "full" mode one computing  $m \times n$  matrix  $L$  and  $n \times n$  matrix  $Q$ . Functions throw `cvmexception` in case of inappropriate sizes of the operands passed. Functions are *redefined* in the class `srmatrix`. See also `rmatrix`, `srmatrix`. Example:

```
using namespace cvm;

treal a[] = {1., 2., 3., 4., 5., 6.};
const cvm::rmatrix mh(a, 2, 3);
const cvm::rmatrix mv(a, 3, 2);
cvm::srmatrix s2(2), s3(3);
cvm::rmatrix h(2,3), v(3,2);

mh.lq(s2,h);
std::cout << (eye_real(2) - h * ~h).norm() << " " << (mh - s2 * h)
           .norm() << std::endl;
mv.lq(s3,v);
std::cout << (eye_real(2) - ~rmatrix(v,1,1,2,2) * rmatrix(v,1,1,2,2))
           .norm() << " " << (mv - s3 * v).norm() << std::endl;
mh.lq(h,s3);
std::cout << (eye_real(3) - s3 * ~s3).norm() << " " << (mh - h * s3)
           .norm() << std::endl;
mv.lq(v,s2);
std::cout << (eye_real(2) - s2 * ~s2).norm() << " " << (mv - v * s2)
           .norm() << std::endl;

prints

+8.6355085e-016 +3.3893638e-015
+1.9229627e-016 +1.6011864e-015
+1.0030483e-015 +3.3893638e-015
+1.9229627e-016 +1.6011864e-015
```

**2.6.54 rq**

Functions

```
void rmatrix::rq (srmatrix& mR, rmatrix& mQ) const throw (cvmexception);
void rmatrix::rq (rmatrix& mR, srmatrix& mQ) const throw (cvmexception);
```

compute RQ factorization as

$$M = RQ$$

where  $M$  is calling matrix, upper triangular matrix  $R$  and orthogonal matrix  $Q$  are  $mR$  and  $mQ$  respectively. First version implements so-called "economy" algorithm which for given  $m \times n$  matrix  $M$  computes  $m \times m$  matrix  $R$  and  $m \times n$  matrix  $Q$ . Second version is "full" mode one computing  $m \times n$  matrix  $R$  and  $n \times n$  matrix  $Q$ . Following [this definition](#) the implementation assumes that  $m \leq n$  and throws `cvmexception` otherwise. Functions also throw `cvmexception` in case of inappropriate sizes of the operands passed. Functions are *redefined* in the class `srmatrix`. See also `rmatrix`, `srmatrix`. Example:

```
using namespace cvm;
```

```
treal a[] = {1., 2., 3., 4., 5., 6.};
const cvm::rmatrix mh(a, 2, 3);
const cvm::rmatrix mv(a, 3, 2);
cvm::srmatrix s2(2), s3(3);
cvm::rmatrix h(2,3), v(3,2);

mh.rq(h,s3);
std::cout << (eye_real(2) - rmatrix(s3,2,1,2,3) * ~rmatrix(s3,2,1,2,3))
           .norm() << " " << (mh - h * s3).norm() << std::endl;

mh.rq(s2,h);
std::cout << (eye_real(2) - h * ~h).norm()
           << " " << (mh - s2 * h).norm() << std::endl;
```

prints

```
+5.5511151e-016 +2.0471501e-015
+5.5511151e-016 +2.0471501e-015
```

## 2.6.55 ql

Functions

```
void rmatrix::ql (rmatrix& mQ, srmatrix& mL) const throw (cvmexception);
void rmatrix::ql (srmatrix& mQ, rmatrix& mL) const throw (cvmexception);
```

compute QL factorization as

$$M = QL$$

where  $M$  is calling matrix, orthogonal matrix  $Q$  and lower triangular matrix  $L$  are  $mQ$  and  $mL$  respectively. First version implements so-called "economy" algorithm which for given  $m \times n$  matrix  $M$  computes  $m \times n$  matrix  $Q$  and  $n \times n$  matrix  $L$ . Second version is "full" mode one computing  $m \times m$  matrix  $Q$  and  $m \times n$  matrix  $L$ . Following [this definition](#) the implementation assumes that  $m \geq n$  and throws `cvmexception` otherwise. Functions also throw `cvmexception` in case of inappropriate sizes of the operands passed. Functions are *redefined* in the class `srmatrix`. See also `rmatrix`, `srmatrix`. Example:

```
using namespace cvm;
```

```
treal a[] = {1., 2., 3., 4., 5., 6.};
const cvm::rmatrix mv(a, 3, 2);
cvm::srmatrix s2(2), s3(3);
cvm::rmatrix v(3,2);
```

```
mv.ql(v,s2);
std::cout << (eye_real(2) - ~v * v).norm()
           << " " << (mv - v * s2).norm() << std::endl;
```

```
mv.ql(s3,v);
std::cout << (eye_real(2) - ~rmatrix(s3,1,2,3,2) * rmatrix(s3,1,2,3,2))
           .norm() << " " << (mv - s3 * v).norm() << std::endl;
```

prints

```
+0.00000000e+000 +2.0947646e-015
+0.00000000e+000 +2.0947646e-015
```

### 2.6.56 **vanish**

Function

```
rmatrix& rmatrix::vanish();
```

sets every element of calling matrix to be equal to zero and returns reference to the matrix changed. This function is faster than, for example, `rmatrix::set(TR)` with zero operand passed. Function is *redefined* in the classes `srmatrix`, `srsmatrix`, `srbmatrix`. See also `rmatrix`. Example:

```
using namespace cvm;
```

```
rmatrix m(3, 4);  
m.randomize(0.,1.);
```

```
std::cout << m << std::endl;  
std::cout << m.vanish ();
```

prints

```
0.856532 0.938261 0.275704 0.186834  
0.651173 0.812159 0.100467 0.536912  
0.0726646 0.695914 0.661824 0.554613
```

```
0 0 0 0  
0 0 0 0  
0 0 0 0
```

### 2.6.57 **randomize**

Function

```
rmatrix& rmatrix::randomize (TR dFrom, TR dTo);
```

fills calling matrix with pseudo-random numbers distributed between dFrom and dTo. Function returns reference to the matrix changed. See also [rmatrix](#). Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (7);

rmatrix m(3,4);
m.randomize(-2.,3.);
std::cout << m;

prints

9.6853542e-01 2.7761467e+00 2.3791009e+00 -3.4452345e-01
2.9029511e+00 -9.5519883e-01 -4.9131748e-01 -1.2561113e+00
1.5219886e+00 -1.4494461e+00 2.8193304e+00 4.8817408e-01
```

## 2.7 *cmatrix*

This is end-user class encapsulating matrix of complex numbers.

```

template <typename TR, typename TC>
class cmatrix : public Matrix <TR,TC> {
public:
    cmatrix ();
    cmatrix (int nM, int nN);
    cmatrix (TC* pD, int nM, int nN);
    cmatrix (const TC* pD, int nM, int nN);
    cmatrix (const cmatrix& m);
    explicit cmatrix (const cvector& v, bool bBeColumn = true);
    explicit cmatrix (const rmatrix& m, bool bRealPart = true);
    cmatrix (const TR* pRe, const TR* pIm, int nM, int nN);
    cmatrix (const rmatrix& mRe, const rmatrix& mIm);
    cmatrix (cmatrix& m, int nRow, int nCol, int nHeight, int nWidth);
    TC& operator () (int im, int in) throw (cvmexception);
    TC operator () (int im, int in) const throw (cvmexception);
    cvector operator () (int i) throw (cvmexception);
    const cvector operator () (int i) const throw (cvmexception);
    cvector operator [] (int i) throw (cvmexception);
    const cvector operator [] (int i) const throw (cvmexception);
    cvector diag (int i) throw (cvmexception);
    const cvector diag (int i) const throw (cvmexception);
    const rmatrix real () const;
    const rmatrix imag () const;
    cmatrix& operator = (const cmatrix& m) throw (cvmexception);
    cmatrix& assign (const cvector& v) throw (cvmexception);
    cmatrix& assign (const TC* pD);
    rmatrix& assign (int nRow, int nCol, const cmatrix& m)
        throw (cvmexception);
    cmatrix& set (TC z);
    cmatrix& assign_real (const rmatrix& mRe) throw (cvmexception);
    cmatrix& assign_imag (const rmatrix& mIm) throw (cvmexception);
    cmatrix& set_real (TR d);
    cmatrix& set_imag (TR d);
    cmatrix& resize (int nNewM, int nNewN) throw (cvmexception);
    bool operator == (const cmatrix& v) const;
    bool operator != (const cmatrix& v) const;
    cmatrix& operator << (const cmatrix& m) throw (cvmexception);
    cmatrix operator + (const cmatrix& m) const throw (cvmexception);
    cmatrix operator - (const cmatrix& m) const throw (cvmexception);

```

```

cmatrix& sum (const cmatrix& m1,
             const cmatrix& m2) throw (cvmexception);
cmatrix& diff (const cmatrix& m1,
             const cmatrix& m2) throw (cvmexception);
cmatrix& operator += (const cmatrix& m) throw (cvmexception);
cmatrix& operator -= (const cmatrix& m) throw (cvmexception);
cmatrix operator - () const;
cmatrix operator * (TR d) const;
cmatrix operator / (TR d) const throw (cvmexception);
cmatrix operator * (TC z) const;
cmatrix operator / (TC z) const throw (cvmexception);
cmatrix& operator *= (TR d);
cmatrix& operator /= (TR d) throw (cvmexception);
cmatrix& operator *= (TC z);
cmatrix& operator /= (TC z) throw (cvmexception);
cmatrix& normalize ();
cmatrix operator ~() const throw (cvmexception);
cmatrix operator !() const throw (cvmexception);
cmatrix& conj (const cmatrix& m) throw (cvmexception);
cmatrix& conj () throw (cvmexception);
cmatrix& transpose (const cmatrix& m) throw (cvmexception);
cmatrix& transpose () throw (cvmexception);
cvector operator * (const cvector& v) const
    throw (cvmexception);
cmatrix operator * (const cmatrix& m) const
    throw (cvmexception);
cmatrix& mult (const cmatrix& m1, const cmatrix& m2)
    throw (cvmexception);
cmatrix& ranklupdate_u (const rvector& vCol,
                    const rvector& vRow) throw (cvmexception);
cmatrix& ranklupdate_c (const rvector& vCol,
                    const rvector& vRow) throw (cvmexception);
cmatrix& swap_rows (int n1, int n2) throw (cvmexception);
cmatrix& swap_cols (int n1, int n2) throw (cvmexception);
cmatrix& solve (const scmatrix& mA,
             const cmatrix& mB, TR& dErr)
    throw (cvmexception);
cmatrix& solve (const scmatrix& mA,
             const cmatrix& mB) throw (cvmexception);
cmatrix& solve_tran (const scmatrix& mA,
                 const cmatrix& mB, TR& dErr)
    throw (cvmexception);
cmatrix& solve_tran (const scmatrix& mA,

```



```

        const cmatrix& mB) throw (cvmexception);
cmatrix& solve_conj (const scmatrix& mA,
        const cmatrix& mB, TR& dErr)
        throw (cvmexception);
cmatrix& solve_conj (const scmatrix& mA,
        const cmatrix& mB) throw (cvmexception);
cmatrix& solve_lu (const scmatrix& mA, const scmatrix& mLU,
        const int* pPivots, const cmatrix& mB, TR& dErr)
        throw (cvmexception);
cmatrix& solve_lu (const scmatrix& mA, const scmatrix& mLU,
        const int* pPivots, const cmatrix& mB)
        throw (cvmexception);
rvector svd () throw (cvmexception);
rvector svd (cmatrix& mLSingVect, cmatrix& mRSingVect)
        throw (cvmexception);
cmatrix pinv (TR threshold = cvmMachSp()) const
        throw (cvmexception);
cmatrix& pinv (const cmatrix& mA,
        TR threshold = cvmMachSp())
        throw (cvmexception);
int rank (TR eps = cvmMachSp()) const throw (cvmexception);
cmatrix gels (bool conjugate, const cmatrix& mB,
        cvector& vErr) const throw (cvmexception);
cmatrix& gels (bool conjugate, const cmatrix& mA, const cmatrix& mB,
        cvector& vErr) throw (cvmexception);
cvector gels (bool conjugate, const cvector& mB,
        TC& dErr) const throw (cvmexception);
cmatrix gelsy (const cmatrix& mB, int& rank,
        TR tol = cvmMachSp()) const throw (cvmexception);
cmatrix& gelsy (const cmatrix& mA, const cmatrix& mB, int& rank,
        TR tol = cvmMachSp()) throw (cvmexception);
cvector gelsy (const cvector& mB, int& rank,
        TR tol = cvmMachSp()) const throw (cvmexception);
rmatrix gelss (const cmatrix& mB, rvector& sv,
        int& rank, TR tol = cvmMachSp()) const
        throw (cvmexception);
rmatrix& gelss (const cmatrix& mA, const cmatrix& mB, rvector& sv,
        int& rank, TR tol = cvmMachSp())
        throw (cvmexception);
rvector gelss (const cvector& mB, rvector& sv,
        int& rank, TR tol = cvmMachSp()) const
        throw (cvmexception);
rmatrix gelsd (const cmatrix& mB, rvector& sv,

```

```

        int& rank, TR tol = cvmMachSp()) const
        throw (cvmexception);
rmatrix& gelsd (const cmatrix& mA, const cmatrix& mB, rvector& sv,
        int& rank, TR tol = cvmMachSp())
        throw (cvmexception);
rvector gelsd (const cvector& mB, rvector& sv,
        int& rank, TR tol = cvmMachSp()) const
        throw (cvmexception);
void qr(cmatrix& mQ, scmatrix& mR) const throw (cvmexception);
void qr(scmatrix& mQ, cmatrix& mR) const throw (cvmexception);
void lq(scmatrix& mL, cmatrix& mQ) const throw (cvmexception);
void lq(cmatrix& mL, scmatrix& mQ) const throw (cvmexception);
void rq(scmatrix& mR, cmatrix& mQ) const throw (cvmexception);
void rq(cmatrix& mR, scmatrix& mQ) const throw (cvmexception);
void ql(cmatrix& mQ, scmatrix& mL) const throw (cvmexception);
void ql(scmatrix& mQ, cmatrix& mL) const throw (cvmexception);
cmatrix& vanish ();
cmatrix& geru (TC alpha, const cvector& vCol,
        const cvector& vRow) throw (cvmexception);
cmatrix& gerc (TC alpha, const cvector& vCol,
        const cvector& vRow) throw (cvmexception);
cmatrix& gemm (const cmatrix& m1, bool bTrans1,
        const cmatrix& m2, bool bTrans2,
        TC dAlpha, TC dBeta) throw (cvmexception);
cmatrix& hemm (bool bLeft, const schmatrix& ms, const cmatrix& m,
        TC dAlpha, TC dBeta) throw (cvmexception);
cmatrix& randomize_real (TR dFrom, TR dTo);
cmatrix& randomize_imag (TR dFrom, TR dTo);
};

```

### 2.7.1 `cmatrix ()`

Constructor

```
cmatrix::cmatrix ();
```

creates empty `cmatrix` object. See also `cmatrix`. Example:

```
using namespace cvm;
```

```
cmatrix m;  
std::cout << m.msize() << std::endl  
           << m.nsize() << std::endl;
```

```
m << eye_complex(3);  
std::cout << m;
```

prints

```
0  
0  
(1,0) (0,0) (0,0)  
(0,0) (1,0) (0,0)  
(0,0) (0,0) (1,0)
```

### 2.7.2 `cmatrix (int,int)`

Constructor

```
cmatrix::cmatrix (int nM, int nN);
```

creates  $m \times n$  `cmatrix` object where  $m$  is passed in `nM` parameter (number of rows) and  $n$  is passed in `nN` (number of columns). Constructor throws `cvmexception` in case of non-positive sizes passed or memory allocation failure. See also `cmatrix`. Example:

```
using namespace cvm;
```

```
cmatrix m (3, 4);  
std::cout << m.msize() << std::endl  
          << m.nsize() << std::endl  
          << m.size() << std::endl << m;
```

prints

```
3  
4  
12  
(0,0) (0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0) (0,0)
```

**2.7.3 *cmatrix* (TC\*,int,int)**

Constructor

```
cmatrix::cmatrix (TC* pD, int nM, int nN);
```

creates  $m \times n$  *cmatrix* object where  $m$  is passed in  $nM$  parameter (number of rows) and  $n$  is passed in  $nN$  (number of columns). Unlike others, this constructor *does not allocate memory*. It just shares memory with array pointed to by  $pD$ . Constructor throws *cvmexception* in case of non-positive size passed. See also *cmatrix*, *cmatrix (const TC\*,int,int)*. Example:

```
using namespace cvm;
double a[] = {5., 5., 5., 5., 5., 5.,
              5., 5., 5., 5., 5., 5.};
cmatrix m ((std::complex<double>*) a, 2, 3);

m(1,1) = std::complex<double>(1.,2.);
std::cout << m << std::endl;
std::cout << a[0] << " " << a[1] << " "
          << a[2] << " " << a[3] << std::endl;
```

prints

```
(1,2) (5,5) (5,5)
(5,5) (5,5) (5,5)
```

```
1 2 5 5
```

**2.7.4 *cmatrix* (const TC\*,int,int)**

Constructor

```
cmatrix::cmatrix (const TC* pD, int nM, int nN);
```

creates  $m \times n$  *cmatrix* object where  $m$  is passed in  $nM$  parameter (number of rows) and  $n$  is passed in  $nN$  (number of columns) and copies  $m * n$  elements of array  $pD$  to it by **columns**. Constructor throws ***cvmexception*** in case of non-positive size passed. See also ***cmatrix***, ***cmatrix* (TC\*,int,int)**. Example:

```
using namespace cvm;
const double a[] = {5., 5., 5., 5., 5., 5.,
                   5., 5., 5., 5., 5., 5.};
cmatrix m ((const std::complex<double>*) a, 2, 3);

m(1,1) = std::complex<double>(1.,2.);
std::cout << m << std::endl;
std::cout << a[0] << " " << a[1] << " "
          << a[2] << " " << a[3] << std::endl;
```

prints

```
(1,2) (5,5) (5,5)
(5,5) (5,5) (5,5)
```

```
5 5 5 5
```

### 2.7.5 `cmatrix (const cmatrix&)`

Copy constructor

```
cmatrix::cmatrix (const cmatrix& m);
```

creates `cmatrix` object as a copy of `m`. Constructor throws `cvmexception` in case of memory allocation failure. See also `cmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6.,  
             7., 8., 9., 10., 11., 12.};  
cmatrix m ((std::complex<double>*) a, 2, 3);  
cmatrix mc(m);
```

```
m(1,1) = std::complex<double>(7.77,7.77);  
std::cout << m << std::endl << mc;
```

prints

```
(7.77,7.77) (5,6) (9,10)  
(3,4) (7,8) (11,12)
```

```
(1,2) (5,6) (9,10)  
(3,4) (7,8) (11,12)
```

### 2.7.6 `cmatrix (const cvector&,bool)`

Constructor

```
explicit cmatrix::cmatrix (const cvector& v, bool bBeColumn = true);
```

creates `cmatrix` object containing `v.size()` rows and 1 column if `bBeColumn` is true or 1 row and `v.size()` columns otherwise. After that it copies vector `v`'s elements to the matrix created. Constructor throws `cvmexception` in case of memory allocation failure. See also `cmatrix`, `cvector`. Example:

```
using namespace cvm;
```

```
cvector v(3);  
v(1) = std::complex<double>(1.,2.);  
v(2) = std::complex<double>(2.,3.);  
v(3) = std::complex<double>(3.,4.);
```

```
cmatrix mc (v);  
cmatrix mr (v, false);
```

```
std::cout << mc << std::endl << mr;
```

prints

```
(1,2)
```

```
(2,3)
```

```
(3,4)
```

```
(1,2) (2,3) (3,4)
```



### 2.7.7 `cmatrix (const rmatrix&,bool)`

Constructor

```
explicit cmatrix::cmatrix (const rmatrix& m, bool bRealPart = true);
```

creates `cmatrix` object containing `m.msize()` rows and `m.nsize()` columns and copies matrix `m` to its real part if `bRealPart` is true or to its imaginary part otherwise. Constructor throws `cvmexception` in case of memory allocation failure. See also `cmatrix`, `rmatrix`.  
Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6.};  
const rmatrix m (a, 2, 3);  
cmatrix mr(m), mi(m, false);
```

```
std::cout << mr << std::endl << mi;
```

prints

```
(1,0) (3,0) (5,0)  
(2,0) (4,0) (6,0)  
  
(0,1) (0,3) (0,5)  
(0,2) (0,4) (0,6)
```

### 2.7.8 `cmatrix (const TR*,const TR*,int,int)`

Constructor

```
cmatrix::cmatrix (const TR* pRe, const TR* pIm,  
                 const int nM, const int nN);
```

creates `cmatrix` object of size `nM` by `nN` and copies every element of arrays pointed to by `pRe` and `pIm` to real and imaginary part of the matrix created respectively. Use `NULL` pointer to fill up appropriate part with zero values. Constructor throws `cvmexception` in case of memory allocation failure. See also `cmatrix`. Example:

```
using namespace cvm;
```

```
double re[] = {1., 2., 3., 4., 5., 6.};  
double im[] = {6., 5., 4., 3., 2., 1.};  
cmatrix m (re, im, 3, 2);  
std::cout << m << std::endl;  
re[1] = 7.77;  
std::cout << m << std::endl;
```

```
const double re2[] = {1., 2., 3., 4.};  
const cmatrix m2 (re2, NULL, 2, 2);  
std::cout << m2;
```

prints

```
(1,6) (4,3)  
(2,5) (5,2)  
(3,4) (6,1)
```

```
(1,6) (4,3)  
(2,5) (5,2)  
(3,4) (6,1)
```

```
(1,0) (3,0)  
(2,0) (4,0)
```

### 2.7.9 `cmatrix (const rmatrix&, const rmatrix&)`

Constructor

```
cmatrix::cmatrix (const rmatrix& mRe, const rmatrix& mIm);
```

creates `cmatrix` object of size `mRe.msize()` by `mRe.nsize()` (if one of these sizes differs from appropriate size of matrix `mIm` then constructor throws `cvexception`) and copies matrices `mRe` and `mIm` to real and imaginary part of the matrix created respectively. Constructor throws `cvexception` in case of memory allocation failure. See also `cmatrix`, `rmatrix`. Example:

```
using namespace cvm;
```

```
rmatrix mr(3,3), mi(3,3);  
mr(1,1) = 1.;  
mr(2,2) = 2.;  
mr(3,3) = 3.;  
mi(1,3) = 6.;  
mi(2,2) = 5.;  
mi(3,1) = 4.;
```

```
const cmatrix mc(mr, mi);  
std::cout << mc;
```

prints

```
(1,0) (0,0) (0,6)  
(0,0) (2,5) (0,0)  
(0,4) (0,0) (3,0)
```

### 2.7.10 **submatrix**

Submatrix constructor

```
cmatrix::cmatrix (cmatrix& m, int nRow, int nCol,  
                 int nHeight, int nWidth);
```

creates *cmatrix* object as *submatrix* of *m*. It means that the matrix object created shares memory with some part of *m*. This part is defined by its upper left corner (parameters *nRow* and *nCol*, both are *l*-based) and its height and width (parameters *nHeight* and *nWidth*). See also *cmatrix*. Example:

```
using namespace cvm;
```

```
cmatrix m(4,5);  
cmatrix subm(m, 2, 2, 2, 2);  
subm.set(std::complex<double>(1.,1.));
```

```
std::cout << m;
```

prints

```
(0,0) (0,0) (0,0) (0,0) (0,0)  
(0,0) (1,1) (1,1) (0,0) (0,0)  
(0,0) (1,1) (1,1) (0,0) (0,0)  
(0,0) (0,0) (0,0) (0,0) (0,0)
```

**2.7.11 operator (,)**

Indexing operators

```
TC& cmatrix::operator () (int im, int in) throw (cvmexception);
TC cmatrix::operator () (int im, int in) const throw (cvmexception);
```

provide access to a particular element of calling matrix. The first version of operator is applicable to non-constant object. This version returns *l-value* in order to make possible write access to an element. Both operators are *l-based*. Operators throw **cvmexception** if *im* is outside of [1, *size()*] range or *in* is outside of [1, *size()*] range. Operators are *inherited* in the classes **scmatrix**, **scbmatrix** and **schmatrix**. See also **cmatrix**. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6.,
                  7., 8., 9., 10., 11., 12.};
    const cmatrix m ((std::complex<double>*) a, 2, 3);
    scmatrix ms(2);
    std::cout << m(1,1) << " "
                << m(2,3) << std::endl << std::endl;
    ms(2,2) = std::complex<double>(7.77,7.77);
    std::cout << ms;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

(1,2) (11,12)

(0,0) (0,0)

(0,0) (7.77,7.77)

**2.7.12 operator ()**

Indexing operators

```
cvector cmatrix::operator () (int i) throw (cvmexception);
const cvector cmatrix::operator () (int i) const throw (cvmexception);
```

provide access to *i*-th column of calling matrix. The first version of operator is applicable to non-constant object and *returns l-value*, i.e. vector returned shares memory with *i*-th column of calling matrix in order to make possible write access to it. The second version creates *copy* of a column and therefore is *not l-value*. Both operators are *l-based*. The operators throw **cvmexception** if parameter *i* is outside of [1, nsize()] range. Operators are *inherited* in the classes **scmatrix**, **scbmatrix** and **schmatrix**. See also **cmatrix**, **Matrix::msize()**. Example:

```
using namespace cvm;

try {
    double a[] = {1., 2., 3., 4., 5., 6.,
                  7., 8., 9., 10., 11., 12.};
    const cmatrix m ((std::complex<double>*) a, 2, 3);
    scmatrix ms(2);

    std::cout << m(2) << std::endl;

    ms(2) = m(3);
    std::cout << ms;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(5,6) (7,8)

(0,0) (9,10)
(0,0) (11,12)
```

**2.7.13 operator []**

Indexing operators

```
cvector cmatrix::operator [] (int i) throw (cvmexception);
const cvector cmatrix::operator [] (int i) const throw (cvmexception);
```

provide access to *i*-th row of calling matrix. The first version of operator is applicable to non-constant object and *returns l-value*, i.e. the vector returned shares memory with *i*-th row of calling matrix in order to make possible write access to it. The second version creates *copy* of a row and therefore is *not l-value*. Both operators are *l-based*. The operators throw `cvmexception` if *i* is outside of `[1,msize()]` range. Operators are *inherited* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix`, `Matrix::msize()`. Example:

```
using namespace cvm;

try {
    double a[] = {1., 2., 3., 4., 5., 6.,
                  7., 8., 9., 10., 11., 12.};
    const cmatrix m ((std::complex<double>*) a, 2, 3);
    scmatrix ms(3);
    std::cout << m[1] << std::endl;
    ms[1] = m[2];
    std::cout << ms;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

(1,2) (5,6) (9,10)

(3,4) (7,8) (11,12)
(0,0) (0,0) (0,0)
(0,0) (0,0) (0,0)
```

**2.7.14 diag**

Functions

```
cvector cmatrix::diag (int i) throw (cvmexception);
const cvector cmatrix::diag (int i) const throw (cvmexception);
```

provide access to *i*-th diagonal of calling matrix, where *i* = 0 for main diagonal, *i* < 0 for lower diagonals and *i* > 0 for upper ones. The first version of function is applicable to non-constant object and *returns l-value*, i.e. the vector returned shares memory with *i*-th diagonal of calling matrix in order to make possible write access to it. The second version creates *copy* of a diagonal and therefore is *not l-value*. Functions throw **cvmexception** if parameter *i* is outside of [-*m*.size()+1,*n*.size()-1] range. Functions are *inherited* in the classes **scmatrix** and **scbmatrix**. Functions are *redefined* in the class **schmatrix**. See also **cmatrix**. Example:

```
using namespace cvm;

try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
                  10., 11., 12., 13., 14., 15., 16., 17., 18.};
    cmatrix m (2, 3);
    const scmatrix ms((std::complex<double>*)a, 3);
    m.diag(-1).set(std::complex<double>(1.,1.));
    m.diag(0).set(std::complex<double>(2.,2.));
    m.diag(1).set(std::complex<double>(3.,3.));
    m.diag(2).set(std::complex<double>(4.,4.));
    std::cout << m << std::endl;
    std::cout << ms << std::endl;
    std::cout << ms.diag(0) << ms.diag(1);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

(2,2) (3,3) (4,4)
(1,1) (2,2) (3,3)

(1,2) (7,8) (13,14)
(3,4) (9,10) (15,16)
(5,6) (11,12) (17,18)

(1,2) (9,10) (17,18)
(7,8) (15,16)
```



### 2.7.15 `real`

Function

```
const rmatrix cmatrix::real () const;
```

creates object of type `const rmatrix` as real part of calling matrix. Please note that, unlike `cvector::real`, this function creates new object *not sharing* memory with real part of calling matrix, i.e. the matrix returned is *not l-value*. Function is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `rmatrix`, `cmatrix`. Example:

```
using namespace cvm;  
  
double a[] = {1., 2., 3., 4., 5., 6.,  
             7., 8., 9., 10., 11., 12.};  
cmatrix m((std::complex<double>*) a, 2, 3);
```

```
std::cout << m << std::endl << m.real();
```

prints

```
(1,2) (5,6) (9,10)  
(3,4) (7,8) (11,12)
```

```
1 5 9  
3 7 11
```

### 2.7.16 `imag`

Function

```
const rmatrix cmatrix::imag () const;
```

creates object of type `const rmatrix` as imaginary part of calling matrix. Please note that, unlike `cvector::imag`, this function creates new object *not sharing* memory with imaginary part of calling matrix, i.e. the matrix returned is *not l-value*. Function is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `rmatrix`, `cmatrix`. Example:

```
using namespace cvm;  
  
double a[] = {1., 2., 3., 4., 5., 6.,  
              7., 8., 9., 10., 11., 12.};  
cmatrix m((std::complex<double>*) a, 2, 3);
```

```
std::cout << m << std::endl << m.imag();
```

prints

```
(1,2) (5,6) (9,10)  
(3,4) (7,8) (11,12)
```

```
2 6 10  
4 8 12
```

**2.7.17 operator = (const cmatrix&)**

Operator

```
cmatrix& cmatrix::operator = (const cmatrix& m) throw (cvmexception);
```

sets every element of calling *matrix* to be equal to appropriate element of matrix *m* and returns reference to the matrix changed. Operator throws **cvmexception** in case of different matrix sizes. Operator is *redefined* in the classes **scmatrix**, **scbmatrix** and **schmatrix**. See also **cmatrix**. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6.,
                  7., 8., 9., 10., 11., 12.};
    const cmatrix m1((std::complex<double>*) a, 2, 3);
    cmatrix m2(2, 3);

    m2 = m1;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(1,2) (5,6) (9,10)
(3,4) (7,8) (11,12)
```

### 2.7.18 `assign (const TC*)`

Function

```
cmatrix& cmatrix::assign (const cvector& v) throw (cvmexception);  
cmatrix& cmatrix::assign (const TC* pD);
```

sets every element of calling matrix to value of appropriate element of a vector `v` or array pointed to by `pD` and returns reference to the matrix changed. In first version it's assumed that vector passed is long enough to fill calling matrix. Function throws `cvmexception` otherwise. Function is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix`. Example:

```
using namespace cvm;
```

```
const double a[] = {1., 2., 3., 4., 5., 6.,  
                   7., 8., 9., 10., 11., 12.};
```

```
cmatrix m(2, 3);
```

```
m.assign ((const std::complex<double>*) a);  
std::cout << m;
```

prints

```
(1,2) (5,6) (9,10)  
(3,4) (7,8) (11,12)
```

**2.7.19 assign (int, int, const cmatrix&)**

Function

```
cmatrix& cmatrix::assign (int nRow, int nCol, const cmatrix& m)
throw (cvmexception);
```

sets sub-matrix of calling matrix beginning with *l*-based row nRow and column nCol to matrix m and returns reference to the matrix changed. Function throws *cvmexception* if nRow or nCol are not positive or matrix m doesn't fit. Function is *redefined* in the classes *scmatrix*, *scbmatrix* and *schmatrix*. See also *cmatrix*. Example:

```
using namespace cvm;
```

```
rmatrix m1(4,5);
rmatrix m2(2,2);
m1.set(1.);
m2.set(2.);
m1.assign(2,3,m2);
std::cout << m1;
```

prints

```
(1,1) (1,1) (1,1) (1,1) (1,1)
(1,1) (1,1) (2,2) (2,2) (1,1)
(1,1) (1,1) (2,2) (2,2) (1,1)
(1,1) (1,1) (1,1) (1,1) (1,1)
```

**2.7.20 set (TC)**

Function

```
cmatrix& cmatrix::set (TC z);
```

sets every element of calling matrix to value of parameter *z* and returns reference to the matrix changed. Use **vanish** to set every element of calling matrix to be equal to zero. Function is *redefined* in the classes **scmatrix**, **scbmatrix** and **schmatrix**. See also **cmatrix**.  
Example:

```
using namespace cvm;
```

```
cmatrix m(3, 4);  
m.set(std::complex<double>(3.,4.));  
std::cout << m;
```

prints

```
(3,4) (3,4) (3,4) (3,4)  
(3,4) (3,4) (3,4) (3,4)  
(3,4) (3,4) (3,4) (3,4)
```

### 2.7.21 `assign_real`

Function

```
cmatrix& cmatrix::assign_real (const rmatrix& mRe) throw (cvmexception);
```

sets real part of every element of calling matrix to value of appropriate element of matrix `mRe` and returns reference to the matrix changed. Function throws `cvmexception` in case of different sizes of the operands. See also `cmatrix` and `rmatrix`. Function is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. Example:

```
using namespace cvm;
```

```
rmatrix m (2,3);  
cmatrix mc(2,3);  
m.randomize (0., 1.);
```

```
mc.assign_real(m);  
std::cout << mc;
```

prints

```
(0.126835,0) (0.57271,0) (0.28312,0)  
(0.784417,0) (0.541673,0) (0.663869,0)
```

### 2.7.22 `assign_imag`

Function

```
cmatrix& cmatrix::assign_imag (const rmatrix& mIm) throw (cvmexception);
```

sets imaginary part of every element of calling matrix to value of appropriate element of matrix `mIm` and returns reference to the matrix changed. Function throws `cvmexception` in case of different sizes of the operands. Function is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix` and `rmatrix`. Example:

```
using namespace cvm;
```

```
rmatrix m (2,3);  
cmatrix mc(2,3);  
m.randomize (0., 1.);
```

```
mc.assign_imag(m);  
std::cout << mc;
```

prints

```
(0,0.13831) (0,0.267373) (0,0.482345)  
(0,0.50618) (0,0.992401) (0,0.444777)
```



### 2.7.23 `set_real`

Function

```
cmatrix& cmatrix::set_real (TR d);
```

sets real part of every element of calling matrix to value of parameter `d` and returns reference to the matrix changed. See also `cmatrix`. Function is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. Example:

```
using namespace cvm;
```

```
cmatrix m(2,3);  
m.set_real(1.);  
std::cout << m;
```

prints

```
(1,0) (1,0) (1,0)  
(1,0) (1,0) (1,0)
```

### 2.7.24 `set_imag`

Function

```
cmatrix& cmatrix::set_imag (TR d);
```

sets imaginary part of every element of calling matrix to value of parameter `d` and returns reference to the matrix changed. See also `cmatrix`. Function is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. Example:

```
using namespace cvm;
```

```
cmatrix m(2,3);  
m.set_imag(1.);  
std::cout << m;
```

prints

```
(0,1) (0,1) (0,1)  
(0,1) (0,1) (0,1)
```

**2.7.25 `resize`**

Function

```
cmatrix& cmatrix::resize (int nNewM, int nNewN) throw (cvmexception);
```

changes size of calling matrix to `nNewM` by `nNewN` and returns reference to the matrix changed. In case of increasing of its size, the matrix is filled up with zeroes. Function throws `cvmexception` in case of negative size passed or memory allocation failure. Function is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6.,
                 7., 8., 9., 10., 11., 12.};
    cmatrix m((std::complex<double>*) a, 2, 3);
    std::cout << m << std::endl;
    m.resize (2, 2);
    std::cout << m << std::endl;
    m.resize (3, 3);
    std::cout << m;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(1,2) (5,6) (9,10)
(3,4) (7,8) (11,12)
```

```
(1,2) (5,6)
(3,4) (7,8)
```

```
(1,2) (5,6) (0,0)
(3,4) (7,8) (0,0)
(0,0) (0,0) (0,0)
```

### 2.7.26 `operator ==`

Operator

```
bool cmatrix::operator == (const cmatrix& m) const;
```

compares calling matrix with matrix `m` and returns true if they have the same sizes and their appropriate elements differ by not more than the **smallest normalized positive number**. Returns false otherwise. Operator is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix`. Example:

```
using namespace cvm;
```

```
cmatrix m1(2, 3);  
cmatrix m2(2, 3);  
m1.set_real(1.);  
m2.set_real(1.);  
std::cout << (m1 == m2) << std::endl;
```

prints

```
1
```

### 2.7.27 operator !=

Operator

```
bool cmatrix::operator != (const cmatrix& m) const;
```

compares calling matrix with matrix *m* and returns true if they have different sizes or at least of their appropriate elements differs by more than the **smallest normalized positive number**. Returns false otherwise. Operator is *redefined* in the classes **scmatrix**, **scbmatrix** and **schmatrix**. See also **cmatrix**. Example:

```
using namespace cvm;
```

```
cmatrix m1(2, 3);  
cmatrix m2(2, 3);  
m1.set_real(1.);  
m2.set_real(1.);  
std::cout << (m1 != m2) << std::endl;
```

prints

```
0
```

**2.7.28 operator <<**

Operator

```
cmatrix& cmatrix::operator << (const cmatrix& m) throw (cvmexception);
```

destroys calling matrix, creates new one as a copy of *m* and returns reference to the matrix changed. Operator throws **cvmexception** in case of memory allocation failure. Operator is *redefined* in the classes **scmatrix**, **scbmatrix** and **schmatrix**. See also **cmatrix**. Example:

```
using namespace cvm;
```

```
try {
    cmatrix m(2,3);
    cmatrix mc(1,1);
    m(1,2) = std::complex<double>(1.,2.);
    m(2,3) = std::complex<double>(2.,4.);
    std::cout << m << mc << std::endl;

    mc << m;
    std::cout << mc;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(0,0) (1,2) (0,0)
(0,0) (0,0) (2,4)
(0,0)
```

```
(0,0) (1,2) (0,0)
(0,0) (0,0) (2,4)
```

**2.7.29 operator +**

Operator

```
cmatrix cmatrix::operator + (const cmatrix& m) const
throw (cvmexception);
```

creates object of type `cmatrix` as sum of calling matrix and matrix `m`. Operator throws `cvmexception` in case of different sizes of the operands. Operator is *redefined* in the classes `scmatrix`, `schmatrix` and `schmatrix`. See also `cvector::sum`, `cmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6.,
                  7., 8., 9., 10., 11., 12.};
    const cmatrix ma ((std::complex<double>*) a, 2, 3);
    cmatrix mb (2, 3);
    mb.set (std::complex<double>(1.,1.));

    std::cout << ma + mb << std::endl;
    std::cout << ma + ma;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(2,3) (6,7) (10,11)
(4,5) (8,9) (12,13)

(2,4) (10,12) (18,20)
(6,8) (14,16) (22,24)
```

**2.7.30 operator -**

Operator

```
cmatrix cmatrix::operator - (const cmatrix& m) const
throw (cvmexception);
```

creates object of type `cmatrix` as difference of calling matrix and matrix `m`. It throws `cvmexception` in case of different sizes of the operands. Operator is *redefined* in the classes `scmatrix`, `schmatrix` and `schmatrix`. See also `cvector::diff`, `cmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6.,
                  7., 8., 9., 10., 11., 12.};
    const cmatrix ma ((std::complex<double>*) a, 2, 3);
    cmatrix mb (2, 3);
    mb.set (std::complex<double>(1.,1.));

    std::cout << ma - mb << std::endl;
    std::cout << ma - ma;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(0,1) (4,5) (8,9)
(2,3) (6,7) (10,11)

(0,0) (0,0) (0,0)
(0,0) (0,0) (0,0)
```



**2.7.31 sum**

Function

```
cmatrix& cmatrix::sum (const cmatrix& m1, const cmatrix& m2)
throw (cvmexception);
```

assigns result of addition of matrices m1 and m2 to calling matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. Function is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix::operator +`, `cmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6.,
                  7., 8., 9., 10., 11., 12.};
    const cmatrix ma ((std::complex<double>*) a, 2, 3);
    cmatrix mb (2, 3);
    cmatrix m (2, 3);
    mb.set (std::complex<double>(1.,1.));

    std::cout << m.sum(ma, mb) << std::endl;
    std::cout << m.sum(m, mb);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(2,3) (6,7) (10,11)
(4,5) (8,9) (12,13)

(3,4) (7,8) (11,12)
(5,6) (9,10) (13,14)
```

**2.7.32 diff**

Function

```
cmatrix& cmatrix::diff (const cmatrix& m1, const cmatrix& m2)
throw (cvmexception);
```

assigns result of subtraction of matrices m1 and m2 to calling matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. Function is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix::operator -`, `cmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6.,
                  7., 8., 9., 10., 11., 12.};
    const cmatrix ma ((std::complex<double>*) a, 2, 3);
    cmatrix mb (2, 3);
    cmatrix m (2, 3);
    mb.set (std::complex<double>(1.,1.));

    std::cout << m.diff(ma, mb) << std::endl;
    std::cout << m.diff(m, mb);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(0,1) (4,5) (8,9)
(2,3) (6,7) (10,11)

(-1,0) (3,4) (7,8)
(1,2) (5,6) (9,10)
```

### 2.7.33 operator +=

Operator

```
cmatrix& cmatrix::operator += (const cmatrix& m) throw (cvmexception);
```

adds matrix *m* to calling matrix and returns reference to the matrix changed. It throws **cvmexception** in case of different sizes of the operands. Operator is *redefined* in the classes **scmatrix**, **scbmatrix** and **schmatrix**. See also **cmatrix::operator +** , **cmatrix::sum**, **cmatrix**. Example:

```
using namespace cvm;
```

```
try {
    cmatrix m1(2, 3);
    cmatrix m2(2, 3);
    m1.set(std::complex<double>(1.,1.));
    m2.set(std::complex<double>(2.,2.));

    m1 += m2;
    std::cout << m1 << std::endl;

    // well, you can do this too, but temporary object would be created
    m2 += m2;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(3,3) (3,3) (3,3)
(3,3) (3,3) (3,3)

(4,4) (4,4) (4,4)
(4,4) (4,4) (4,4)
```

**2.7.34 operator -=**

Operator

```
cmatrix& cmatrix::operator -= (const cmatrix& m) throw (cvmexception);
```

subtracts matrix *m* from calling matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. Operator is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix::operator -`, `cmatrix::diff`, `cmatrix`. Example:

```
using namespace cvm;
```

```
try {
    cmatrix m1(2, 3);
    cmatrix m2(2, 3);
    m1.set(std::complex<double>(1.,1.));
    m2.set(std::complex<double>(2.,2.));

    m1 -= m2;
    std::cout << m1 << std::endl;

    // well, you can do this too, but temporary object would be created
    m2 -= m2;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(-1,-1) (-1,-1) (-1,-1)
(-1,-1) (-1,-1) (-1,-1)
```

```
(0,0) (0,0) (0,0)
(0,0) (0,0) (0,0)
```

### 2.7.35 operator - ()

Operator

```
cmatrix cmatrix::operator - () const throw (cvmexception);
```

creates object of type `cmatrix` as calling matrix multiplied by  $-1$ . Operator is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6.,  
              7., 8., 9., 10., 11., 12.};  
const cmatrix ma ((std::complex<double>*) a, 2, 3);
```

```
std::cout << - ma;
```

prints

```
(-1,-2) (-5,-6) (-9,-10)  
(-3,-4) (-7,-8) (-11,-12)
```

### 2.7.36 operator \* (TR)

Operator

```
cmatrix cmatrix::operator * (TR d) const;
```

creates object of type `cmatrix` as product of calling matrix and real number `d`. Operator is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix::operator *=`, `cmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6.,  
              7., 8., 9., 10., 11., 12.};  
const cmatrix ma ((std::complex<double>*) a, 2, 3);
```

```
std::cout << ma * 5.;
```

prints

```
(5,10) (25,30) (45,50)  
(15,20) (35,40) (55,60)
```

### 2.7.37 operator / (TR)

Operator

```
cmatrix cmatrix::operator / (TR d) const throw (cvmexception);
```

creates object of type `cmatrix` as quotient of calling matrix and real number `d`. It throws `cvmexception` if `d` has absolute value equal or less than the **smallest normalized positive number**. Operator is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix::operator /=`, `cmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6.,
                 7., 8., 9., 10., 11., 12.};
    const cmatrix ma ((std::complex<double>*) a, 2, 3);

    std::cout << ma / 4.;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(0.25,0.5) (1.25,1.5) (2.25,2.5)
(0.75,1) (1.75,2) (2.75,3)
```

### 2.7.38 `operator *` (TC)

Operator

```
cmatrix cmatrix::operator * (TC z) const;
```

creates object of type `cmatrix` as product of calling matrix and complex number `z`. Operator is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix`, `cmatrix::operator *=`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6.,  
             7., 8., 9., 10., 11., 12.};  
const cmatrix ma ((std::complex<double>*) a, 2, 3);
```

```
std::cout << ma * std::complex<double>(5.,2.);
```

prints

```
(1,12) (13,40) (25,68)  
(7,26) (19,54) (31,82)
```



**2.7.39 operator / (TC)**

Operator

```
cmatrix cmatrix::operator / (TC z) const throw (cvmexception);
```

creates object of type `cmatrix` as quotient of calling matrix and complex number `z`. It throws `cvmexception` if `z` has absolute value equal or less than the **smallest normalized positive number**. Operator is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix::operator /=`, `cmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6.,
                 7., 8., 9., 10., 11., 12.};
    const cmatrix ma ((std::complex<double>*) a, 2, 3);

    std::cout << ma / std::complex<double>(4.,2.);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(0.4,0.3) (1.6,0.7) (2.8,1.1)
(1,0.5) (2.2,0.9) (3.4,1.3)
```

**2.7.40 operator \*= (TR)**

Operator

```
cmatrix& cmatrix::operator *= (TR d);
```

multiplies calling matrix by real number *d* and returns reference to the matrix changed. Operator is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix`, `cmatrix::operator *`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6.,  
              7., 8., 9., 10., 11., 12.};  
cmatrix ma ((std::complex<double>*) a, 2, 3);
```

```
ma *= 2.;  
std::cout << ma;
```

prints

```
(2,4) (10,12) (18,20)  
(6,8) (14,16) (22,24)
```

**2.7.41 operator /= (TR)**

Operator

```
cmatrix& cmatrix::operator /= (TR d) throw (cvmexception);
```

divides calling matrix by real number d and returns reference to the matrix changed. It throws **cvmexception** if d has absolute value equal or less than the **smallest normalized positive number**. Operator is *redefined* in the classes **scmatrix**, **scbmatrix** and **schmatrix**. See also **cmatrix::operator /** , **cmatrix**. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6.,
                  7., 8., 9., 10., 11., 12.};
    cmatrix ma ((std::complex<double>*) a, 2, 3);

    ma /= 2.;
    std::cout << ma;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(0.5,1) (2.5,3) (4.5,5)
(1.5,2) (3.5,4) (5.5,6)
```

**2.7.42 operator \*= (TC)**

Operator

```
cmatrix& cmatrix::operator *= (TC z);
```

multiplies calling matrix by complex number *z* and returns reference to the matrix changed. Operator is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix::operator *`, `cmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6.,  
             7., 8., 9., 10., 11., 12.};  
cmatrix ma ((std::complex<double>*) a, 2, 3);
```

```
ma *= std::complex<double>(2.,1.);  
std::cout << ma;
```

prints

```
(0,5) (4,17) (8,29)  
(2,11) (6,23) (10,35)
```

**2.7.43 operator /= (TC)**

Operator

```
cmatrix& cmatrix::operator /= (TC z) throw (cvmexception);
```

divides calling matrix by complex number *z* and returns reference to the matrix changed. It throws **cvmexception** if *z* has absolute value equal or less than the **smallest normalized positive number**. Operator is *redefined* in the classes **scmatrix**, **scbmatrix** and **schmatrix**. See also **cmatrix::operator /** , **cmatrix**. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6.,
                  7., 8., 9., 10., 11., 12.};
    cmatrix ma ((std::complex<double>*) a, 2, 3);

    ma /= std::complex<double>(2.,1.);
    std::cout << ma;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(0.8,0.6) (3.2,1.4) (5.6,2.2)
(2,1) (4.4,1.8) (6.8,2.6)
```

### 2.7.44 **normalize**

Function

```
cmatrix& cmatrix::normalize ();
```

normalizes calling matrix so its **Euclidean norm** becomes equal to 1 if it was greater than the **smallest normalized positive number** before the call (otherwise function does nothing). Function is *redefined* in the classes **scmatrix**, **scbmatrix** and **schmatrix**. See also **cmatrix**.  
Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6.,  
             7., 8., 9., 10., 11., 12.};  
cmatrix ma ((std::complex<double>*) a, 2, 3);
```

```
ma.normalize();  
std::cout << ma << ma.norm() << std::endl;
```

prints

```
(0.0392232,0.0784465) (0.196116,0.235339) (0.353009,0.392232)  
(0.11767,0.156893) (0.274563,0.313786) (0.431455,0.470679)  
1
```

**2.7.45 conjugation**

Operator and functions

```
cmatrix cmatrix::operator ~ () const throw (cvmexception);
cmatrix& cmatrix::conj (const cmatrix& m) throw (cvmexception);
cmatrix& cmatrix::conj () throw (cvmexception);
```

implement complex matrix conjugation. First operator creates object of type `cmatrix` as conjugated calling matrix (it throws `cvmexception` in case of memory allocation failure). Second function sets calling matrix to be equal to matrix `m` conjugated (it throws `cvmexception` in case of not appropriate sizes of the operands), third one makes it to be equal to conjugated itself (it also throws `cvmexception` in case of memory allocation failure). Functions are *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix::tramspose`, `cmatrix`. Example:

```
using namespace cvm;

double a[] = {1., 2., 3., 4., 5., 6.,
              7., 8., 9., 10., 11., 12.};
cmatrix m((std::complex<double>*) a, 2, 3);
cmatrix mc(3,2);
std::cout << m << std::endl << ~m << std::endl ;
mc.conj(m);
std::cout << mc << std::endl;
mc.conj();
std::cout << mc;
```

prints

```
(1,2) (5,6) (9,10)
(3,4) (7,8) (11,12)
```

```
(1,-2) (3,-4)
(5,-6) (7,-8)
(9,-10) (11,-12)
```

```
(1,-2) (3,-4)
(5,-6) (7,-8)
(9,-10) (11,-12)
```

```
(1,2) (5,6) (9,10)
(3,4) (7,8) (11,12)
```

**2.7.46 transposition**

Operator and functions

```
cmatrix cmatrix::operator ! () const throw (cvmexception);
cmatrix& cmatrix::transpose (const cmatrix& m) throw (cvmexception);
cmatrix& cmatrix::transpose () throw (cvmexception);
```

implement complex matrix transposition (*not* conjugation). First operator creates object of type `cmatrix` as transposed calling matrix (it throws `cvmexception` in case of memory allocation failure). Second function sets calling matrix to be equal to matrix `m` transposed (it throws `cvmexception` in case of not appropriate sizes of the operands), third one makes it to be equal to transposed itself (it also throws `cvmexception` in case of memory allocation failure). Functions are *redefined* in the classes `scmatrix`, `schmatrix` and `schmatrix`. See also `cmatrix::conj`, `cmatrix`. Example:

```
using namespace cvm;

double a[] = {1., 2., 3., 4., 5., 6.,
              7., 8., 9., 10., 11., 12.};
cmatrix m((std::complex<double>*) a, 2, 3);
cmatrix mc(3,2);
std::cout << m << std::endl << !m << std::endl ;
mc.transpose(m);
std::cout << mc << std::endl;
mc.transpose();
std::cout << mc;
```

prints

```
(1,2) (5,6) (9,10)
(3,4) (7,8) (11,12)
```

```
(1,2) (3,4)
(5,6) (7,8)
(9,10) (11,12)
```

```
(1,2) (3,4)
(5,6) (7,8)
(9,10) (11,12)
```

```
(1,2) (5,6) (9,10)
(3,4) (7,8) (11,12)
```



### 2.7.47 `operator * (const cvector&)`

Operator

```
cvector cmatrix::operator * (const cvector& v) const  
throw (cvmexception);
```

creates object of type `cvector` as product of calling matrix and vector `v`. It throws `cvmexception` if number of columns of calling matrix differs from size of vector `v`. Use `cvector::mult` in order to avoid of new object creation. Function is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix`, `cvector`. Example:

```
using namespace cvm;
```

```
cmatrix m(2, 3);  
cvector v(3);  
m.set(std::complex<double>(1.,1.));  
v.set(std::complex<double>(1.,1.));
```

```
std::cout << m * v;
```

prints

```
(0,6) (0,6)
```

**2.7.48 operator \* (const cmatrix&)**

Operator

```
cmatrix cmatrix::operator * (const cmatrix& m) const  
throw (cvmexception);
```

creates object of type `cmatrix` as product of calling matrix and matrix `m`. It throws `cvmexception` if number of columns of calling matrix differs from number of rows of matrix `m`. Use `cmatrix::mult` in order to avoid of new object creation. Operator is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix`. Example:

```
using namespace cvm;
```

```
cmatrix m1(2, 3);  
cmatrix m2(3, 2);  
m1.set(std::complex<double>(1.,1.));  
m2.set(std::complex<double>(1.,1.));
```

```
std::cout << m1 * m2;
```

prints

```
(0,6) (0,6)  
(0,6) (0,6)
```

### 2.7.49 `mult`

Function

```
cmatrix& cmatrix::mult (const cmatrix& m1, const cmatrix& m2)
throw (cvmexception);
```

sets calling matrix to be equal to product of matrix `m1` by matrix `m2` and returns reference to the matrix changed. Function throws `cvmexception` in case of inappropriate sizes of the operands. Function is *inherited* in the class `scmatrix` and *redefined* in the classes `scbmatrix` and `schmatrix`. See also `cmatrix`. Example:

```
using namespace cvm;
```

```
cmatrix m1(2, 3);
cmatrix m2(3, 2);
scmatrix m(2);
m1.set(std::complex<double>(1.,1.));
m2.set(std::complex<double>(1.,1.));
```

```
m.mult(m1, m2);
std::cout << m;
```

prints

```
(0,6) (0,6)
(0,6) (0,6)
```

### 2.7.50 `ranklupdate_u`

Function

```
cmatrix&  
cmatrix::ranklupdate_u (const cvector& vCol, const cvector& vRow)  
throw (cvmexception);
```

sets calling matrix to be equal to **rank-1 update** (uncojugated) of vectors `vCol` and `vRow` and returns reference to the matrix changed. Function throws **cvmexception** if number of rows of calling matrix is not equal to `vCol.size()` or number of columns is not equal to `vRow.size()`. Function is *inherited* in the class **scmatrix** and *not applicable* to objects of the classes **scbmatrix** and **schmatrix** (i.e. **cvmexception** would be thrown in case of using it for objects of those classes). See also **cvector::ranklupdate\_u**, **cmatrix**. Example:

```
using namespace cvm;  
  
cvector vc(3), vr(2);  
cmatrix m(3, 2);  
vc.set(std::complex<double>(1.,1.));  
vr.set(std::complex<double>(1.,1.));  
  
std::cout << m.ranklupdate_u (vc, vr);
```

prints

```
(0,2) (0,2)  
(0,2) (0,2)  
(0,2) (0,2)
```

**2.7.51 ranklupdate\_c**

Function

```
cmatrix&
cmatrix::ranklupdate_c (const cvector& vCol, const cvector& vRow)
throw (cvmexception);
```

sets calling matrix to be equal to **rank-1 update** (conjugated) of vectors `vCol` and `vRow` and returns reference to the matrix changed. Function throws **cvmexception** if number of rows of calling matrix is not equal to `vCol.size()` or number of columns is not equal to `vRow.size()`. Function is *inherited* in the class **scmatrix** and *not applicable* to objects of the classes **scbmatrix** and **schmatrix** (i.e. **cvmexception** would be thrown in case of using it for objects of those classes). See also **cvector::ranklupdate\_c**, **cmatrix**. Example:

```
using namespace cvm;

cvector vc(3), vr(2);
cmatrix m(3, 2);
vc.set(std::complex<double>(1.,1.));
vr.set(std::complex<double>(1.,1.));

std::cout << m.ranklupdate_c (vc, vr);
```

prints

```
(2,0) (2,0)
(2,0) (2,0)
(2,0) (2,0)
```

### 2.7.52 `swap_rows`

Function

```
cmatrix& cmatrix::swap_rows (int n1, int n2) throw (cvmexception);
```

swaps two rows of calling matrix and returns reference to the matrix changed. `n1` and `n2` are numbers of rows to be swapped, both are *ℓ-based*). Function throws `cvmexception` if one of parameters is outside of the range `[1,msize()]`. Function is *redefined* in the class `scmatrix` and *not applicable* to objects of the classes `scbmatrix` and `schmatrix` (i.e. `cvmexception` would be thrown in case of using it for objects of those classes). See also `cmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6.,  
              7., 8., 9., 10., 11., 12.};  
cmatrix m ((std::complex<double>*)a, 3, 2);
```

```
std::cout << m << std::endl;  
std::cout << m.swap_rows(2,3);
```

prints

```
(1,2) (7,8)  
(3,4) (9,10)  
(5,6) (11,12)
```

```
(1,2) (7,8)  
(5,6) (11,12)  
(3,4) (9,10)
```

### 2.7.53 `swap_cols`

Function

```
cmatrix& cmatrix::swap_cols (int n1, int n2) throw (cvmexception);
```

swaps two columns of calling matrix and returns reference to the matrix changed. `n1` and `n2` are numbers of columns to be swapped, both are *ℓ-based*). Function throws `cvmexception` if one of parameters is outside of the range `[1, nsize()]`. Function is *redefined* in the class `scmatrix` and *not applicable* to objects of the classes `scbmatrix` and `schmatrix` (i.e. `cvmexception` would be thrown in case of using it for objects of those classes). See also `cmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6.,  
             7., 8., 9., 10., 11., 12.};  
cmatrix m ((std::complex<double>*)a, 2, 3);
```

```
std::cout << m << std::endl;  
std::cout << m.swap_cols(2,3);
```

prints

```
(1,2) (5,6) (9,10)  
(3,4) (7,8) (11,12)
```

```
(1,2) (9,10) (5,6)  
(3,4) (11,12) (7,8)
```

**2.7.54 solve**

Functions

```
cmatrix&
cmatrix::solve (const scmatrix& mA,
                const cmatrix& mB, TR& dErr) throw (cvmexception);

cmatrix&
cmatrix::solve (const scmatrix& mA,
                const cmatrix& vB) throw (cvmexception);
```

set calling matrix to be equal to solution  $X$  of the matrix linear equation  $A * X = B$  where parameter `mA` is square matrix  $A$  and parameter `vB` is matrix  $B$ . Every function returns reference to the matrix changed. The first version also sets output parameter `dErr` to be equal to the norm of computation error. Functions throw `cvmexception` in case of inappropriate sizes of operands or when matrix  $A$  is close to cingular. Functions are *redefined* in the class `scmatrix` and *inherited* thereafter in the classes `scbmatrix` and `schmatrix`. See also `cvector::solve`, `cmatrix`, `scmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (4);

scmatrix ma(3);
cmatrix mb(3,2);
cmatrix mx(3,2);
double dErr;
ma.randomize_real(0.,10.); ma.randomize_imag(0.,10.);
mb.randomize_real(0.,10.); mb.randomize_imag(0.,10.);

mx.solve (ma, mb, dErr);
std::cout << mx << std::endl << ma * mx - mb
          << dErr << std::endl;
```

prints

```
(+3.2103e-001,+2.8598e-002) (+1.1907e+000,-1.1101e+000)
(+3.7451e-001,+4.2471e-001) (-2.3858e-001,+3.3028e-001)
(+3.0085e-001,-3.0427e-001) (-7.6874e-001,+4.1100e-001)

(+4.4409e-016,+0.0000e+000) (-8.8818e-016,+1.3323e-015)
(+7.7716e-016,-8.8818e-016) (+0.0000e+000,+1.7764e-015)
(+0.0000e+000,-8.8818e-016) (-1.7764e-015,+8.8818e-016)
+5.4774e-015
```



**2.7.55 solve\_tran**

Functions

```
cmatrix&
cmatrix::solve_tran (const scmatrix& mA,
                    const cmatrix& mB, TR& dErr) throw (cvmexception);

cmatrix&
cmatrix::solve_tran (const scmatrix& mA,
                    const cmatrix& vB) throw (cvmexception);
```

set calling matrix to be equal to solution  $X$  of matrix linear equation  $A^T * X = B$  (which is equivalent to  $X^T * A = B^T$ ) where parameter `mA` is square matrix  $A$  and parameter `vB` is matrix  $B$ . Every function returns reference to the matrix changed. The first version also sets output parameter `dErr` to be equal to the norm of computation error. Functions throw `cvmexception` in case of inappropriate sizes of the operands or when the matrix  $A$  is close to singular. Functions are *redefined* in the class `scmatrix` and *inherited* thereafter in the classes `scbmatrix` and `schmatrix`. See also `cmatrix::solve`, `cvector::solve_tran`, `cmatrix`, `scmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (3);

scmatrix ma(3);
cmatrix mb(3,2);
cmatrix mx(3,2);
double dErr;
ma.randomize_real(0.,10.); ma.randomize_imag(0.,10.);
mb.randomize_real(0.,10.); mb.randomize_imag(0.,10.);

mx.solve_tran (ma, mb, dErr);

std::cout << !ma * mx - mb << dErr << std::endl;
std::cout << !mx * ma - !mb << std::endl;

prints

(+6.661e-016,+1.776e-015) (+0.000e+000,+5.690e-016)
(-1.776e-015,-1.776e-015) (-8.882e-016,-8.882e-016)
(-2.665e-015,+1.776e-015) (+0.000e+000,-1.110e-015)
+7.079e-015
(+6.661e-016,+1.776e-015) (-1.776e-015,-1.776e-015) (-1.776e-015,+1.776e-015)
(+0.000e+000,+5.690e-016) (-8.882e-016,-8.882e-016) (+0.000e+000,-1.554e-015)
```

**2.7.56 solve\_conj**

Functions

```

cmatrix&
cmatrix::solve_conj (const scmatrix& mA,
                    const cmatrix& mB, TR& dErr) throw (cvmexception);

cmatrix&
cmatrix::solve_conj (const scmatrix& mA,
                    const cmatrix& vB) throw (cvmexception);

```

set calling matrix to be equal to solution  $X$  of matrix linear equation  $A^H * X = B$  (which is equivalent to  $X^H * A = B^H$ ), where  $A^H$  is conjugated  $A$ . Here parameter `mA` is square matrix  $A$  and parameter `vB` is matrix  $B$ . Every function returns reference to the matrix changed. The first version also sets output parameter `dErr` to be equal to the norm of computation error. Functions throw `cvmexception` in case of inappropriate sizes of the operands or when the matrix  $A$  is close to cingular. Functions are *redefined* in the class `scmatrix` and *inherited* thereafter in the classes `scbmatrix` and `schmatrix`. See also `cmatrix::solve`, `cvector::solve_conj`, `cmatrix`, `scmatrix`. Example:

```

using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (3);

scmatrix ma(3);
cmatrix  mb(3,2);
cmatrix  mx(3,2);
double dErr;
ma.randomize_real(0.,10.); ma.randomize_imag(0.,10.);
mb.randomize_real(0.,10.); mb.randomize_imag(0.,10.);

mx.solve_conj (ma, mb, dErr);

std::cout << ~ma * mx - mb << dErr << std::endl;
std::cout << ~mx * ma - ~mb << std::endl;

prints

(+7.105e-015,-1.776e-015) (+1.332e-015,+8.882e-016)
(+1.776e-015,-8.882e-016) (+8.882e-016,-8.882e-016)
(+0.000e+000,-6.661e-016) (-1.776e-015,-2.665e-015)
+1.169e-014
(+7.105e-015,+1.776e-015) (+1.776e-015,+8.882e-016) (-2.665e-015,+6.661e-016)
(+1.332e-015,-8.882e-016) (+8.882e-016,+8.882e-016) (-1.776e-015,+2.665e-015)

```

**2.7.57 solve\_lu**

Functions

```
cmatrix&
cmatrix::solve_lu (const scmatrix& mA, const scmatrix& mLU,
                  const int* pPivots, const cmatrix& mB, TR& dErr)
                  throw (cvmexception);
```

```
cmatrix&
cmatrix::solve_lu (const scmatrix& mA, const scmatrix& mLU,
                  const int* pPivots, const cmatrix& vB)
                  throw (cvmexception);
```

set calling matrix to be equal to solution  $X$  of the matrix linear equation  $A * X = B$  where parameter `mA` is square complex matrix  $A$ , parameter `mLU` is [LU factorization](#) of matrix  $A$ , parameter `pPivots` is array of pivot numbers created while factorizing matrix  $A$  and parameter `vB` is matrix  $B$ . Every function returns reference to the matrix changed. The first version also sets output parameter `dErr` to be equal to the norm of computation error. These functions are useful when you need to solve few linear equations of kind  $AX = B$  with the same matrix  $A$  and different matrices  $B$ . In such case you save on matrix  $A$  factorization since it's needed to be performed just one time. Functions throw [cvmexception](#) in case of inappropriate sizes of operands or when matrix  $A$  is close to singular. See also [cvector::solve](#), [cmatrix](#), [scmatrix](#). Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (4);
try {
    scmatrix ma(3);
    scmatrix mLU(3);
    cmatrix mb1(3,2);
    cmatrix mb2(3,2);
    cmatrix mx1(3,2);
    cmatrix mx2(3,2);
    iarray nPivots(3);
    double dErr = 0.;
    ma.randomize_real(0.,10.); ma.randomize_imag(0.,10.);
    mb1.randomize_real(0.,10.); mb1.randomize_imag(0.,10.);
    mb2.randomize_real(0.,10.); mb2.randomize_imag(0.,10.);

    mLU.low_up(ma, nPivots);
    std::cout << mx1.solve_lu (ma, mLU, nPivots, mb1, dErr);
    std::cout << dErr << std::endl;
```

```
    std::cout << mx2.solve_lu (ma, mLU, nPivots, mb2) << std::endl;
    std::cout << ma * mx1 - mb1 << std::endl << ma * mx2 - mb2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(4.2888e-01,8.2409e-02) (-1.1261e-01,-5.7778e-01)
(5.8052e-01,3.2179e-01) (2.5811e-01,-3.8609e-02)
(-3.1499e-02,-7.0014e-01) (1.2652e+00,4.5309e-01)
5.2931e-15
(3.0153e-01,-5.6606e-01) (-1.6308e-01,1.8217e-01)
(7.4971e-01,-1.1305e-01) (5.2187e-01,2.3441e-01)
(-1.9916e-01,1.4493e+00) (9.1046e-02,3.5242e-01)

(0.0000e+00,-8.8818e-16) (0.0000e+00,-8.8818e-16)
(-4.4409e-16,0.0000e+00) (0.0000e+00,-8.8818e-16)
(0.0000e+00,0.0000e+00) (0.0000e+00,0.0000e+00)

(-8.8818e-16,8.8818e-16) (-8.8818e-16,-1.7764e-15)
(0.0000e+00,0.0000e+00) (2.2204e-16,-8.8818e-16)
(4.4409e-16,-8.8818e-16) (1.3878e-17,-4.4409e-16)
```

**2.7.58 svd**

Functions

rvector

`cmatrix::svd () throw (cvmexception);`

rvector

`cmatrix::svd (scmatrix& mU, scmatrix& mVH) throw (cvmexception);`

create object of type `rvector` as vector of **singular values** of calling matrix. The second version sets output parameter `mU` to be equal to matrix `U` of size  $m \times m$  (and change size of the object if it's needed) and `mVH` to be equal to matrix  $V^H$  of size  $n \times n$ . All functions throw **cvmexception** in case of inappropriate sizes of the operands or in case of convergence error. Use `rvector::svd` in order to avoid of new vector creation. Function is *redefined* in the classes `scmatrix`, `scbmatrix`, `schmatrix`. See also `rvector`, `cmatrix`. Example:

`using namespace cvm;``std::cout.setf (std::ios::scientific | std::ios::left);``std::cout.precision (3);``try {`

```
double m[] = {1., -1., 1., 2., -2., 1.,
              3., -2., 1., 0., -2., 1.};
```

`cmatrix mA((std::complex<double>*) m, 2, 3);``cmatrix mSigma(2,3);``rvector v(2);``scmatrix mU(2), mVH(3);``v = mA.svd(mU, mVH);``mSigma.diag(0) = cvector(v);``std::cout << mU << std::endl;``std::cout << mVH << std::endl;``std::cout << mSigma << std::endl;``std::cout << (mA * ~mVH - mU * mSigma).norm() << std::endl;``std::cout << (~mA * mU - ~(mSigma * mVH)).norm() << std::endl;``}``catch (std::exception& e) {``std::cout << "Exception " << e.what () << std::endl;``}``prints`

(-4.861e-01,0.000e+00) (8.739e-01,0.000e+00)  
(7.956e-01,-3.616e-01) (4.425e-01,-2.012e-01)

(-7.590e-02,4.474e-01) (7.488e-01,-1.820e-01) (-4.474e-01,1.327e-02)  
(8.084e-01,1.878e-01) (-1.576e-02,5.238e-01) (-1.878e-01,3.558e-02)  
(1.065e-01,3.065e-01) (3.597e-01,4.669e-02) (8.727e-01,4.012e-02)

(5.452e+00,0.000e+00) (0.000e+00,0.000e+00) (0.000e+00,0.000e+00)  
(0.000e+00,0.000e+00) (1.131e+00,0.000e+00) (0.000e+00,0.000e+00)

1.357e-15

1.267e-15

**2.7.59 pinv**

Functions

```

cmatrix  cmatrix::pinv (TR threshold = cvmMachSp()) const
            throw (cvmexception);
cmatrix& cmatrix::pinv (const basic_rmatrix& mA,
            TR threshold = cvmMachSp())
            throw (cvmexception);

```

implement complex matrix pseudo inversion [6], p. 33 (or Moore-Penrose generalized inversion [5], p. 421). Strictly defined,  $n \times m$  matrix  $A^+$  is *pseudo inversion* of  $m \times n$  matrix  $A$  if the following two equations are satisfied:

$$\begin{aligned}
 AA^+A &= A, \\
 A^+ &= QA^H = A^HP
 \end{aligned}$$

where  $Q$  and  $P$  are some matrices. To compute pseudo inversion, we use [Singular Value Decomposition \(SVD\)](#)

$$A = U\Sigma V^H$$

of matrix  $A$ , thus

$$A^+ = V\Sigma^{-1}U^H,$$

where  $\Sigma^{-1}$  is diagonal  $n \times m$  matrix having inverted diagonal values of matrix  $\Sigma$  if they are greater than some threshold, and zeros otherwise.

First version creates object of type `cmatrix` as pseudo inverted calling matrix (it throws `cvmexception` in case of memory allocation failure). Second function sets calling matrix to be equal to matrix `mA` pseudo inverted (it throws `cvmexception` in case of not appropriate sizes of the operands). The threshold parameter sets minimum distinguishable from zero singular value to be used to compute the pseudo inversion. All values equal or less than the threshold are treated as zeros. Functions are *inherited* in the classes `scmatrix` and `schmatrix` and *redefined* in `scbmatrix`. See also `cmatrix`. Example:

```

using namespace cvm;
try {
    cmatrix mA(2,3);
    mA(1,1) = tcomplex(1.,2.);  mA(1,2) = tcomplex(-1.,3.);
    mA(1,3) = tcomplex(0.,-1.);
    mA(2,1) = tcomplex(1.,-2.); mA(2,2) = tcomplex(1.,-4.);
    mA(2,3) = tcomplex(1.,0.);

    cmatrix mX = mA.pinv();
    std::cout << mX << (mA * mX * mA - mA).norm2() << std::endl;
}
catch (std::exception& e) {

```

```

    std::cout << "Exception " << e.what () << std::endl;
}

prints

(+3.2407407e-01,-1.1111111e-01) (+2.5925926e-01,-1.8518519e-02)
(-1.3888889e-01,-2.7777778e-02) (-5.5555556e-02,+1.6666667e-01)
(+1.4814815e-01,+2.3148148e-01) (+1.6666667e-01,+1.4814815e-01)
+3.3217718e-15

```

Band matrix example:

```

using namespace cvm;

try {
    scbmatrix mA (40, 1, 0);
    mA.diag(0).randomize_real(-1.,1.);
    mA.diag(0).randomize_imag(-3.,2.);
    mA.diag(-1).randomize_real(5.,10.);
    mA.diag(-1).randomize_imag(-3.,7.);

    scmatrix mX (40);
    mX.pinv(mA);
    std::cout << (mA * mX * mA - mA).norm2() << std::endl;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

+6.3113999e-14

```



**2.7.60 gels**

Functions

```

cmatrix  cmatrix::gels (bool conjugate, const cmatrix& mB,
                      cvector& vErr) const throw (cvmexception);
cmatrix& cmatrix::gels (bool conjugate,
                      const cmatrix& mA, const cmatrix& mB,
                      cvector& vErr) throw (cvmexception);
cvector  cmatrix::gels (bool conjugate, const cvector& mB,
                      TC& cErr) const throw (cvmexception);

```

solve overdetermined or underdetermined linear systems

$$A * x = b$$

for  $m \times n$  matrix  $A$  (or its conjugated) where  $b$  is a vector of length  $k$  or systems

$$A * X = B$$

for multiple vectors  $b$  stored as columns of  $k \times l$  matrix  $B$  where  $k = m$  in non-transposed case and  $k = n$  otherwise. The algorithm uses QR or LQ factorization of  $A$ . It is assumed that  $A$  has full rank, infinity returned otherwise. Internally functions use ?GELS LAPACK routines. If  $m > n$  and `conjugate=false` or  $m < n$  and `conjugate=true`, then the system is overdetermined, thus the algorithm tries to find the least squares solution  $x$  of the problem

$$\|A * x - b\|_2 \rightarrow \min \quad \text{or} \quad \|A^H * x - b\|_2 \rightarrow \min$$

respectively. Output vector `vErr` of length  $l$  (or complex number `cErr` for single vector  $b$ ) returns residual sum of squares. The system is underdetermined otherwise, and the algorithm finds its minimum norm solution. In this case `vErr` (or `cErr`) is set to zero. In both cases the solution computed satisfies  $x = \text{pinv}(A) * b$ , but this algorithm is faster than pseudo inversion. Matrix  $A$  is passed as argument `mA` or (in second case) it's calling object. First and third versions return the solution as new matrix or vector object. Second version sets calling object to be the solution. Functions throw `cvmexception` in case of inappropriate sizes of the operands. Functions are *inherited* in all complex matrix classes. See also `cmatrix`, `cvector::gels`, `cmatrix::pinv`. Example:

```
using namespace cvm;
```

```

cmatrix a(7, 5);
cmatrix b(7, 2);
cmatrix bt(5, 2);
cvector bv(5);
cvector vErr(2);
tcomplex dErr;

```

```
a.randomize_real(-10., 10.);
a.randomize_imag(-10., 10.);
b.randomize_real(-10., 10.);
b.randomize_imag(-10., 10.);
bt.randomize_real(-10., 10.);
bt.randomize_imag(-10., 10.);
bv.randomize_real(-10., 10.);
bv.randomize_imag(-10., 10.);

cmatrix x = a.gels(false, b, vErr);
cvector vt = a.gels(true, bv, dErr);
cmatrix xt(7,2);
xt.gels(true, a, bt, vErr);

std::cout << (a.pinv()*b - x).norm() << std::endl;
std::cout << (~a.pinv()*bv - vt).norm() << std::endl;
std::cout << (~a.pinv()*bt - xt).norm() << std::endl;

prints

+1.3258216e-015
+1.0696705e-015
+1.1685394e-015
```

**2.7.61 gelsy**

Functions

```

cmatrix  cmatrix::gelsy (const cmatrix& mB, int& rank,
                        TR tol = cvmMachSp()) const throw (cvmexception);
cmatrix& cmatrix::gelsy (const cmatrix& mA, const cmatrix& mB, int& rank,
                        TR tol = cvmMachSp()) throw (cvmexception);
cvector  cmatrix::gelsy (const cvector& mB, int& rank,
                        TR tol = cvmMachSp()) const throw (cvmexception);

```

compute the minimum-norm solution to the linear least squares problem

$$\|A * x - b\|_2 \rightarrow \min$$

using complete orthogonal factorization of  $m \times n$  matrix  $A$ . Here  $b$  is a vector of length  $m$ . Multiple vectors  $b$  can be stored as columns of  $m \times l$  matrix  $B$ . Matrix  $A$  may be rank-deficient, functions return its effective rank in `rank` output parameter using `tol` tolerance. Internally functions use ?GELSY LAPACK routines, see more details about the algorithm in those routines' documentation. Matrix  $A$  is passed as argument `mA` or (in second case) it's calling object. First and third versions return the solution as new matrix or vector object. Second version sets calling object to be the solution. Functions throw `cvmexception` in case of inappropriate sizes of the operands. Functions are *inherited* in all complex matrix classes. See also `cmatrix`, `cvector::gelsy`, `cmatrix::gelss`, `cmatrix::gelsd`. Example:

```
using namespace cvm;
```

```

cmatrix a(4, 5);
cmatrix b(4, 2);
cvector bv(4);
a.randomize_real(-10., 10.);
a.randomize_imag(-10., 10.);
b.randomize_real(-10., 10.);
b.randomize_imag(-10., 10.);
bv.randomize_real(-10., 10.);
bv.randomize_imag(-10., 10.);
tint rank;

cmatrix x = a.gelsy(b, rank);
cvector xv = a.gelsy(bv, rank);
cmatrix x2(5,2);
x2.gelsy(a, b, rank);

std::cout << (a*x - b).norm() << std::endl;
std::cout << (a*xv - bv).norm() << std::endl;

```

```
std::cout << (a*x2 - b).norm() << std::endl;  
std::cout << rank << " " << a.rank() << std::endl;
```

prints

```
+8.9701623e-015  
+6.9342294e-015  
+8.9701623e-015  
+4 +4
```

**2.7.62 gelss**

Functions

```

cmatrix  cmatrix::gelss (const cmatrix& mB, int& rank, rvector& sv,
                        TR tol = cvmMachSp()) const throw (cvmexception);
cmatrix& cmatrix::gelss (const cmatrix& mA, const cmatrix& mB,
                        rvector& sv, int& rank, TR tol = cvmMachSp())
                        throw (cvmexception);
cvector  cmatrix::gelss (const cvector& mB, rvector& sv, int& rank,
                        TR tol = cvmMachSp()) const throw (cvmexception);

```

compute the minimum-norm solution to the linear least squares problem

$$\|A * x - b\|_2 \rightarrow \min$$

using singular value decomposition of  $m \times n$  matrix  $A$ . Here  $b$  is a vector of length  $m$ . Multiple vectors  $b$  can be stored as columns of  $m \times l$  matrix  $B$ . Matrix  $A$  may be rank-deficient, functions return its effective rank in `rank` output parameter using `tol` tolerance. These functions also compute singular values of  $A$  in decreasing order and return them in `sv` output parameter having  $\min(m, n)$  size. Internally functions use ?GELSS LAPACK routines, see more details about the algorithm in those routines' documentation. Matrix  $A$  is passed as argument `mA` or (in second case) it's calling object. First and third versions return the solution as new matrix or vector object. Second version sets calling object to be the solution. Functions throw `cvmexception` in case of inappropriate sizes of the operands. Functions are *inherited* in all complex matrix classes. See also `cmatrix`, `cvector::gelss`, `cmatrix::gelsy`, `cmatrix::gelsd`. Example:

```

using namespace cvm;

cmatrix a(4, 5);
cmatrix b(4, 2);
cvector bv(4);
rvector sv(4);
tint rank;
a.randomize_real(-10., 10.);
a.randomize_imag(-10., 10.);
b.randomize_real(-10., 10.);
b.randomize_imag(-10., 10.);
bv.randomize_real(-10., 10.);
bv.randomize_imag(-10., 10.);

cmatrix x = a.gelss(b, sv, rank);
cvector xv = a.gelss(bv, sv, rank);
cmatrix xt(5,2);

```

```
xt.gelss(a, b, sv, rank);

std::cout << (a*x - b).norm() << std::endl;
std::cout << (a*xv - bv).norm() << std::endl;
std::cout << (a*xt - b).norm() << std::endl;
std::cout << (sv - a.svd()).norm() << std::endl;
std::cout << rank << " " << a.rank() << std::endl;
```

prints

```
+2.3444861e-013
+4.6335022e-014
+2.3444861e-013
+1.1234667e-014
+4 +4
```

**2.7.63 gelsd**

Functions

```

cmatrix  cmatrix::gelsd (const cmatrix& mB, int& rank, rvector& sv,
                        TR tol = cvmMachSp()) const throw (cvmexception);
cmatrix& cmatrix::gelsd (const cmatrix& mA, const cmatrix& mB,
                        rvector& sv, int& rank, TR tol = cvmMachSp())
                        throw (cvmexception);
cvector  cmatrix::gelsd (const cvector& mB, rvector& sv, int& rank,
                        TR tol = cvmMachSp()) const throw (cvmexception);

```

compute the minimum-norm solution to the linear least squares problem

$$\|A * x - b\|_2 \rightarrow \min$$

using singular value decomposition of  $m \times n$  matrix  $A$  and divide and conquer method. Here  $b$  is a vector of length  $m$ . Multiple vectors  $b$  can be stored as columns of  $m \times l$  matrix  $B$ . Matrix  $A$  may be rank-deficient, functions return its effective rank in `rank` output parameter using `tol` tolerance. These functions also compute singular values of  $A$  in decreasing order and return them in `sv` output parameter having  $\min(m, n)$  size. Internally functions use ?GELSD LAPACK routines, see more details about the algorithm in those routines' documentation. Matrix  $A$  is passed as argument `mA` or (in second case) it's calling object. First and third versions return the solution as new matrix or vector object. Second version sets calling object to be the solution. Functions throw `cvmexception` in case of inappropriate sizes of the operands. Functions are *inherited* in all complex matrix classes. See also `cmatrix`, `cvector::gelsd`, `cmatrix::gelsy`, `cmatrix::gelss`. Example:

```
using namespace cvm;
```

```

cmatrix a(4, 5);
cmatrix b(4, 2);
cvector bv(4);
rvector sv(4);
tint rank;
a.randomize_real(-10., 10.);
a.randomize_imag(-10., 10.);
b.randomize_real(-10., 10.);
b.randomize_imag(-10., 10.);
bv.randomize_real(-10., 10.);
bv.randomize_imag(-10., 10.);

cmatrix x = a.gelsd(b, sv, rank);
cvector xv = a.gelsd(bv, sv, rank);
cmatrix xt(5,2);

```

```
xt.gelsd(a, b, sv, rank);

std::cout << (a*x - b).norm() << std::endl;
std::cout << (a*xv - bv).norm() << std::endl;
std::cout << (a*xt - b).norm() << std::endl;
std::cout << (sv - a.svd()).norm() << std::endl;
std::cout << rank << " " << a.rank() << std::endl;
```

prints

```
+1.2865301e-014
+1.2824878e-014
+1.2865301e-014
+1.2809491e-014
+4 +4
```



### 2.7.64 rank

Function

```
int cmatrix::rank (TR eps = cvmMachSp()) const throw (cvmexception);
```

returns rank of calling matrix as number of **singular values** with **normalized** absolute value greater than or equal to parameter `eps` (this is the **largest relative spacing** by default). Function throws **cvmexception** in case of convergence error. Function is *inherited* in the classes **scmatrix**, **scbmatrix**, **schmatrix**. See also **cmatrix**. Example:

```
using namespace cvm;

double a[] = {1., 2., 3., 4., 5., 6.,
              7., 8., 9., 10., 11., 12.};
cmatrix m(a, NULL, 3, 4);

std::cout << m << m.rank() << std::endl;
m(3,4) = std::complex<double>(0.,1.);
std::cout << m.rank() << std::endl;
```

prints

```
(1,0) (4,0) (7,0) (10,0)
(2,0) (5,0) (8,0) (11,0)
(3,0) (6,0) (9,0) (12,0)
2
3
```

**2.7.65 qr**

Functions

```
void cmatrix::qr (cmatrix& mQ, scmatrix& mR) const throw (cvmexception);
void cmatrix::qr (scmatrix& mQ, cmatrix& mR) const throw (cvmexception);
```

compute QR factorization as

$$M = QR$$

where  $M$  is calling matrix, unitary matrix  $Q$  and upper triangular matrix  $R$  are  $mQ$  and  $mR$  respectively. First version implements so-called "economy" algorithm which for given  $m \times n$  matrix  $M$  computes  $m \times n$  matrix  $Q$  and  $n \times n$  matrix  $R$ . Second version is "full" mode one computing  $m \times m$  matrix  $Q$  and  $m \times n$  matrix  $R$ . Functions throw `cvmexception` in case of inappropriate sizes of the operands. Functions are *redefined* in the class `scmatrix`. See also `cmatrix`, `scmatrix`. Example:

```
using namespace cvm;
```

```
treal ar[] = {1., 2., 3., 4., 5., 6.};
treal ai[] = {1., -1., 2., -2., 3., -3.};
const cvm::cmatrix mh(ar, ai, 2, 3);
const cvm::cmatrix mv(ar, ai, 3, 2);
cvm::scmatrix s2(2), s3(3);
cvm::cmatrix h(2,3), v(3,2);

mh.qr(h,s3);
std::cout << (eye_complex(2)-~cmatrix(h,1,1,2,2)*cmatrix(h,1,1,2,2)).norm()
           << " " << (mh - h * s3).norm() << std::endl;
mh.qr(s2,h);
std::cout << (eye_complex(2) - ~s2 * s2).norm()
           << " " << (mh - s2 * h).norm() << std::endl;
mv.qr(v,s2);
std::cout << (eye_complex(2) - ~v * v).norm()
           << " " << (mv - v * s2).norm() << std::endl;
mv.qr(s3,v);
std::cout << (eye_complex(3) - ~s3 * s3).norm()
           << " " << (mv - s3 * v).norm() << std::endl;
```

prints

```
+2.5145832e-16 +1.3506446e-15
+2.5145832e-16 +1.3506446e-15
+2.5367068e-16 +1.4432899e-15
+3.8435519e-16 +1.4432899e-15
```

**2.7.66 lq**

Functions

```
void cmatrix::lq (scmatrix& mL, cmatrix& mQ) const throw (cvmexception);
void cmatrix::lq (cmatrix& mL, scmatrix& mQ) const throw (cvmexception);
```

compute LQ factorization as

$$M = LQ$$

where  $M$  is calling matrix, upper triangular (trapezoidal) matrix  $L$  and unitary matrix  $Q$  are  $mL$  and  $mQ$  respectively. First version implements so-called "economy" algorithm which for given  $m \times n$  matrix  $M$  computes  $m \times m$  matrix  $L$  and  $m \times n$  matrix  $Q$ . Second version is "full" mode one computing  $m \times n$  matrix  $L$  and  $n \times n$  matrix  $Q$ . Functions throw **cvmexception** in case of inappropriate sizes of the operands. Functions are *redefined* in the class **scmatrix**. See also **cmatrix**, **scmatrix**. Example:

```
using namespace cvm;
```

```
treal ar[] = {1., 2., 3., 4., 5., 6.};
treal ai[] = {1., -1., 2., -2., 3., -3.};
const cvm::cmatrix mh(ar, ai, 2, 3);
const cvm::cmatrix mv(ar, ai, 3, 2);
cvm::scmatrix s2(2), s3(3);
cvm::cmatrix h(2,3), v(3,2);

mh.lq(s2,h);
std::cout << (eye_complex(2) - h * ~h).norm() << " " << (mh - s2 * h)
           .norm() << std::endl;
mv.lq(s3,v);
std::cout << (eye_complex(2) - ~cmatrix(v,1,1,2,2) * cmatrix(v,1,1,2,2))
           .norm() << " " << (mv - s3 * v).norm() << std::endl;
mh.lq(h,s3);
std::cout << (eye_complex(3) - s3 * ~s3).norm() << " " << (mh - h * s3)
           .norm() << std::endl;
mv.lq(v,s2);
std::cout << (eye_complex(2) - s2 * ~s2).norm() << " " << (mv - v * s2)
           .norm() << std::endl;
```

prints

```
+1.3374617e-015 +7.4806297e-015
+6.6728929e-016 +2.9205408e-015
+1.4349634e-015 +7.4806297e-015
+6.7217793e-016 +2.9205408e-015
```

**2.7.67 rq**

Functions

```
void cmatrix::rq (scmatrix& mR, cmatrix& mQ) const throw (cvmexception);
void cmatrix::rq (cmatrix& mR, scmatrix& mQ) const throw (cvmexception);
```

compute RQ factorization as

$$M = RQ$$

where  $M$  is calling matrix, upper triangular matrix  $R$  and unitary matrix  $Q$  are  $mR$  and  $mQ$  respectively. First version implements so-called "economy" algorithm which for given  $m \times n$  matrix  $M$  computes  $m \times m$  matrix  $R$  and  $m \times n$  matrix  $Q$ . Second version is "full" mode one computing  $m \times n$  matrix  $R$  and  $n \times n$  matrix  $Q$ . Following [this definition](#) the implementation assumes that  $m \leq n$  and throws `cvmexception` otherwise. Functions also throw `cvmexception` in case of inappropriate sizes of the operands passed. Functions are *redefined* in the class `srmatrix`. See also `rmatrix`, `srmatrix`. Example:

```
using namespace cvm;
```

```
treal ar[] = {1., 2., 3., 4., 5., 6.};
treal ai[] = {1., -1., 2., -2., 3., -3.};
const cvm::cmatrix mh(ar, ai, 2, 3);
const cvm::cmatrix mv(ar, ai, 3, 2);
cvm::scmatrix s2(2), s3(3);
cvm::cmatrix h(2,3), v(3,2);
```

```
mh.rq(h,s3);
std::cout << (eye_complex(2) - cmatrix(s3,2,1,2,3) * ~cmatrix(s3,2,1,2,3))
            .norm() << " " << (mh - h * s3).norm() << std::endl;
```

```
mh.rq(s2,h);
std::cout << (eye_complex(2) - h * ~h).norm()
            << " " << (mh - s2 * h).norm() << std::endl;
```

prints

```
+3.1898416e-016 +2.2342807e-015
+3.1898416e-016 +2.2342807e-015
```

**2.7.68 ql**

Functions

```
void cmatrix::ql (cmatrix& mQ, scmatrix& mL) const throw (cvmexception);
void cmatrix::ql (scmatrix& mQ, cmatrix& mL) const throw (cvmexception);
```

compute QL factorization as

$$M = QL$$

where  $M$  is calling matrix, unitary matrix  $Q$  and lower triangular matrix  $L$  are  $mQ$  and  $mL$  respectively. First version implements so-called "economy" algorithm which for given  $m \times n$  matrix  $M$  computes  $m \times n$  matrix  $Q$  and  $n \times n$  matrix  $L$ . Second version is "full" mode one computing  $m \times m$  matrix  $Q$  and  $m \times n$  matrix  $L$ . Following [this definition](#) the implementation assumes that  $m \geq n$  and throws `cvmexception` otherwise. Functions also throw `cvmexception` in case of inappropriate sizes of the operands passed. Functions are *redefined* in the class `scmatrix`. See also `cmatrix`, `scmatrix`. Example:

```
using namespace cvm;
```

```
treal ar[] = {1., 2., 3., 4., 5., 6.};
treal ai[] = {1., -1., 2., -2., 3., -3.};
const cvm::cmatrix mv(ar, ai, 3, 2);
cvm::scmatrix s2(2), s3(3);
cvm::cmatrix v(3,2);
```

```
mv.ql(v,s2);
std::cout << (eye_complex(2) - ~v * v).norm()
           << " " << (mv - v * s2).norm() << std::endl;
```

```
mv.ql(s3,v);
std::cout << (eye_complex(2) - ~cmatrix(s3,1,2,3,2) * cmatrix(s3,1,2,3,2))
           .norm() << " " << (mv - s3 * v).norm() << std::endl;
```

prints

```
+1.1857187e-016 +9.4857497e-016
+1.1857187e-016 +9.4857497e-016
```

### 2.7.69 **vanish**

Function

```
cmatrix& cmatrix::vanish();
```

sets every element of calling matrix to be equal to zero and returns reference to the matrix changed. This function is faster than, for example, `cmatrix::set(TC)` with zero operand passed. Function is *redefined* in the classes `scmatrix`, `scbmatrix`, `schmatrix`. See also `cmatrix`. Example:

```
using namespace cvm;
```

```
cmatrix m(4, 3);  
m.randomize_real(0.,1.);  
m.randomize_imag(1.,2.);
```

```
std::cout << m << std::endl;  
std::cout << m.vanish ();
```

prints

```
(0.851527,1.16376) (0.557512,1.90188) (0.0343638,1.52068)  
(0.478042,1.29106) (0.561724,1.19764) (0.320994,1.35804)  
(0.264534,1.40986) (0.113468,1.75137) (0.37727,1.54994)  
(0.521409,1.83035) (0.559465,1.35072) (0.809198,1.12537)
```

```
(0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0)
```

**2.7.70 geru**

Function

```
cmatrix&
cmatrix::geru (TC dAlpha, const cvector& vCol, const cvector& vRow)
throw (cvmexception);
```

calls one of ?GERU routines of the [BLAS library](#) performing **rank-1 update** (unconjugated) matrix-vector operation defined as

$$M = \alpha \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} (y_1 \ y_2 \ \cdots \ y_n) + M,$$

where  $\alpha$  is complex number (parameter dAlpha), M is calling matrix and x and y are complex vectors (parameters vCol and vRow respectively). Function returns reference to the matrix changed and throws **cvmexception** in case of inappropriate sizes of the operands. Function is *inherited* in the class **scmatrix** and *not applicable* to objects of the classes **scbmatrix** and **schmatrix** (i.e. **cvmexception** would be thrown in case of using it for objects of those classes). See also **cvector**, **cmatrix**. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (5);
```

```
std::complex<double> alpha = std::complex<double>(1.2,4.11);
cmatrix m(3,2);
cvector vc(3);
cvector vr(2);
m.randomize_real(-1., 2.); m.randomize_imag(-3., 2.);
vc.randomize_real(-1., 3.); vc.randomize_imag(1., 3.);
vr.randomize_real(0., 2.); vr.randomize_imag(-1., 2.);
```

```
std::cout << m + vc.rank1update_u (vr) * alpha << std::endl;
std::cout << m.geru(alpha, vc, vr);
```

prints

```
(2.88144e+00,3.54299e+00) (-8.14760e+00,-1.03789e+00)
(6.33361e-01,3.35209e+00) (-4.81787e+00,-8.53964e+00)
(5.44811e-01,1.37156e+00) (-5.97006e+00,-5.00794e+00)
```

(2.88144e+00,3.54299e+00) (-8.14760e+00,-1.03789e+00)  
(6.33361e-01,3.35209e+00) (-4.81787e+00,-8.53964e+00)  
(5.44811e-01,1.37156e+00) (-5.97006e+00,-5.00794e+00)



**2.7.71 gerc**

Function

```
cmatrix&
cmatrix::gerc (TC dAlpha, const cvector& vCol, const cvector& vRow)
throw (cvexception);
```

calls one of ?GERC routines of the [BLAS library](#) performing **rank-1 update** (conjugated) matrix-vector operation defined as

$$M = \alpha \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} (y_1^* \ y_2^* \ \dots \ y_n^*) + M,$$

where  $\alpha$  is complex number (parameter dAlpha), M is calling matrix and x and y are complex vectors (parameters vCol and vRow respectively). Function returns reference to the matrix changed and throws **cvexception** in case of inappropriate sizes of the operands. Function is *inherited* in the class **scmatrix** and *not applicable* to objects of the classes **scbmatrix** and **schmatrix** (i.e. **cvexception** would be thrown in case of using it for objects of those classes). See also **cvector**, **cmatrix**. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (5);
```

```
std::complex<double> alpha = std::complex<double>(1.2,4.11);
cmatrix m(3,2);
cvector vc(3);
cvector vr(2);
m.randomize_real(-1., 2.); m.randomize_imag(-3., 2.);
vc.randomize_real(-1., 3.); vc.randomize_imag(1., 3.);
vr.randomize_real(0., 2.); vr.randomize_imag(-1., 2.);
```

```
std::cout << m + vc.rank1update_c (vr) * alpha << std::endl;
std::cout << m.gerc(alpha, vc, vr);
```

prints

```
(1.27138e+01,1.58049e+01) (1.00616e+01,2.21197e+01)
(1.93326e+01,1.41763e+01) (1.74769e+01,2.49013e+01)
(8.09961e+00,1.36259e+01) (5.86738e+00,1.97800e+01)
```

(1.27138e+01,1.58049e+01) (1.00616e+01,2.21197e+01)  
(1.93326e+01,1.41763e+01) (1.74769e+01,2.49013e+01)  
(8.09961e+00,1.36259e+01) (5.86738e+00,1.97800e+01)

**2.7.72 gemm**

Function

```
cmatrix& cmatrix::gemm (const cmatrix& m1, bool bConj1,
                      const cmatrix& m2, bool bConj2,
                      TC dAlpha, TC dBeta) throw (cvmexception);
```

calls one of ?GEMM routines of the [BLAS library](#) performing matrix-matrix operation defined as

$$M = \alpha \mathcal{C}(M_1) \cdot \mathcal{C}(M_2) + \beta M,$$

where  $\alpha$  and  $\beta$  are complex numbers (parameters `dAlpha` and `dBeta`),  $M$  is calling matrix and  $M_1$  and  $M_2$  are matrices (parameters `m1` and `m2` respectively). Function  $\mathcal{C}(M_i)$  conjugates matrix  $M_i$  if appropriate boolean parameter `bConj*` is equal to `true` and does nothing otherwise. Function returns reference to the matrix changed and throws `cvmexception` in case of inappropriate sizes of the operands. Function is *inherited* in the class `scmatrix` and *not applicable* to objects of the classes `scbmatrix` and `schmatrix` (i.e. `cvmexception` would be thrown in case of using it for objects of those classes). See also `cmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (3);

try {
    std::complex<double> alpha = std::complex<double>(1.1,2.1);
    std::complex<double> beta = std::complex<double>(0.71,0.12);
    cmatrix m1(4,3); cmatrix m2(4,3);
    cmatrix m(3,3);
    m.randomize_real(-1., 2.); m.randomize_imag(1., 3.);
    m1.randomize_real(-1., 3.); m1.randomize_imag(-2., 4.);
    m2.randomize_real(0., 2.); m2.randomize_imag(-3., 2.);

    std::cout << ~m1 * m2 * alpha + m * beta << std::endl;
    std::cout << m.gemm(m1, true, m2, false, alpha, beta);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

(3.225e+01,3.611e+01) (2.042e+01,1.206e+01) (5.065e+01,-2.261e+01)
(3.009e+01,3.665e+00) (2.167e+01,-3.327e+00) (4.305e+01,-1.960e+01)
(1.156e+01,-4.966e+00) (4.067e+00,-1.181e+01) (1.121e+01,-2.684e+01)
```

(3.225e+01,3.611e+01) (2.042e+01,1.206e+01) (5.065e+01,-2.261e+01)  
(3.009e+01,3.665e+00) (2.167e+01,-3.327e+00) (4.305e+01,-1.960e+01)  
(1.156e+01,-4.966e+00) (4.067e+00,-1.181e+01) (1.121e+01,-2.684e+01)

**2.7.73 hemm**

Function

```
cmatrix& cmatrix::hemm (bool bLeft, const schmatrix& ms,
                      const cmatrix& m, TC dAlpha, TC dBeta)
                      throw (cvmexception);
```

calls one of ?HEMM routines of the [BLAS library](#) performing one of matrix-matrix operations defined as

$$M = \alpha M_h \cdot M_1 + \beta M \quad \text{or} \quad M = \alpha M_1 \cdot M_h + \beta M,$$

where  $\alpha$  and  $\beta$  are complex numbers (parameters `dAlpha` and `dBeta`),  $M$  is calling matrix,  $M_h$  is hermitian matrix and  $M_1$  is complex matrix (parameters `ms` and `m` respectively). First operation is performed if `bLeft` passed is true and second one otherwise. Function returns reference to the matrix changed and throws `cvmexception` in case of inappropriate sizes of the operands. Function is *inherited* in the classes `scmatrix` and `schmatrix` and *not applicable* to objects of the class `scbmatrix` (i.e. `cvmexception` would be thrown in case of using it for objects of that class). See also `schmatrix`, `cmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (3);

try {
    std::complex<double> alpha = std::complex<double>(1.3,0.21);
    std::complex<double> beta = std::complex<double>(0.5,-0.1);
    cmatrix m1(2,3);
    cmatrix m2(3,2);
    schmatrix ms(2);
    cmatrix m(2,3);
    m.randomize_real(-1., 2.); m.randomize_imag(1., 3.);
    m1.randomize_real(-1., 3.); m1.randomize_imag(1., 2.);
    m2.randomize_real(0., 2.); m2.randomize_imag(-3., -1.);
    ms.randomize_real(-3., 1.); ms.randomize_imag(-1.3, 4.);

    std::cout << ms * m1 * alpha + m * beta << std::endl;
    std::cout << m.hemm (true, ms, m1, alpha, beta) << std::endl;

    m.resize(3,2);
    m.randomize_real(-1.4, 1.3); m.randomize_imag(1.1, 3.);
    std::cout << m2 * ms * alpha + m * beta << std::endl;
    std::cout << m.hemm (false, ms, m2, alpha, beta);
}
catch (std::exception& e) {
```

```
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(1.096e+00,-7.692e+00) (-7.923e+00,-3.909e+00) (-1.324e+01,-5.264e+00)
(2.415e+00,1.240e+00) (4.384e-01,-1.771e+00) (7.495e-01,-2.740e+00)
```

```
(1.096e+00,-7.692e+00) (-7.923e+00,-3.909e+00) (-1.324e+01,-5.264e+00)
(2.415e+00,1.240e+00) (4.384e-01,-1.771e+00) (7.495e-01,-2.740e+00)
```

```
(-5.007e+00,1.010e+01) (2.341e+00,3.248e+00)
(-8.753e+00,7.854e+00) (3.152e+00,4.491e+00)
(-9.162e+00,6.401e+00) (-1.168e+00,3.973e+00)
```

```
(-5.007e+00,1.010e+01) (2.341e+00,3.248e+00)
(-8.753e+00,7.854e+00) (3.152e+00,4.491e+00)
(-9.162e+00,6.401e+00) (-1.168e+00,3.973e+00)
```

### 2.7.74 `randomize_real`

Function

```
cmatrix& cmatrix::randomize_real (TR dFrom, TR dTo);
```

fills real part of calling matrix with pseudo-random numbers distributed between `dFrom` and `dTo`. Function returns reference to the matrix changed. Function is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (3);

cmatrix m(2,3);
m.randomize_real(-1., 2.);
std::cout << m;

prints

(1.090e+00,0.000e+00) (-6.375e-01,0.000e+00) (1.248e+00,0.000e+00)
(-1.272e-01,0.000e+00) (-8.557e-01,0.000e+00) (4.848e-01,0.000e+00)
```

### 2.7.75 `randomize_imag`

Function

```
cmatrix& cmatrix::randomize_imag (TR dFrom, TR dTo);
```

fills imaginary part of calling matrix with pseudo-random numbers distributed between `dFrom` and `dTo`. Function returns reference to the matrix changed. Function is *redefined* in the classes `scmatrix`, `scbmatrix` and `schmatrix`. See also `cmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (3);

cmatrix m(2,3);
m.randomize_imag(-1., 2.);
std::cout << m;

prints

(0.000e+00,1.113e+00) (0.000e+00,6.615e-01) (0.000e+00,1.017e+00)
(0.000e+00,-3.397e-01) (0.000e+00,1.577e+00) (0.000e+00,8.071e-01)
```



## 2.8 srmatrix

This is end-user class encapsulating square matrix of real numbers.

```

template <typename TR>
class srmatrix : public rmatrix <TR>, public SqMatrix <TR,TR> {
public:
    srmatrix ();
    explicit srmatrix (int nMN);
    srmatrix (TR* pD, int nMN);
    srmatrix (const TR* pD, int nMN);
    srmatrix (const srmatrix& m);
    srmatrix (const rmatrix& m);
    explicit srmatrix (const rvector& v);
    srmatrix (rmatrix& m, int nRow, int nCol, int nSize);
    TR& operator () (int im, int in) throw (cvmexception);
    TR operator () (int im, int in) const throw (cvmexception);
    rvector operator () (int i) throw (cvmexception);
    const rvector operator () (int i) const throw (cvmexception);
    rvector operator [] (int i) throw (cvmexception);
    const rvector operator [] (int i) const throw (cvmexception);
    srmatrix& operator = (const srmatrix& m) throw (cvmexception);
    srmatrix& assign (const rvector& v) throw (cvmexception);
    srmatrix& assign (const TR* pD);
    srmatrix& assign (int nRow, int nCol, const rmatrix& m)
        throw (cvmexception);
    srmatrix& set (TR x);
    srmatrix& resize (int nNewMN) throw (cvmexception);
    srmatrix& operator << (const srmatrix& m) throw (cvmexception);
    srmatrix operator + (const srmatrix& m) const
        throw (cvmexception);
    srmatrix operator - (const srmatrix& m) const
        throw (cvmexception);
    srmatrix& sum (const srmatrix& m1,
        const srmatrix& m2) throw (cvmexception);
    srmatrix& diff (const srmatrix& m1,
        const srmatrix& m2) throw (cvmexception);
    srmatrix& operator += (const srmatrix& m) throw (cvmexception);
    srmatrix& operator -= (const srmatrix& m) throw (cvmexception);
    srmatrix operator - () const;
    srmatrix& operator ++ ();
    srmatrix& operator ++ (int);
    srmatrix& operator -- ();

```

```

srmatrix& operator -- (int);
srmatrix operator * (TR d) const;
srmatrix operator / (TR d) const throw (cvmexception);
srmatrix& operator *= (TR d);
srmatrix& operator /= (TR d) throw (cvmexception);
srmatrix& normalize ();
srmatrix operator ~ () const throw (cvmexception);
srmatrix& transpose (const srmatrix& m) throw (cvmexception);
srmatrix& transpose ();
rvector operator * (const rvector& v) const throw (cvmexception);
rmatrix operator * (const rmatrix& m) const throw (cvmexception);
srmatrix operator * (const srmatrix& m) const throw (cvmexception);
srmatrix& operator *= (const srmatrix& m) throw (cvmexception);
srmatrix& swap_rows (int n1, int n2) throw (cvmexception);
srmatrix& swap_cols (int n1, int n2) throw (cvmexception);
rvector solve (const rvector& vB) const throw (cvmexception);
rmatrix solve (const rmatrix& mB) const throw (cvmexception);
rvector solve (const rvector& vB, TR& dErr) const
    throw (cvmexception);
rmatrix solve (const rmatrix& mB, TR& dErr) const
    throw (cvmexception);
rvector solve_tran (const rvector& vB) const throw (cvmexception);
rmatrix solve_tran (const rmatrix& mB) const throw (cvmexception);
rvector solve_tran (const rvector& vB, TR& dErr) const
    throw (cvmexception);
rmatrix solve_tran (const rmatrix& mB, TR& dErr) const
    throw (cvmexception);
rvector operator % (const rvector& vB) const throw (cvmexception);
rvector operator / (const rvector& vB) const throw (cvmexception);
rvector solve_lu (const srmatrix& mLU, const int* pPivots,
    const rvector& vB, TR& dErr) throw (cvmexception);
rvector solve_lu (const srmatrix& mLU, const int* pPivots,
    const rvector& vB) throw (cvmexception);
rmatrix solve_lu (const srmatrix& mLU, const int* pPivots,
    const rmatrix& mB, TR& dErr) throw (cvmexception);
rmatrix solve_lu (const srmatrix& mLU, const int* pPivots,
    const rmatrix& mB) throw (cvmexception);
TR det () const throw (cvmexception);
srmatrix& low_up (const srmatrix& m,
    int* nPivots) throw (cvmexception);
srmatrix low_up (int* nPivots) const throw (cvmexception);
TR cond () const throw (cvmexception);
srmatrix& inv (const srmatrix& mArg) throw (cvmexception);

```

```
srmatrix inv () const throw (cvmexception);
srmatrix& exp (const srmatrix& m,
              TR tol = cvmMachSp ()) throw (cvmexception);
srmatrix exp (TR tol = cvmMachSp ()) const throw (cvmexception);
srmatrix& polynom (const srmatrix& m, const rvector& v)
                throw (cvmexception);
srmatrix polynom (const rvector& v) const throw (cvmexception);
cvector eig (scmatrix& mEigVect,
            bool bRightVect = true) const throw (cvmexception);
cvector eig () const throw (cvmexception);
srmatrix& cholesky (const srmatrix& m) throw (cvmexception);
srmatrix& bunch_kaufman (const srmatrix& m,
                       int* pivots) throw (cvmexception);
void qr(srmatrix& mQ, srmatrix& mR) const throw (cvmexception);
void lq(srmatrix& mL, srmatrix& mQ) const throw (cvmexception);
void rq(srmatrix& mR, srmatrix& mQ) const throw (cvmexception);
void ql(srmatrix& mQ, srmatrix& mL) const throw (cvmexception);
srmatrix& identity ();
srmatrix& vanish ();
srmatrix& randomize (TR dFrom, TR dTo);
};
```

### 2.8.1 `srmatrix ()`

Constructor

```
srmatrix::srmatrix ();
```

creates empty `srmatrix` object. See also `srmatrix`. Example:

```
using namespace cvm;
```

```
srmatrix m;  
std::cout << m.msize() << std::endl << m.nsize() << std::endl;  
std::cout << m.size() << std::endl;
```

```
m.resize (3);  
std::cout << m;
```

prints

```
0  
0  
0  
0 0 0  
0 0 0  
0 0 0
```

### 2.8.2 srmatrix (int)

Constructor

```
explicit srmatrix::srmatrix (int nMN);
```

creates  $n \times n$  srmatrix object where  $n$  is passed in nMN parameter. Constructor sets all elements to zero. Constructor throws `cvmexception` in case of non-positive size passed or memory allocation failure. See also `srmatrix`. Example:

```
using namespace cvm;
```

```
srmatrix m (4);  
std::cout << m.msize() << std::endl << m.nsize()  
          << std::endl << m.size() << std::endl << m;
```

prints

```
4  
4  
16  
0 0 0 0  
0 0 0 0  
0 0 0 0  
0 0 0 0
```

### 2.8.3 srmatrix (TR\*,int)

Constructor

```
srmatrix::srmatrix (TR* pD, int nMN);
```

creates  $n \times n$  srmatrix object where  $n$  is passed in nMN parameter. Unlike others, this constructor *does not allocate memory*. It just shares memory with array pointed to by pD. Constructor throws `cvmexception` in case of non-positive size passed. See also `srmatrix`, `srmatrix (const TR*,int)`. Example:

```
using namespace cvm;
double a[] = {1., 1., 1., 1., 1., 1., 1., 1., 1.};
srmatrix m (a, 3);

m(1,1) = 5.;
std::cout << m << std::endl;
std::cout << a[0] << " " << a[1] << " " << a[2] << std::endl;
```

prints

```
5 1 1
1 1 1
1 1 1

5 1 1
```

### 2.8.4 srmatrix (const TR\*,int)

Constructor

```
srmatrix::srmatrix (const TR* pD, int nMN);
```

creates  $n \times n$  srmatrix object where  $n$  is passed in nMN parameter and copies  $n * n$  elements of array pD to it by columns. Constructor throws `cvmexception` in case of non-positive size passed. See also `srmatrix`, `srmatrix (TR*,int)`. Example:

```
using namespace cvm;
const double a[] = {1., 1., 1., 1., 1., 1., 1., 1., 1.};
srmatrix m (a, 3);

m(1,1) = 5.;
std::cout << m << std::endl;
std::cout << a[0] << " " << a[1] << " " << a[2] << std::endl;
```

prints

```
5 1 1
1 1 1
1 1 1

1 1 1
```

### 2.8.5 srmatrix (const srmatrix&)

Copy constructor

```
srmatrix::srmatrix (const srmatrix& m);
```

creates srmatrix object as a copy of m. Constructor throws `cvmexception` in case of memory allocation failure. See also `srmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};  
srmatrix m(a, 3);  
srmatrix mc(m);
```

```
m(1,1) = 7.77;  
std::cout << m << std::endl << mc;
```

prints

```
7.77 4 7  
2 5 8  
3 6 9
```

```
1 4 7  
2 5 8  
3 6 9
```



### 2.8.6 srmatrix (const rmatrix&)

Constructor

```
srmatrix::srmatrix (const rmatrix& m);
```

creates srmatrix object as a copy of matrix m. It's assumed that  $m \times n$  matrix m must have equal sizes, i.e.  $m = n$  is satisfied. Constructor throws `cvmexception` if this is not true or in case of memory allocation failure. Please note that this constructor is *not explicit* anymore. See also `srmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6.};  
rmatrix m(a, 2, 3);  
std::cout << m << std::endl;
```

```
m.resize(3, 3);  
srmatrix ms (m);  
std::cout << ms;
```

prints

```
1 3 5  
2 4 6
```

```
1 3 5  
2 4 6  
0 0 0
```

### 2.8.7 srmatrix (const rvector&)

Constructor

```
explicit srmatrix::srmatrix (const rvector& v);
```

creates srmatrix object of size `v.size()` by `v.size()` and assigns vector `v` to its main diagonal. Constructor throws `cvmexception` in case of memory allocation failure. See also `srmatrix`, `rvector`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5.};  
rvector v(a, 5);  
srmatrix m(v);  
std::cout << m;
```

prints

```
1 0 0 0 0  
0 2 0 0 0  
0 0 3 0 0  
0 0 0 4 0  
0 0 0 0 5
```

### 2.8.8 submatrix

Submatrix constructor

```
srmatrix::srmatrix (rmatrix& m, int nRow, int nCol, int nSize);
```

creates srmatrix object as *submatrix* of m. It means that the matrix object created shares memory with some part of m. This part is defined by its upper left corner (parameters nRow and nCol, both are *l-based*) and its size (parameter nSize). See also [srmatrix](#). Example:

```
using namespace cvm;
```

```
rmatrix m(4,5);  
srmatrix subm(m, 2, 2, 2);  
subm.set(1.);
```

```
std::cout << m;
```

prints

```
0 0 0 0 0  
0 1 1 0 0  
0 1 1 0 0  
0 0 0 0 0
```

### 2.8.9 operator (,)

Indexing operators

```
TR& srmatrix::operator () (int im, int in) throw (cvmexception);
TR srmatrix::operator () (int im, int in) const throw (cvmexception);
```

provide access to a particular element of calling matrix. The first version is applicable to non-constant object. This version returns *l-value* in order to make possible write access to a particular element. Both operators are *l-based*. Operators throw `cvmexception` if some of parameters passed is outside of `[1,msize()]` range. Operators are *inherited* in the class `srbmatrix` and *redefined* in the class `srsmatrix`. See also `srmatrix`, `Matrix::msize()`, `Matrix::nsize()`. Example:

```
using namespace cvm;

try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
    const srmatrix m (a, 3);
    srmatrix ms(m);

    std::cout << m(1,1) << " " << m(2,3) << std::endl << std::endl;

    ms(2,2) = 7.77;
    std::cout << ms;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

1 8

1 4 7
2 7.77 8
3 6 9
```

**2.8.10 operator ()**

Indexing operators

```
rvector srmatrix::operator () (int i) throw (cvmexception);
const rvector srmatrix::operator () (int i) const throw (cvmexception);
```

provide access to *i*-th column of calling matrix. The first version is applicable to non-constant object and *returns l-value*, i.e. the vector returned shares memory with *i*-th column of calling matrix in order to make possible write access to it. The second version creates *copy* of a column and therefore it's *not l-value*. Both operators are *l-based*. Operators throw **cvmexception** if parameter *i* is outside of [1, nsize()] range. Operators are *redefined* in the classes **srbmatrix** and **srsrmatrix**. See also **srmatrix**. Example:

```
using namespace cvm;

std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
    const srmatrix m (a, 3);
    srmatrix ms(3);

    std::cout << m(2) << std::endl;

    ms(2) = m(3);
    std::cout << ms;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
4.00e+00 5.00e+00 6.00e+00

0.00e+00 7.00e+00 0.00e+00
0.00e+00 8.00e+00 0.00e+00
0.00e+00 9.00e+00 0.00e+00
```

### 2.8.11 operator []

Indexing operators

```
rvector srmatrix::operator [] (int i) throw (cvmexception);
const rvector srmatrix::operator [] (int i) const throw (cvmexception);
```

provide access to *i*-th row of calling matrix. The first version is applicable to non-constant object and *returns l-value*, i.e. the vector returned shares memory with *i*-th row of calling matrix in order to make possible write access to it. The second version creates *copy* of a row and therefore it's *not l-value*. Both operators are *ℓ-based*. Operators throw **cvmexception** if parameter *i* is outside of [1,msize()] range. Operators are *redefined* in the classes **srbmatrix** and **srsrmatrix**. See also **srmatrix**, **Matrix::msize()**. Example:

```
using namespace cvm;

std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
    const srmatrix m (a, 3);
    srmatrix ms(3);

    std::cout << m[2] << std::endl;

    ms[2] = m[3];
    std::cout << ms;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
2.00e+00 5.00e+00 8.00e+00

0.00e+00 0.00e+00 0.00e+00
3.00e+00 6.00e+00 9.00e+00
0.00e+00 0.00e+00 0.00e+00
```

### 2.8.12 operator = (const srmatrix&)

Operator

```
srmatrix& srmatrix::operator = (const srmatrix& m)
throw (cvmexception);
```

sets every element of calling matrix to value of appropriate element of matrix *m* and returns reference to the matrix changed. Operator throws **cvmexception** in case of different sizes of the operands. Operator is *redefined* in the classes **srbmatrix** and **srsrmatrix**. See also **srmatrix**. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
    const srmatrix m1(a, 3);
    srmatrix m2(3);

    m2 = m1;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
1.00e+00 4.00e+00 7.00e+00
2.00e+00 5.00e+00 8.00e+00
3.00e+00 6.00e+00 9.00e+00
```

### 2.8.13 assign (const TR\*)

Function

```
srmatrix& srmatrix::assign (const rvector& v) throw (cvmexception);  
srmatrix& srmatrix::assign (const TR* pD);
```

sets every element of calling matrix to value of appropriate element of vector *v* or array pointed to by *pD* and returns reference to the matrix changed. In first version it's assumed that vector passed is long enough to fill calling matrix. Function throws `cvmexception` otherwise. Function is *redefined* in the classes `srbmatrix` and `srsmatrix`. See also `srmatrix`. Example:

```
using namespace cvm;  
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);  
  
const double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};  
srmatrix m(3);  
  
m.assign(a);  
std::cout << m;  
  
prints  
  
1.00e+00 4.00e+00 7.00e+00  
2.00e+00 5.00e+00 8.00e+00  
3.00e+00 6.00e+00 9.00e+00
```



### 2.8.14 assign (int, int, const rmatrix&)

Function

```
srmatrix& srmatrix::assign (int nRow, int nCol, const rmatrix& m)
throw (cvmexception);
```

sets sub-matrix of calling matrix beginning with *l*-based row nRow and column nCol to matrix m and returns reference to the matrix changed. Function throws *cvmexception* if nRow or nCol are not positive or matrix m doesn't fit. Function is *redefined* in the class *srsrmatrix*. See also *rmatrix*, *srmatrix*. Example:

```
using namespace cvm;
```

```
srmatrix m1(5);
rmatrix m2(2,3);
m1.set(1.);
m2.set(2.);
m1.assign(2,3,m2);
std::cout << m1;
```

prints

```
1 1 1 1 1
1 1 2 2 2
1 1 2 2 2
1 1 1 1 1
1 1 1 1 1
```

### 2.8.15 `set (TR)`

Function

```
srmatrix& srmatrix::set (TR x);
```

sets every element of calling matrix to value of parameter `x` and returns reference to the matrix changed. Use `vanish` to set every element of calling matrix to be equal to zero. Function is *redefined* in the classes `srbmatrix` and `srsmatrix`. See also `srmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
srmatrix m(3);
```

```
m.set(3.);
std::cout << m;
```

prints

```
3.00e+00 3.00e+00 3.00e+00
3.00e+00 3.00e+00 3.00e+00
3.00e+00 3.00e+00 3.00e+00
```

### 2.8.16 `resize`

Function

```
srmatrix& srmatrix::resize (int nNewMN) throw (cvmexception);
```

changes size of calling matrix to `nNewMN` by `nNewMN` and returns reference to the matrix changed. In case of increasing of its size, calling matrix is filled up with zeroes. Function throws `cvmexception` in case of negative size passed or memory allocation failure. Function is *redefined* in the classes `srbmatrix` and `srsmatrix`. See also `srmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4.};
    srmatrix m(a, 2);

    std::cout << m << std::endl;

    m.resize (3);
    std::cout << m;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
1 3
2 4

1 3 0
2 4 0
0 0 0
```

**2.8.17 operator <<**

Operator

```
srmatrix& srmatrix::operator << (const srmatrix& m)
throw (cvmexception);
```

destroys calling matrix, creates new one as a copy of `m` and returns reference to the matrix changed. Operator throws `cvmexception` in case of memory allocation failure. Operator is *redefined* in the classes `srbmatrix` and `srsmatrix`. See also `srmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);

try {
    srmatrix m(3);
    srmatrix mc(1);
    m(1,2) = 1.;
    m(2,3) = 2.;
    std::cout << m << std::endl << mc << std::endl;

    mc << m;
    std::cout << mc;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
0.00e+00 1.00e+00 0.00e+00
0.00e+00 0.00e+00 2.00e+00
0.00e+00 0.00e+00 0.00e+00
```

```
0.00e+00
```

```
0.00e+00 1.00e+00 0.00e+00
0.00e+00 0.00e+00 2.00e+00
0.00e+00 0.00e+00 0.00e+00
```

### 2.8.18 operator +

Operator

```
srmatrix srmatrix::operator + (const srmatrix& m) const  
throw (cvmexception);
```

creates object of type `srmatrix` as sum of calling matrix and matrix `m`. Operator throws `cvmexception` in case of different sizes of the operands. Operator is *redefined* in the classes `srbmatrix` and `srsrmatrix`. See also `srmatrix::sum`, `srmatrix`. Example:

```
using namespace cvm;
```

```
try {  
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};  
    double b[] = {10., 20., 30., 40., 50., 60., 70., 80., 90.};  
    srmatrix m1(a, 3);  
    srmatrix m2(b, 3);  
  
    std::cout << m1 + m2 << std::endl << m1 + m1;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
11 44 77  
22 55 88  
33 66 99
```

```
2 8 14  
4 10 16  
6 12 18
```

**2.8.19 operator -**

Operator

```
srmatrix srmatrix::operator - (const srmatrix& m) const
throw (cvmexception);
```

creates object of type `srmatrix` as difference of calling matrix and matrix `m`. It throws `cvmexception` in case of different sizes of the operands. Operator is *redefined* in the classes `srbmatrix` and `srsrmatrix`. See also `srmatrix::diff`, `srmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
    double b[] = {10., 20., 30., 40., 50., 60., 70., 80., 90.};
    srmatrix m1(a, 3);
    srmatrix m2(b, 3);

    std::cout << m2 - m1 << std::endl << m1 - m1;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
9 36 63
18 45 72
27 54 81
```

```
0 0 0
0 0 0
0 0 0
```

**2.8.20 sum**

Function

```
srmatrix& srmatrix::sum (const srmatrix& m1, const srmatrix& m2)
throw (cvmexception);
```

assigns result of addition of matrices `m1` and `m2` to calling matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. Function is *redefined* in the classes `srbmatrix` and `srsrmatrix`. See also `srmatrix::operator +`, `srmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
    const srmatrix m1(a, 3);
    srmatrix m2(3);
    srmatrix m(3);
    m2.set(1.);

    std::cout << m.sum(m1, m2) << std::endl;
    std::cout << m.sum(m, m2);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
2 5 8
3 6 9
4 7 10

3 6 9
4 7 10
5 8 11
```

**2.8.21 diff**

Function

```
srmatrix& srmatrix::diff (const srmatrix& m1, const srmatrix& m2)
throw (cvmexception);
```

assigns result of subtraction of matrices m1 and m2 to calling matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. Function is *redefined* in the classes `srbmatrix` and `srsrmatrix`. See also `srmatrix::operator -`, `srmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
    const srmatrix m1(a, 3);
    srmatrix m2(3);
    srmatrix m(3);
    m2.set(1.);

    std::cout << m.diff(m1, m2) << std::endl;
    std::cout << m.diff(m, m2);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
0 3 6
1 4 7
2 5 8
```

```
-1 2 5
0 3 6
1 4 7
```



**2.8.22 operator +=**

Operator

```
srmatrix& srmatrix::operator += (const srmatrix& m) throw (cvmexception);
```

adds matrix *m* to calling matrix and returns reference to the matrix changed. It throws **cvmexception** in case of different sizes of the operands. Operator is *redefined* in the classes **srbmatrix** and **srsrmatrix**. See also **srmatrix::operator +** , **srmatrix::sum**, **srmatrix**. Example:

```
using namespace cvm;
try {
    srmatrix m1(3);
    srmatrix m2(3);
    m1.set(1.);
    m2.set(2.);

    m1 += m2;
    std::cout << m1 << std::endl;

    // well, you can do this too, but temporary object would be created
    m2 += m2;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
}
```

prints

```
3 3 3
3 3 3
3 3 3
```

```
4 4 4
4 4 4
4 4 4
```

**2.8.23 operator -=**

Operator

```
srmatrix& srmatrix::operator -= (const srmatrix& m) throw (cvmexception);
```

subtracts matrix *m* from calling matrix and returns reference to the matrix changed. It throws **cvmexception** in case of different sizes of the operands. Operator is *redefined* in the classes **srbmatrix** and **srsmatrix**. See also **srmatrix::operator -** , **srmatrix::diff**, **srmatrix**. Example:

```
using namespace cvm;
```

```
try {
    srmatrix m1(3);
    srmatrix m2(3);
    m1.set(1.);
    m2.set(2.);

    m1 -= m2;
    std::cout << m1 << std::endl;

    // well, you can do this too, but temporary object would be created
    m2 -= m2;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
-1 -1 -1
-1 -1 -1
-1 -1 -1
```

```
0 0 0
0 0 0
0 0 0
```

### 2.8.24 operator - ()

Operator

```
srmatrix srmatrix::operator - () const throw (cvmexception);
```

creates object of type `srmatrix` as calling matrix multiplied by  $-1$ . Operator is *redefined* in the classes `srbmatrix` and `srsrmatrix`. See also `srmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};  
srmatrix m(a, 3);
```

```
std::cout << -m;
```

prints

```
-1 -4 -7  
-2 -5 -8  
-3 -6 -9
```

### 2.8.25 operator ++

Operator

```
srmatrix& srmatrix::operator ++ ();  
srmatrix& srmatrix::operator ++ (int);
```

adds identity matrix to calling matrix and returns reference to the matrix changed. Operator is *redefined* in the classes `srbmatrix` and `srsmatrix`. See also `srmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};  
srmatrix m (a, 3);
```

```
m++;  
std::cout << m << std::endl;  
std::cout << ++m;
```

prints

```
2 4 7  
2 6 8  
3 6 10
```

```
3 4 7  
2 7 8  
3 6 11
```

### 2.8.26 operator --

Operator

```
srmatrix& srmatrix::operator -- ();  
srmatrix& srmatrix::operator -- (int);
```

subtracts identity matrix from calling matrix and returns reference to the matrix changed. Operator is *redefined* in the classes `srbmatrix` and `srsmatrix`. See also `srmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};  
srmatrix m (a, 3);
```

```
m--;  
std::cout << m << std::endl;  
std::cout << --m;
```

prints

```
0 4 7  
2 4 8  
3 6 8
```

```
-1 4 7  
2 3 8  
3 6 7
```

### 2.8.27 operator \* (TR)

Operator

```
srmatrix srmatrix::operator * (TR d) const;
```

creates object of type srmatrix as product of calling matrix and number d. Operator is *redefined* in the classes `srbmatrix` and `srsrmatrix`. See also `srmatrix::operator *=`, `srmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};  
srmatrix m(a, 3);
```

```
std::cout << m * 5.;
```

prints

```
5 20 35  
10 25 40  
15 30 45
```

### 2.8.28 operator / (TR)

Operator

```
srmatrix srmatrix::operator / (TR d) const throw (cvmexception);
```

creates object of type `srmatrix` as quotient of calling matrix and number `d`. It throws `cvmexception` if `d` has absolute value equal or less than the **smallest normalized positive number**. `operator /` is *redefined* in the classes `srbmatrix` and `srsmatrix`. See also `srmatrix::operator /=`, `srmatrix`. Example:

```
using namespace cvm;
```

```
try {  
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};  
    srmatrix m(a, 3);  
  
    std::cout << m / 4.;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
0.25 1 1.75  
0.5 1.25 2  
0.75 1.5 2.25
```

### 2.8.29 operator \*= (TR)

Operator

```
srmatrix& srmatrix::operator *= (TR d);
```

multiplies calling matrix by number d and returns reference to the matrix changed. Operator is *redefined* in the classes `srbmatrix` and `srsmatrix`. See also `srmatrix::operator *`, `srmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};  
srmatrix m(a, 3);
```

```
m *= 2.;  
std::cout << m;
```

prints

```
2 8 14  
4 10 16  
6 12 18
```



### 2.8.30 operator /= (TR)

Operator

```
srmatrix& srmatrix::operator /= (TR d) throw (cvmexception);
```

divides calling matrix by number d and returns reference to the matrix changed. It throws `cvmexception` if d has absolute value equal or less than the `smallest normalized positive number`. Operator is *redefined* in the classes `srbmatrix` and `srsmatrix`. See also `srmatrix::operator /`, `srmatrix`. Example:

```
using namespace cvm;
```

```
try {  
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};  
    srmatrix m(a, 3);  
  
    m /= 2.;  
    std::cout << m;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
0.5 2 3.5  
1 2.5 4  
1.5 3 4.5
```

### 2.8.31 normalize

Function

```
srmatrix& srmatrix::normalize ();
```

normalizes calling matrix so its **Euclidean norm** becomes equal to 1 if it was greater than the **smallest normalized positive number** before the call (otherwise function does nothing). Function is *redefined* in the classes **srbmatrix** and **srsmatrix**. See also **srmatrix**. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (3);
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};  
srmatrix m(a, 3);
```

```
m.normalize();  
std::cout << m << m.norm() << std::endl;
```

prints

```
5.923e-02 2.369e-01 4.146e-01  
1.185e-01 2.962e-01 4.739e-01  
1.777e-01 3.554e-01 5.331e-01  
1.000e+00
```

**2.8.32 transposition**

Operator and functions

```
srmatrix srmatrix::operator ~ () const throw (cvmexception);
srmatrix& srmatrix::transpose (const srmatrix& m) throw (cvmexception);
srmatrix& srmatrix::transpose ();
```

implement matrix transposition. First operator creates object of type srmatrix as transposed calling matrix (it throws **cvmexception** in case of memory allocation failure). Second function sets calling matrix to be equal to matrix m transposed (it throws **cvmexception** in case of not appropriate sizes of the operands), third one makes it to be equal to transposed itself. Functions are *redefined* in the classes **srbmatrix** and **srsrmatrix**. See also **srmatrix**. Example:

```
using namespace cvm;
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
    srmatrix m(a,3);
    srmatrix mt(3);
    std::cout << ~m << std::endl ;
    mt.transpose(m);
    std::cout << mt << std::endl;
    mt.transpose();
    std::cout << mt;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
1 2 3
4 5 6
7 8 9
```

```
1 2 3
4 5 6
7 8 9
```

```
1 4 7
2 5 8
3 6 9
```

### 2.8.33 operator \* (const rvector&)

Operator

```
rvector srmatrix::operator * (const rvector& v) const  
throw (cvmexception);
```

creates object of type rvector as product of calling matrix and vector v. It throws `cvmexception` if number of columns of calling matrix differs from size of vector v. Use `rvector::mult` in order to avoid new object creation. Function is *redefined* in the classes `srbmatrix` and `srsmatrix`. See also `srmatrix`, `rvector`. Example:

```
using namespace cvm;
```

```
try {  
    srmatrix m(3);  
    rvector v(3);  
    m.set(1.);  
    v.set(1.);  
  
    std::cout << m * v;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
3 3 3
```

### 2.8.34 operator \* (const rmatrix&)

Operator

```
rmatrix srmatrix::operator * (const rmatrix& m) const  
throw (cvmexception);
```

creates object of type `rmatrix` as product of calling matrix and matrix `m`. It throws `cvmexception` if number of columns of calling matrix differs from number of rows of matrix `m`. Use `rmatrix::mult` in order to avoid new object creation. Operator is *redefined* in the classes `srbmatrix` and `srsrmatrix`. See also `rmatrix`, `srmatrix`. Example:

```
using namespace cvm;
```

```
try {  
    srmatrix ms(3);  
    rmatrix m(3,2);  
    ms.set(1.);  
    m.set(1.);  
  
    std::cout << ms * m;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
3 3  
3 3  
3 3
```

### 2.8.35 operator \* (const srmatrix&)

Operator

```
srmatrix srmatrix::operator * (const srmatrix& m) const  
throw (cvmexception);
```

creates object of type srmatrix as product of calling matrix and matrix m. It throws **cvmexception** if the operands have different sizes. Use **rmatrix::mult** in order to avoid new object creation. Operator is *inherited* in the class **srbmatrix** and *redefined* in **srsrmatrix**. See also **srmatrix**. Example:

```
using namespace cvm;
```

```
try {  
    srmatrix m1(3);  
    srmatrix m2(3);  
    m1.set(1.);  
    m2.set(1.);  
  
    std::cout << m1 * m2;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
3 3 3  
3 3 3  
3 3 3
```

**2.8.36 operator \*= (const srmatrix&)**

Operator

```
srmatrix& srmatrix::operator *= (const srmatrix& m)
throw (cvmexception);
```

sets calling matrix to be equal to product of itself and square matrix *m* and returns reference to the object it changes. Operator throws **cvmexception** in case of different sizes of the operands. Operator is *inherited* in the class **srbmatrix** and *redefined* in **srsrmatrix**. See also **srmatrix**. Example:

```
using namespace cvm;
```

```
try {
    srmatrix m1(3);
    srmatrix m2(3);
    m1.set(1.);
    m2.set(1.);

    m1 *= m2;
    std::cout << m1 << std::endl;
    m1 *= m1;
    std::cout << m1;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
3 3 3
3 3 3
3 3 3
```

```
27 27 27
27 27 27
27 27 27
```

### 2.8.37 swap\_rows

Function

```
srmatrix& srmatrix::swap_rows (int n1, int n2) throw (cvmexception);
```

swaps two rows of calling matrix and returns reference to the matrix changed. `n1` and `n2` are numbers of rows to be swapped, both are *ℓ*-based). Function throws `cvmexception` if one of the parameters is outside of the range `[1,msize()]`. Function is *not applicable* to objects of the classes `srbmatrix` and `srsmatrix` (i.e. `cvmexception` would be thrown in case of using it for objects of those classes). See also `srmatrix`. Example:

```
using namespace cvm;
```

```
try {  
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};  
    srmatrix m (a, 3);  
  
    std::cout << m << std::endl;  
    std::cout << m.swap_rows(2,3);  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
1 4 7  
2 5 8  
3 6 9
```

```
1 4 7  
3 6 9  
2 5 8
```



### 2.8.38 swap\_cols

Function

```
srmatrix& srmatrix::swap_cols (int n1, int n2) throw (cvmexception);
```

swaps two columns of calling matrix and returns reference to the matrix changed. `n1` and `n2` are numbers of columns to be swapped, both are *ℓ-based*). Function throws `cvmexception` if one of the parameters is outside of the range `[1, nsize()`]. Function is *not applicable* to objects of the classes `srbmatrix` and `srsrmatrix` (i.e. `cvmexception` would be thrown in case of using it for objects of those classes). See also `srmatrix`. Example:

```
using namespace cvm;
```

```
try {  
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};  
    srmatrix m (a, 3);  
  
    std::cout << m << std::endl;  
    std::cout << m.swap_cols(2,3);  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
1 4 7  
2 5 8  
3 6 9
```

```
1 7 4  
2 8 5  
3 9 6
```

**2.8.39 solve**

Functions

```

rvector srmatrix::solve (const rvector& vB) const throw (cvmexception);
rmatrix srmatrix::solve (const rmatrix& mB) const throw (cvmexception);
rvector srmatrix::solve (const rvector& vB, TR& dErr) const
throw (cvmexception);
rmatrix srmatrix::solve (const rmatrix& mB, TR& dErr) const
throw (cvmexception);

```

return solution of linear equation of kind  $A * x = b$  or  $A * X = B$  where  $A$  is calling matrix. The first and third versions solve equation  $A * x = b$  where vector  $b$  is passed in parameter  $vB$  and the second and fourth versions solve equation  $A * X = B$  where matrix  $B$  is passed in parameter  $mB$ . The last two versions also set output parameter  $dErr$  to be equal to the norm of computation error. Functions throw `cvmexception` in case of inappropriate sizes of the operands. Function is *inherited* in the classes `srbmatrix` and `srsrmatrix`. See also `rvector::solve`, `rmatrix::solve`, `rmatrix::solve_tran`, `srmatrix::solve_tran`, `srmatrix`. Example:

```

using namespace cvm;

std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (3);
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 10.};
    srmatrix ma(a, 3);
    rmatrix mb(3,4);
    rmatrix mx(3,4);
    double dErr;

    mb(1).set(1.);
    mb(2).set(2.);
    mb(3).set(3.);
    mb(1,4) = 1.; mb(2,4) = 2.; mb(3,4) = 3.;

    mx = ma.solve (mb, dErr);
    std::cout << mx << dErr
              << std::endl << ma * mx - mb << std::endl;

    rvector vb(3), vx(3);
    vb = mb(2);
    vx = ma.solve (vb, dErr);
    std::cout << vx << dErr << std::endl << ma * vx - vb;
}

```

```
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
-3.333e-01 -6.667e-01 -1.000e+00 1.000e+00  
3.333e-01 6.667e-01 1.000e+00 0.000e+00  
6.661e-16 1.332e-15 0.000e+00 0.000e+00  
3.301e-14  
0.000e+00 0.000e+00 0.000e+00 0.000e+00  
-1.110e-16 -2.220e-16 0.000e+00 0.000e+00  
2.220e-16 4.441e-16 0.000e+00 0.000e+00  
  
-6.667e-01 6.667e-01 1.332e-15  
3.301e-14  
0.000e+00 -2.220e-16 4.441e-16
```

**2.8.40 solve\_tran**

Functions

```

rvector srmatrix::solve_tran (const rvector& vB)
const throw (cvmexception);
rmatrix srmatrix::solve_tran (const rmatrix& mB)
const throw (cvmexception);
rvector srmatrix::solve_tran (const rvector& vB, TR& dErr) const
throw (cvmexception);
rmatrix srmatrix::solve_tran (const rmatrix& mB, TR& dErr) const
throw (cvmexception);

```

return solution of linear equation of kind  $A^T * x = b$  or  $A^T * X = B$  where A is calling matrix. The first and third versions solve equation  $A^T * x = b$  where vector b is passed in parameter vB and the second and fourth versions solve equation  $A^T * X = B$  where matrix B is passed in parameter mB. The last two versions also set output parameter dErr to be equal to the norm of computation error. Functions throw `cvmexception` in case of inappropriate sizes of the operands. Function is *inherited* in the classes `srbmatrix` and `srsrmatrix`. See also `rvector::solve`, `rmatrix::solve`, `srmatrix::solve`, `srmatrix`. Example:

```

using namespace cvm;

std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (3);
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 10.};
    srmatrix ma(a, 3);
    rmatrix mb(3,4);
    rmatrix mx(3,4);
    double dErr;

    mb(1).set(1.);
    mb(2).set(2.);
    mb(3).set(3.);
    mb(1,4) = 1.; mb(2,4) = 2.; mb(3,4) = 3.;

    mx = ma.solve_tran (mb, dErr);
    std::cout << mx << dErr
                << std::endl << ~ma * mx - mb << std::endl;

    rvector vb(3), vx(3);
    vb = mb(2);
    vx = ma.solve_tran (vb, dErr);

```

```
    std::cout << vx << dErr << std::endl << vx * ma - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
-1.000e+000 -2.000e+000 -3.000e+000 -3.333e-001
+1.000e+000 +2.000e+000 +3.000e+000 +6.667e-001
+0.000e+000 +0.000e+000 -1.332e-015 +0.000e+000
+3.513e-014
+0.000e+000 +0.000e+000 +0.000e+000 +0.000e+000
+0.000e+000 +0.000e+000 +8.882e-016 -2.220e-016
+0.000e+000 +0.000e+000 -2.665e-015 +0.000e+000

-2.000e+000 +2.000e+000 +0.000e+000
+3.168e-014
+0.000e+000 +0.000e+000 +0.000e+000
```

**2.8.41 operator % (const rvector&)**

Operator

```
rvector operator % (const rvector& vB) const throw (cvmexception);
```

returns solution  $x$  of linear equation  $A^T * x = b$  (which is equivalent to  $x * A = b$ ) where calling matrix is square matrix  $A$  and vector  $b$  is passed in parameter  $vB$ . This operator throws exception of type `cvmexception` in case of inappropriate sizes of the objects or when matrix  $A$  is close to singular. See also `rvector::solve_tran`, `srmatrix::solve_tran`, `rvector.operator /`, `rvector`, `srmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (12);

try {
    double m[] = {1., -1., 1., 2., -2., 1., 3., -2., 1.};
    double b[] = {1., 2., 3.};
    srmatrix ma(m, 3);
    rvector vb(b, 3);
    rvector vx(3);

    vx = ma % vb;

    std::cout << vx * ma - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
+0.00000000000000e+000 +0.00000000000000e+000 +0.00000000000000e+000
```

**2.8.42 operator / (const rvector&)**

Operator

```
rvector operator / (const rvector& vB) const throw (cvmexception);
```

returns solution  $x$  of linear equation  $A * x = b$  where calling matrix is square matrix  $A$  and vector  $b$  is passed in parameter  $vB$ . This operator throws exception of type `cvmexception` in case of inappropriate sizes of the objects or when matrix  $A$  is close to singular. It's *redefined* in classes `srbmatrix` and `srsmatrix`. See also `rvector::solve`, `srmatrix::solve`, `rvector.operator %`, `rvector`, `srmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (12);

try {
    double m[] = {1., -1., 1., 2., -2., 1., 3., -2., 1.};
    double b[] = {1., 2., 3.};
    srmatrix ma(m, 3);
    rvector vb(b, 3);
    rvector vx(3);

    vx = ma / vb;

    std::cout << ma * vx - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
+0.00000000000000e+000 +0.00000000000000e+000 +0.00000000000000e+000
```

**2.8.43 solve\_lu**

Functions

```
rvector
srmatrix::solve_lu (const srmatrix& mLU, const int* pPivots,
                  const rvector& vB, TR& dErr) throw (cvmexception);
```

```
rvector
srmatrix::solve_lu (const srmatrix& mLU, const int* pPivots,
                  const rvector& vB) throw (cvmexception);
```

```
rmatrix
srmatrix::solve_lu (const srmatrix& mLU, const int* pPivots,
                  const rmatrix& mB, TR& dErr) throw (cvmexception);
```

```
rmatrix
srmatrix::solve_lu (const srmatrix& mLU, const int* pPivots,
                  const rmatrix& mB) throw (cvmexception);
```

create object of type `rvector` or `rmatrix` as solution  $x$  or  $X$  of matrix linear equation  $A * x = b$  or  $A * X = B$  respectively. Here  $A$  is calling matrix, parameter `mLU` is **LU factorization** of matrix  $A$ , parameter `pPivots` is an array of pivot numbers created while factorizing the matrix  $A$  and parameters `vB` and `mB` are vector  $b$  and matrix  $B$  respectively. The first and third version also set output parameter `dErr` to be equal to the norm of computation error. These functions are useful when you need to solve few linear equations of kind  $A * x = b$  or  $A * X = B$  with the same matrix  $A$  and different vectors  $b$  or matrices  $B$ . In such case you save on matrix  $A$  factorization since it's needed to be performed just one time. Functions throw `cvmexception` in case of inappropriate sizes of the operands or when the matrix  $A$  is close to singular. Function is *inherited* in the classes `srbmatrix` and `srsrmatrix`. See also `rvector::solve`, `rmatrix`, `srmatrix`. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (3);
try {
    double a[] = {1., -1., 1., 2., -2., 1., 3., -2., 1.};
    srmatrix ma(a,3);
    srmatrix mLU(3);
    rmatrix mb1(3,2); rvector vb1(3);
    rmatrix mb2(3,2); rvector vb2(3);
    rmatrix mx1(3,2); rvector vx1(3);
    rmatrix mx2(3,2); rvector vx2(3);
    iarray nPivots(3);
    double dErr = 0.;
```



```

mb1.randomize(-1.,3.); vb1.randomize(-2.,4.);
mb2.randomize(-2.,5.); vb2.randomize(-3.,1.);

mLU.low_up(ma, nPivots);
mx1 = ma.solve_lu (mLU, nPivots, mb1, dErr);
std::cout << mx1 << dErr << std::endl;
mx2 = ma.solve_lu (mLU, nPivots, mb2);
std::cout << mx2 << std::endl;;
std::cout << ma * mx1 - mb1 << std::endl << ma * mx2 - mb2;

vx1 = ma.solve_lu (mLU, nPivots, vb1, dErr);
std::cout << vx1 << dErr << std::endl;
vx2 = ma.solve_lu (mLU, nPivots, vb2);
std::cout << vx2 << std::endl;;
std::cout << ma * vx1 - vb1 << std::endl << ma * vx2 - vb2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
prints
2.807e+00 1.107e+00
-3.651e-01 -4.843e+00
-5.412e-01 3.095e+00
6.438e-15
-7.639e-01 1.082e+01
-2.869e-01 -1.110e+01
4.890e-01 3.443e+00

0.000e+00 -4.441e-16
1.110e-16 -4.441e-16
-4.441e-16 4.441e-16

0.000e+00 -4.441e-16
0.000e+00 8.882e-16
0.000e+00 -4.441e-16
-1.651e+00 2.361e-01 -6.384e-02
3.828e-15
-5.886e+00 7.038e+00 -3.125e+00

0.000e+00 0.000e+00 0.000e+00

0.000e+00 0.000e+00 2.220e-16

```

### 2.8.44 det

Function

```
TR srmatrix::det () const throw (cvmexception);
```

returns determinant of calling matrix. It uses the [LU factorization](#) internally and may throw the same exceptions as the factorizer. Function is *inherited* in the classes [srbmatrix](#) and [srsrmatrix](#). See also [srmatrix](#). Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (3);
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 10.};
    srmatrix m(a, 3);

    std::cout << m << std::endl << m.det() << std::endl;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
1.000e+00 4.000e+00 7.000e+00
2.000e+00 5.000e+00 8.000e+00
3.000e+00 6.000e+00 1.000e+01

-3.000e+00
```

**2.8.45 low\_up**

Functions

```
srmatrix&
srmatrix::low_up (const srmatrix& m, int* nPivots) throw (cvmexception);
srmatrix
srmatrix::low_up (int* nPivots) const throw (cvmexception);
```

compute the LU factorization of square matrix A as

$$A = PLU$$

where P is permutation matrix, L is lower triangular matrix with unit diagonal elements and U is upper triangular matrix. All functions store result as the matrix L without main diagonal combined with U. All functions return pivot indices as array of integers (it should support at least `mSize()` elements) pointed to by `nPivots` so *i*-th row was interchanged with `nPivots[i]`-th row. The first version sets calling matrix to be equal to matrix *m*'s LU factorization and the second one creates object of type `srmatrix` as calling matrix LU factorization. Functions throw `cvmexception` in case of inappropriate sizes of the operands or when the matrix to be factorized is close to singular. It is recommended to use `iarray` for pivot values. Function is *redefined* in the class `srbmatrix` and *inherited* in `srsrmatrix`. See also `srmatrix`. Example:

```
using namespace cvm;

std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (3);
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 10.};
    srmatrix m(a, 3);
    srmatrix mLU(3), mLo(3), mUp(3);
    iarray naPivots(3);

    mLU.low_up (m, naPivots);

    mLo.identity ();

    mLo.diag(-2) = mLU.diag(-2);
    mLo.diag(-1) = mLU.diag(-1);
    mUp.diag(0) = mLU.diag(0);
    mUp.diag(1) = mLU.diag(1);
    mUp.diag(2) = mLU.diag(2);

    std::cout << mLo << std::endl << mUp
```

```
        << std::endl << naPivots << std::endl;

    mLU = mLo * mUp;
    for (int i = 3; i >= 1; i--) {
        mLU.swap_rows (i, naPivots[i]);
    }
    std::cout << mLU;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

1.000e+00 0.000e+00 0.000e+00
3.333e-01 1.000e+00 0.000e+00
6.667e-01 5.000e-01 1.000e+00

3.000e+00 6.000e+00 1.000e+01
0.000e+00 2.000e+00 3.667e+00
0.000e+00 0.000e+00 -5.000e-01

3 3 3

1.000e+00 4.000e+00 7.000e+00
2.000e+00 5.000e+00 8.000e+00
3.000e+00 6.000e+00 1.000e+01
```

**2.8.46 cond**

Function

TR srmatrix::cond () const throw (cvmexception);

returns condition number reciprocal of calling matrix  $A$  in the **infinity-norm**:

$$\kappa_{\infty} = \|A\|_{\infty} \|A^{-1}\|_{\infty}.$$

Less value returned means that matrix  $A$  is closer to singular. Zero value returned means estimation underflow or that matrix  $A$  is singular. The condition number is used for error analysis of systems of linear equations. Function throws **cvmexception** in case of LAPACK subroutines failure. Function is *inherited* in the classes **srbmatrix** and **srsmatrix**. See also **srmatrix::solve**, **srmatrix**. Example:

```
using namespace cvm;

std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (3);
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
    srmatrix m(a, 3);
    std::cout << m.cond() << std::endl
              << m.det() << std::endl << std::endl;
    m(3,3) = 10.;
    std::cout << m.cond() << std::endl << m.det() << std::endl;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

0.000e+00
0.000e+00

7.519e-03
-3.000e+00
```

**2.8.47 inv**

Functions

```
srmatrix& srmatrix::inv (const srmatrix& m) throw (cvmexception);
srmatrix srmatrix::inv () const throw (cvmexception);
```

implement matrix inversion. The first version sets calling matrix to be equal to *m* inverted and the second one creates object of type *srmatrix* as inverted calling matrix. Functions throw *cvmexception* in case of inappropriate sizes of the operands or when the matrix to be inverted is close to singular. Function is *redefined* in the class *srsmatrix* and *inherited* in *srbmatrix*. See also *srmatrix*. Example:

```
using namespace cvm;

std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (10);
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 10.};
    srmatrix m(a, 3);
    srmatrix mi(3);
    mi.inv (m);
    std::cout << mi << std::endl << mi * m - eye_real(3);
    std::cout << std::endl << mi.inv() * mi - eye_real(3);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
-6.6666666667e-01 -6.6666666667e-01 1.0000000000e+00
-1.3333333333e+00 3.6666666667e+00 -2.0000000000e+00
1.0000000000e+00 -2.0000000000e+00 1.0000000000e+00

0.0000000000e+00 0.0000000000e+00 1.7763568394e-15
1.7763568394e-15 3.5527136788e-15 0.0000000000e+00
0.0000000000e+00 0.0000000000e+00 1.7763568394e-15

0.0000000000e+00 1.7763568394e-15 -1.7763568394e-15
-8.8817841970e-16 3.5527136788e-15 -3.5527136788e-15
0.0000000000e+00 0.0000000000e+00 -1.7763568394e-15
```

**2.8.48 exp**

Functions

```
srmatrix& srmatrix::exp (const srmatrix& m, TR tol = cvmMachSp ())
throw (cvmexception);
```

```
srmatrix srmatrix::exp (TR tol = cvmMachSp ()) const
throw (cvmexception);
```

compute exponent of square matrix using Padé approximation defined as

$$R_{pq}(z) = D_{pq}(z)^{-1} N_{pq}(z) = 1 + z + \dots + z^p/p!,$$

where

$$N_{pq}(z) = \sum_{k=0}^p \frac{(p+q-k)!p!}{(p+q)!k!(q-k)!} z^k,$$

$$D_{pq}(z) = \sum_{k=0}^q \frac{(p+q-k)!p!}{(p+q)!k!(q-k)!} (-z)^k$$

along with the matrix normalizing as described in [2], p. 572. Functions use DMEXP (or SMEXP for float version) FORTRAN subroutine implementing the algorithm. The first version sets calling matrix to be equal to exponent of *m* and returns reference to the matrix changed. The second version creates object of type *srmatrix* as exponent of calling matrix. The algorithm uses parameter *tol* as  $\varepsilon(p, q)$  in order to choose constants *p* and *q* so that

$$\varepsilon(p, q) \geq 2^{3-(p+q)} \frac{p!q!}{(p+q)!(p+q+1)!}.$$

This parameter is equal to the **largest relative spacing** by default. Functions throw **cvmexception** in case of inappropriate sizes of the operands or when LAPACK subroutine fails. Functions are *inherited* in the classes **srbmatrix** and **srsrmatrix**. The second version is *redefined* in **srbmatrix**. See also **srmatrix**. Example (see [2], p. 567, example 11.2.2):

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (15);
try {
    srmatrix m(2);
    m(1,1) = -49.;
    m(1,2) = 24.;
    m(2,1) = -64.;
```

```
m(2,2) = 31.;

std::cout << m << std::endl << m.exp();
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
-4.9000000000000000e+01 2.4000000000000000e+01
-6.4000000000000000e+01 3.1000000000000000e+01

-7.357587581448284e-01 5.518190996581556e-01
-1.471517599088415e+00 1.103638240715692e+00
```

Matlab output:

```
-7.357587581446907e-001    5.518190996580505e-001
-1.471517599088136e+000    1.103638240715478e+000
```



**2.8.49 polynomial**

Functions

```
srmatrix& srmatrix::polynom (const srmatrix& m, const rvector& v)
throw (cvmexception);
```

```
srmatrix srmatrix::polynom (const rvector& v) const
throw (cvmexception);
```

compute matrix polynomial defined as

$$p(A) = b_0I + b_1A + \dots + b_qA^q$$

using the Horner's rule:

$$p(A) = \sum_{k=0}^r B_k(A^s)^k, \quad s = \text{floor}(\sqrt{q}), \quad r = \text{floor}(q/s)$$

where

$$B_k = \begin{cases} \sum_{i=0}^{s-1} b_{sk+i}A^i, & k = 0, 1, \dots, r-1 \\ \sum_{i=0}^{q-sr} b_{sr+i}A^i, & k = r. \end{cases}$$

See also [2], p. 568. The coefficients  $b_0, b_1, \dots, b_q$  are passed in the parameter  $v$ , where  $q$  is equal to  $v.size() - 1$ , so the functions compute matrix polynomial equal to

$$v[1] * I + v[2] * m + \dots + v[v.size()] * m^{v.size()-1}$$

The first version sets calling matrix to be equal to the polynomial of  $m$  and the second one creates object of type `srmatrix` as the polynomial of calling matrix. Functions use DPOLY (or SPOLY for float version) FORTRAN subroutine implementing the Horner's algorithm. Functions throw `cvmexception` in case of inappropriate sizes of the operands. Functions are *inherited* in the class `srbmatrix` and *redefined* in `srsrmatrix`. See also `srmatrix`. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (15);
try {
    double a[] = {2.2, 1.3, 1.1, -0.9, 0.2,
                  -0.45, 45, -30, 10, 3, 3.2};
    const rvector v(a, 11);
    srmatrix m(2), mp(2);
```

```
m(1,1) = 1.;
m(1,2) = 0.5;
m(2,1) = -1.;
m(2,2) = 0.3;

mp.polynom (m, v);
std::cout << mp;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
prints
-7.963641665999998e+00 -7.551532476200001e+00
1.510306495240000e+01 2.608503800680002e+00
```

Matlab output:

```
-7.963641665999999e+000    -7.551532476200002e+000
1.510306495240000e+001    2.608503800680002e+000
```

**2.8.50 eig**

Functions

```
cvector srmatrix::eig (scmatrix& mEigVect, bool bRightVect = true) const
throw (cvmexception);
```

```
cvector srmatrix::eig () const throw (cvmexception);
```

solve the **nonsymmetric eigenvalue problem** and return complex vector with eigenvalues of calling matrix. The first version sets output parameter `mEigVect` to be equal to square matrix containing right (if parameter `bRightVect` is true, which is default value) or left (if parameter `bRightVect` is false) eigenvectors as columns. All functions throw **cvmexception** in case of inappropriate sizes of the operands or in case of convergence error. Functions are *inherited* in the class **srbmatrix** and *redefined* in **srsrmatrix**. See also **cvector**, **scmatrix** and **srmatrix**. Example:

```
using namespace cvm;
try {
    scmatrix m(3), me(3);
    cvector vl(3);

    m(1,1) = 0.1;  m(1,2) = 0.2;  m(1,3) = 0.1;
    m(2,1) = 0.11; m(2,2) = -2.9; m(2,3) = -8.4;
    m(3,1) = 0.;  m(3,2) = 2.91; m(3,3) = 8.2;

    vl = m.eig (me);
    std::cout << vl;

    m(2,2) = 2.9;
    vl = m.eig (me);
    std::cout << vl << std::endl;

    std::cout.setf (std::ios::scientific | std::ios::showpos);
    std::cout.precision (1);

    std::cout << m * me(1) - me(1) * vl(1);
    std::cout << m * me(2) - me(2) * vl(2);
    std::cout << m * me(3) - me(3) * vl(3);
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
}
```

prints

(-0.0555784,0) (0.285327,0) (5.17025,0)  
(0.0968985,-1.38778e-017) (5.55155,4.1733) (5.55155,-4.1733)

(+1.4e-017,+2.8e-017) (-1.0e-016,-5.4e-017) (+3.9e-017,+4.3e-017)  
(-5.0e-016,+1.7e-016) (-7.1e-015,+2.9e-015) (+3.6e-015,+2.7e-015)  
(-3.1e-016,+2.8e-017) (-2.7e-015,-8.9e-016) (+8.9e-016,+3.6e-015)

### 2.8.51 Cholesky

Function

```
srmatrix& srmatrix::cholesky (const srmatrix& m)
throw (cvmexception);
```

forms the Cholesky factorization of symmetric positive-definite matrix A defined as

$$A = U^T U,$$

where U is upper triangular matrix. It utilizes one of ?POTRF routines of the [LAPACK library](#). Function sets calling matrix to be equal to the factorization of symmetric positive-definite matrix m. Function throws [cvmexception](#) in case of inappropriate sizes of the operands or in case of convergence error. See also [srmatrix](#) and [srsrmatrix](#). Example:

```
using namespace cvm;

try {
    double a[] = {1., 2., 1., 2., 5., -1., 1., -1., 20.};
    const srmatrix m(a, 3);
    srmatrix h(3);

    h.cholesky(m);
    std::cout << h << std::endl;
    std::cout << ~h * h - m;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}

prints

1 2 1
0 1 -3
0 0 3.16228

0 0 0
0 0 0
0 0 0
```

### 2.8.52 Bunch-Kaufman

Function

```
srmatrix& srmatrix::bunch_kaufman (const srsrmatrix& m, int* pivots)
throw (cvmexception);
```

forms the Bunch-Kaufman factorization of symmetric matrix (cited from the MKL library documentation):

$$A = PUDU^T P^T,$$

where  $A$  is the input matrix passed in parameter  $m$ ,  $P$  is a permutation matrix,  $U$  and  $L$  are upper and lower triangular matrices with unit diagonal, and  $D$  is a symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks.  $U$  and  $L$  have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of  $D$ . It utilizes one of ?SYTRF routines of the [LAPACK library](#). Function sets calling matrix to be equal to the factorization of symmetric positive-definite matrix  $m$ . Function throws `cvmexception` in case of inappropriate sizes of the operands or in case of convergence error. See also `srmatrix` and `srsrmatrix`. Function is mostly designed to be used for subsequent calls of ?SYTRS, ?SYCON and ?SYTRI routines of the [LAPACK library](#). Currently it's used internally in `srmatrix::det` flow when argument is symmetric but not positive-definite.

**2.8.53 qr**

Function

```
void srmatrix::qr (srmatrix& mQ, srmatrix& mR) const throw (cvmexception);
```

computes QR factorization as

$$M = QR$$

where  $M$  is calling square matrix, orthogonal matrix  $Q$  and upper triangular matrix  $R$  are  $mQ$  and  $mR$  respectively. Function throws `cvmexception` in case if inappropriate sizes of the operands passed. See also `rmatrix`, `rmatrix::qr`, `srmatrix`. Example:

```
using namespace cvm;
```

```
treal a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
```

```
const cvm::srmatrix m(a, 3);
```

```
cvm::srmatrix q(3), r(3);
```

```
m.qr(q,r);
```

```
std::cout << (eye_real(3) - ~q * q).norm()
```

```
<< " " << (m - q * r).norm() << std::endl;
```

prints

```
+5.2889959e-16 +7.0854500e-15
```

**2.8.54 lq**

Function

```
void srmatrix::lq (srmatrix& mL, srmatrix& mQ) const throw (cvmexception);
```

computes LQ factorization as

$$M = LQ$$

where  $M$  is calling square matrix, lower triangular matrix  $L$  and orthogonal matrix  $Q$  are  $mL$  and  $mQ$  respectively. Function throws `cvmexception` in case if inappropriate sizes of the operands passed. See also `rmatrix`, `rmatrix::lq`, `srmatrix`. Example:

```
using namespace cvm;
```

```
treal a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
```

```
const cvm::srmatrix m(a, 3);
```

```
cvm::srmatrix l(3), q(3);
```

```
m.lq(l,q);
```

```
std::cout << (eye_real(3) - ~q * q).norm()
```

```
    << " " << (m - l * q).norm() << std::endl;
```

prints

```
+7.3329369e-016 +8.1523942e-015
```



**2.8.55 ql**

Function

```
void srmatrix::ql (srmatrix& mQ, srmatrix& mL) const throw (cvmexception);
```

computes QL factorization as

$$M = QL$$

where  $M$  is calling square matrix, orthogonal matrix  $Q$  and lower triangular matrix  $L$  are  $mQ$  and  $mL$  respectively. Function throws `cvmexception` in case if inappropriate sizes of the operands passed. See also `rmatrix`, `rmatrix::ql`, `srmatrix`. Example:

```
using namespace cvm;
```

```
treal a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
```

```
const cvm::srmatrix m(a, 3);
```

```
cvm::srmatrix q(3), l(3);
```

```
m.ql(q,l);
```

```
std::cout << (eye_real(3) - ~q * q).norm()
```

```
    << " " << (m - q * l).norm() << std::endl;
```

prints

```
+1.6146017e-015 +4.3341378e-015
```

**2.8.56 rq**

Function

```
void srmatrix::rq (srmatrix& mR, srmatrix& mQ) const throw (cvmexception);
```

computes RQ factorization as

$$M = RQ$$

where  $M$  is calling square matrix, upper triangular matrix  $R$  and orthogonal matrix  $Q$  are  $mR$  and  $mQ$  respectively. Function throws `cvmexception` in case if inappropriate sizes of the operands passed. See also `rmatrix`, `rmatrix::rq`, `srmatrix`. Example:

```
using namespace cvm;
```

```
treal a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
```

```
const cvm::srmatrix m(a, 3);
```

```
cvm::srmatrix r(3), q(3);
```

```
m.rq(r,q);
```

```
std::cout << (eye_real(3) - ~q * q).norm()
```

```
    << " " << (m - r * q).norm() << std::endl;
```

```
prints
```

```
+3.7030557e-016 +2.2752801e-015
```

### 2.8.57 identity

Function

```
srmatrix& srmatrix::identity();
```

sets calling matrix to be equal to identity matrix and returns reference to the matrix changed. Function is *redefined* in the classes `srbmatrix` and `srsmatrix`. See also `srmatrix`. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (3);  
srmatrix m(3);  
m.randomize(0.,1.);
```

```
std::cout << m << std::endl;  
std::cout << m.identity();
```

prints

```
9.423e-01 2.950e-01 8.429e-01  
2.013e-01 3.250e-01 2.904e-01  
7.920e-01 2.405e-02 7.801e-01  
  
1.000e+00 0.000e+00 0.000e+00  
0.000e+00 1.000e+00 0.000e+00  
0.000e+00 0.000e+00 1.000e+00
```

### 2.8.58 vanish

Function

```
srmatrix& srmatrix::vanish();
```

sets every element of calling matrix to be equal to zero and returns reference to the matrix changed. This function is faster than `srmatrix::set(TR)` with zero operand passed. Function is *redefined* in the classes `srsrmatrix` and `srbmatrix`. See also `srmatrix`. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (3);  
srmatrix m(3);  
m.randomize(0.,1.);
```

```
std::cout << m << std::endl;  
std::cout << m.vanish ();
```

prints

```
1.747e-01 7.563e-01 5.163e-01  
9.657e-01 6.619e-01 8.036e-01  
6.392e-01 6.658e-01 6.495e-01  
  
0.000e+00 0.000e+00 0.000e+00  
0.000e+00 0.000e+00 0.000e+00  
0.000e+00 0.000e+00 0.000e+00
```

### 2.8.59 **randomize**

Function

```
srmatrix& srmatrix::randomize (TR dFrom, TR dTo);
```

fills calling matrix with pseudo-random numbers distributed between dFrom and dTo. Function returns reference to the matrix changed. See also [srmatrix](#). Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (7);
```

```
srmatrix m(3);
m.randomize(-2.,3.);
std::cout << m;
```

prints

```
-1.6790979e+00 5.0233467e-02 -1.9559008e+00
-1.7987609e-01 -5.2092044e-01 -1.8211615e+00
6.8242439e-01 9.0688803e-01 -1.7891171e+00
```

## 2.9 *scmatrix*

This is end-user class encapsulating square matrix of complex numbers.

```

template <typename TR, typename TC>
class scmatrix : public cmatrix <TR,TC>, public SqMatrix <TR,TC> {
public:
    scmatrix ();
    explicit scmatrix (int nMN);
    scmatrix (TC* pD, int nMN);
    scmatrix (const TC* pD, int nMN);
    scmatrix (const scmatrix& m);
    explicit scmatrix (const cmatrix& m);
    explicit scmatrix (const cvector& v);
    explicit scmatrix (const srmatrix& m, bool bRealPart = true);
    scmatrix (const TR* pRe, const TR* pIm, int nMN);
    scmatrix (const srmatrix& mRe, const srmatrix& mIm);
    scmatrix (cmatrix& m, int nRow, int nCol, int nSize);
    TC& operator () (int im, int in) throw (cvmexception);
    TC operator () (int im, int in) const throw (cvmexception);
    cvector operator () (int i) throw (cvmexception);
    const cvector operator () (int i) const throw (cvmexception);
    cvector operator [] (int i) throw (cvmexception);
    const cvector operator [] (int i) const throw (cvmexception);
    const srmatrix real () const;
    const srmatrix imag () const;
    scmatrix& operator = (const scmatrix& m) throw (cvmexception);
    scmatrix& assign (const cvector& v) throw (cvmexception);
    scmatrix& assign (const TC* pD);
    scmatrix& assign (int nRow, int nCol, const cmatrix& m)
        throw (cvmexception);
    scmatrix& set (TC x);
    scmatrix& assign_real (const srmatrix& mRe) throw (cvmexception);
    scmatrix& assign_imag (const srmatrix& mIm) throw (cvmexception);
    scmatrix& set_real (TR d);
    scmatrix& set_imag (TR d);
    scmatrix& resize (int nNewMN) throw (cvmexception);
    scmatrix& operator << (const scmatrix& m) throw (cvmexception);
    scmatrix operator + (const scmatrix& m) const
        throw (cvmexception);
    scmatrix operator - (const scmatrix& m) const
        throw (cvmexception);
    scmatrix& sum (const scmatrix& m1,

```

```

        const scmatrix& m2) throw (cvmexception);
scmatrix& diff (const scmatrix& m1,
               const scmatrix& m2) throw (cvmexception);
scmatrix& operator += (const scmatrix& m) throw (cvmexception);
scmatrix& operator -= (const scmatrix& m) throw (cvmexception);
scmatrix operator - () const;
scmatrix& operator ++ ();
scmatrix& operator ++ (int);
scmatrix& operator -- ();
scmatrix& operator -- (int);
scmatrix operator * (TR d) const;
scmatrix operator / (TR d) const throw (cvmexception);
scmatrix operator * (TC z) const;
scmatrix operator / (TC z) const throw (cvmexception);
scmatrix& operator *= (TR d);
scmatrix& operator /= (TR d) throw (cvmexception);
scmatrix& operator *= (TC z);
scmatrix& operator /= (TC z) throw (cvmexception);
scmatrix& normalize ();
scmatrix operator ~ () const;
scmatrix operator ! () const;
scmatrix& conj (const scmatrix& m) throw (cvmexception);
scmatrix& conj ();
scmatrix& transpose (const scmatrix& m) throw (cvmexception);
scmatrix& transpose ();
cvector operator * (const cvector& v) const
                throw (cvmexception);
cmatrix operator * (const cmatrix& m) const
                throw (cvmexception);
scmatrix operator * (const scmatrix& m) const
                throw (cvmexception);
scmatrix& operator *= (const scmatrix& m)
                throw (cvmexception);
scmatrix& swap_rows (int n1, int n2) throw (cvmexception);
scmatrix& swap_cols (int n1, int n2) throw (cvmexception);
cvector solve (const cvector& vB) const throw (cvmexception);
cmatrix solve (const cmatrix& mB) const throw (cvmexception);
cvector solve (const cvector& vB, TR& dErr) const
                throw (cvmexception);
cmatrix solve (const cmatrix& mB, TR& dErr) const
                throw (cvmexception);
cvector solve_tran (const cvector& vB) const throw (cvmexception);
cmatrix solve_tran (const cmatrix& mB) const throw (cvmexception);

```

```

cvector solve_tran (const cvector& vB, TR& dErr) const
    throw (cvmexception);
cmatrix solve_tran (const cmatrix& mB, TR& dErr) const
    throw (cvmexception);
cvector solve_conj (const cvector& vB) const throw (cvmexception);
cmatrix solve_conj (const cmatrix& mB) const throw (cvmexception);
cvector solve_conj (const cvector& vB, TR& dErr) const
    throw (cvmexception);
cmatrix solve_conj (const cmatrix& mB, TR& dErr) const
    throw (cvmexception);
cvector operator % (const cvector& vB) const throw (cvmexception);
cvector operator / (const cvector& vB) const throw (cvmexception);
cvector solve_lu (const scmatrix& mLU, const int* pPivots,
    const cvector& vB, TR& dErr) throw (cvmexception);
cvector solve_lu (const scmatrix& mLU, const int* pPivots,
    const cvector& vB) throw (cvmexception);
cmatrix solve_lu (const scmatrix& mLU, const int* pPivots,
    const cmatrix& mB, TR& dErr) throw (cvmexception);
cmatrix solve_lu (const scmatrix& mLU, const int* pPivots,
    const cmatrix& mB) throw (cvmexception);
TC det () const throw (cvmexception);
scmatrix& low_up (const scmatrix& m,
    int* nPivots) throw (cvmexception);
scmatrix low_up (int* nPivots) const throw (cvmexception);
TR cond () const throw (cvmexception);
scmatrix& inv (const scmatrix& mArg) throw (cvmexception);
scmatrix inv () const throw (cvmexception);
scmatrix& exp (const scmatrix& mArg, TR tol = cvmMachSp ())
    throw (cvmexception);
scmatrix exp (TR tol = cvmMachSp ()) const throw (cvmexception);
scmatrix& polynom (const scmatrix& m, const cvector& v)
    throw (cvmexception);
scmatrix polynom (const cvector& v) const
    throw (cvmexception);
cvector eig (scmatrix& mEigVect, bool bRightVect = true) const
    throw (cvmexception);
cvector eig () const throw (cvmexception);
scmatrix& cholesky (const schmatrix& m) throw (cvmexception);
scmatrix& bunch_kaufman (const schmatrix& m,
    int* pivots) throw (cvmexception);
void qr(scmatrix& mQ, scmatrix& mR) const throw (cvmexception);
void lq(scmatrix& mL, scmatrix& mQ) const throw (cvmexception);
void rq(scmatrix& mR, scmatrix& mQ) const throw (cvmexception);

```



```
void ql(scmatrix& mQ, scmatrix& mL) const throw (cvmexception);  
scmatrix& identity ();  
scmatrix& vanish ();  
scmatrix& randomize_real (TR dFrom, TR dTo);  
scmatrix& randomize_imag (TR dFrom, TR dTo);  
};
```

### 2.9.1 `scmatrix ()`

Constructor

```
scmatrix::scmatrix ();
```

creates empty `scmatrix` object. See also `scmatrix`. Example:

```
using namespace cvm;
```

```
scmatrix m;  
std::cout << m.msize() << std::endl  
           << m.nsize() << std::endl  
           << m.size() << std::endl;  
m.resize(3);  
std::cout << m;
```

prints

```
0  
0  
0  
(0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0)
```

### 2.9.2 *scmatrix* (int)

Constructor

```
explicit scmatrix::scmatrix (int nMN);
```

creates  $n \times n$  *scmatrix* object where  $n$  is passed in *nMN* parameter. Constructor throws *cvmexception* in case of non-positive size passed or memory allocation failure. See also *scmatrix*. Example:

```
using namespace cvm;
```

```
scmatrix m (4);  
std::cout << m.msize() << std::endl  
          << m.nsize() << std::endl  
          << m.size() << std::endl << m;
```

prints

```
4  
4  
16  
(0,0) (0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0) (0,0)
```

**2.9.3 *scmatrix* (TC\*,int)**

Constructor

```
scmatrix::scmatrix (TC* pD, int nMN);
```

creates  $n \times n$  *scmatrix* object where  $n$  is passed in *nMN* parameter. Unlike others, this constructor *does not allocate memory*. It just shares memory with array pointed to by *pD*. Constructor throws *cvmexception* in case of non-positive size passed. See also *scmatrix*, *scmatrix (const TC\*,int)*. Example:

```
using namespace cvm;
double a[] = {1., 1., 1., 1., 1., 1., 1., 1., 1.,
              1., 1., 1., 1., 1., 1., 1., 1., 1.};
scmatrix m ((std::complex<double>*) a, 3);

m(1,1) = std::complex<double>(2.,3.);
std::cout << m << std::endl;
std::cout << a[0] << " " << a[1] << " " << a[2] << " "
          << a[3] << " " << a[4] << " " << a[5] << " " << std::endl;
```

prints

```
(2,3) (1,1) (1,1)
(1,1) (1,1) (1,1)
(1,1) (1,1) (1,1)
```

```
2 3 1 1 1 1
```

**2.9.4 *s*cmatrix (const TC\*,int)**

Constructor

```
scmatrix::scmatrix (const TC* pD, int nMN);
```

creates  $n \times n$  *s*cmatrix object where  $n$  is passed in *nMN* parameter and copies  $n * n$  elements of array *pD* to it by **columns**. Constructor throws **cvmexception** in case of non-positive size passed. See also **s**cmatrix, **s**cmatrix (TC\*,int). Example:

```
using namespace cvm;
const double a[] = {1., 1., 1., 1., 1., 1., 1., 1., 1.,
                    1., 1., 1., 1., 1., 1., 1., 1., 1.};
scmatrix m ((const std::complex<double>*) a, 3);

m(1,1) = std::complex<double>(2.,3.);
std::cout << m << std::endl;
std::cout << a[0] << " " << a[1] << " " << a[2] << " "
          << a[3] << " " << a[4] << " " << a[5] << " " << std::endl;
```

prints

```
(2,3) (1,1) (1,1)
(1,1) (1,1) (1,1)
(1,1) (1,1) (1,1)
```

```
1 1 1 1 1 1
```

### 2.9.5 `scmatrix (const scmatrix&)`

Copy constructor

```
scmatrix::scmatrix (const scmatrix& m)
```

creates `scmatrix` object as a copy of `m`. Constructor throws `cvmexception` in case of memory allocation failure. See also `scmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
scmatrix m ((std::complex<double>*) a, 2);  
scmatrix mc(m);
```

```
m(1,1) = std::complex<double>(7.77,7.77);  
std::cout << m << std::endl << mc;
```

prints

```
(7.77,7.77) (5,6)  
(3,4) (7,8)
```

```
(1,2) (5,6)  
(3,4) (7,8)
```

**2.9.6 *s*cmatrix (const *c*matrix&)**

Constructor

```
explicit scmatrix::scmatrix (const cmatrix& m)
```

creates *s*cmatrix object as a copy of matrix *m*. It's assumed that  $m \times n$  matrix *m* must have equal sizes, i.e.  $m = n$  is satisfied. Constructor throws ***cv*mexception** if this is not true or in case of memory allocation failure. Please note that this constructor is *not explicit* anymore. See also ***s*cmatrix**. Example:

```
using namespace cvm;

double a[] = {1., 2., 3., 4., 5., 6.,
              7., 8., 9., 10., 11., 12.};
cmatrix m((std::complex<double>*) a, 2, 3);
std::cout << m << std::endl;

m.resize(3, 3);
scmatrix ms (m);
std::cout << ms;
```

prints

```
(1,2) (5,6) (9,10)
(3,4) (7,8) (11,12)

(1,2) (5,6) (9,10)
(3,4) (7,8) (11,12)
(0,0) (0,0) (0,0)
```

### 2.9.7 *scmatrix* (const *cvector*&)

Constructor

```
explicit scmatrix::scmatrix (const cvector& v);
```

creates *scmatrix* object of size `v.size()` by `v.size()` and assigns vector `v` to its main diagonal. Constructor throws `cvmexception` in case of memory allocation failure. See also `scmatrix`, `cvector`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
cvector v((std::complex<double>*) a, 4);  
scmatrix m(v);  
std::cout << m;
```

prints

```
(1,2) (0,0) (0,0) (0,0)  
(0,0) (3,4) (0,0) (0,0)  
(0,0) (0,0) (5,6) (0,0)  
(0,0) (0,0) (0,0) (7,8)
```



### 2.9.8 *s*cmatrix (const *s*rmatrix&,bool)

Constructor

```
explicit scmatrix::scmatrix (const srmatrix& m, bool bRealPart = true);
```

creates *s*cmatrix object having the same dimension as real matrix *m* and copies matrix *m* to its real part if *bRealPart* is true or to its imaginary part otherwise. See also [s](#)cmatrix, [s](#)rmatrix. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};  
const srmatrix m(a, 3);  
scmatrix mr(m), mi(m, false);  
std::cout << mr << std::endl << mi;
```

prints

```
(1,0) (4,0) (7,0)  
(2,0) (5,0) (8,0)  
(3,0) (6,0) (9,0)  
  
(0,1) (0,4) (0,7)  
(0,2) (0,5) (0,8)  
(0,3) (0,6) (0,9)
```

**2.9.9 *scmatrix* (const TR\*,const TR\*,int)**

Constructor

```
scmatrix::scmatrix (const TR* pRe, const TR1* pIm, int nMN);
```

creates *scmatrix* object of size nMN by nMN and copies every element of arrays pointed to by pRe and pIm to real and imaginary part of the matrix created respectively. Use NULL pointer to fill up appropriate part with zero values. Constructor throws *cvmexception* in case of memory allocation failure. See also *scmatrix*. Example:

```
using namespace cvm;
```

```
double re[] = {1., 2., 3., 4.};
double im[] = {4., 3., 2., 1.};
scmatrix m(re, im, 2);
std::cout << m << std::endl;
re[0] = 7.777;
std::cout << m << std::endl;
```

```
const double rec[] = {1., 2., 3., 4.};
const scmatrix mc (rec, NULL, 2);
std::cout << mc;
```

prints

```
(1,4) (3,2)
(2,3) (4,1)
```

```
(1,4) (3,2)
(2,3) (4,1)
```

```
(1,0) (3,0)
(2,0) (4,0)
```

**2.9.10 *scmatrix* (const *srmatrix*&, const *srmatrix*&)**

Constructor

```
scmatrix::scmatrix (const srmatrix& mRe, const srmatrix& mIm);
```

creates *scmatrix* object of the same size as *mRe* and *mIm* has (it throws *cvmexception* if *mRe* and *mIm* have different sizes) and copies matrices *mRe* and *mIm* to real and imaginary part of the matrix created. Constructor throws *cvmexception* in case of memory allocation failure. See also *scmatrix*, *srmatrix*. Example:

```
using namespace cvm;  
  
srmatrix mr(3), mi(3);  
mr.set(1.);  
mi.set(2.);  
const scmatrix mc(mr, mi);  
std::cout << mc;
```

prints

```
(1,2) (1,2) (1,2)  
(1,2) (1,2) (1,2)  
(1,2) (1,2) (1,2)
```

### 2.9.11 **submatrix**

Submatrix constructor

```
scmatrix::scmatrix (cmatrix& m, int nRow, int nCol, int nSize);
```

creates *scmatrix* object as *submatrix* of *m*. It means that the matrix object created shares memory with some part of *m*. This part is defined by its upper left corner (parameters *nRow* and *nCol*, both are *l-based*) and its size (parameter *nSize*). See also *scmatrix*. Example:

```
using namespace cvm;
```

```
cmatrix m(4,5);  
scmatrix subm(m,2,2,2);  
subm.set(std::complex<double>(1.,2.));  
std::cout << m;
```

prints

```
(0,0) (0,0) (0,0) (0,0) (0,0)  
(0,0) (1,2) (1,2) (0,0) (0,0)  
(0,0) (1,2) (1,2) (0,0) (0,0)  
(0,0) (0,0) (0,0) (0,0) (0,0)
```

**2.9.12 operator (,)**

Indexing operators

```
TC& scmatrix::operator () (int im, int in) throw (cvmexception);
TC scmatrix::operator () (int im, int in) const throw (cvmexception);
```

provide access to a particular element of calling matrix. The first version of operator is applicable to non-constant object. This version returns *l-value* in order to make possible write access to an element. Both operators are *l-based*. Operators throw **cvmexception** if some of parameters passed is outside of [1,msize()] range. Operators are *inherited* in the class **scmatrix** and *redefined* in the class **schmatrix**. See also **scmatrix**, **Matrix::msize()**, **Matrix::nsize()**. Example:

```
using namespace cvm;

scmatrix m (3);
m.set(std::complex<double>(1.,2.));
std::cout << m(1,1) << std::endl;

m(2,2) = std::complex<double>(7.77,7.77);
std::cout << m;
```

prints

```
(1,2)
(1,2) (1,2) (1,2)
(1,2) (7.77,7.77) (1,2)
(1,2) (1,2) (1,2)
```

**2.9.13 operator ()**

Indexing operators

```
cvector scmatrix::operator () (int i) throw (cvmexception);
const rvector scmatrix::operator () (int i) const throw (cvmexception);
```

provide access to *i*-th column of calling matrix. The first version of operator is applicable to non-constant object and *returns l-value*, i.e. vector returned shares memory with *i*-th column of calling matrix in order to make possible write access to it. The second version creates *copy* of a column and therefore is *not l-value*. Both operators are *ℓ-based*. Operators throw *cvmexception* if parameter *i* is outside of [1, *nsize()*] range. Operators are *redefined* in the classes *scbmatrix* and *schmatrix*. See also *scmatrix*. Example:

```
using namespace cvm;

try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
                  10., 11., 12., 13., 14., 15., 16., 17., 18.};
    const scmatrix m ((std::complex<double>*)a, 3);
    scmatrix ms(3);
    std::cout << m(2) << std::endl;

    ms(2) = m(3);
    std::cout << ms;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

(7,8) (9,10) (11,12)

(0,0) (13,14) (0,0)
(0,0) (15,16) (0,0)
(0,0) (17,18) (0,0)
```

**2.9.14 operator []**

Indexing operators

```
cvector scmatrix::operator [] (int i) throw (cvmexception);
const cvector scmatrix::operator [] (int i) const throw (cvmexception);
```

provide access to *i*-th row of calling matrix. The first version is applicable to non-constant object and *returns l-value*, i.e. vector returned shares memory with *i*-th row of calling matrix in order to make possible write access to it. The second version creates *copy* of a row and therefore is *not l-value*. Both operators are *l-based*. Operators throw **cvmexception** if parameter *i* is outside of [1,msize()] range. Operators are *redefined* in the classes **scbmatrix** and **schmatrix**. See also **scmatrix**, **Matrix::msize()**. Example:

```
using namespace cvm;

try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
                  10., 11., 12., 13., 14., 15., 16., 17., 18.};
    const scmatrix m ((std::complex<double>*)a, 3);
    scmatrix ms(3);
    std::cout << m[2] << std::endl;

    ms[2] = m[3];
    std::cout << ms;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

(3,4) (9,10) (15,16)

(0,0) (0,0) (0,0)
(5,6) (11,12) (17,18)
(0,0) (0,0) (0,0)
```

### 2.9.15 `real`

Function

```
const srmatrix scmatrix::real () const;
```

creates object of type `const srmatrix` as real part of calling matrix. Please note that, unlike `cvector::real`, this function creates new object *not sharing* memory with real part of calling matrix, i.e. matrix returned is *not l-value*. Function is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `srmatrix`, `scmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
scmatrix m((std::complex<double>*) a, 2);  
std::cout << m << std::endl << m.real();
```

prints

```
(1,2) (5,6)  
(3,4) (7,8)
```

```
1 5  
3 7
```



### 2.9.16 `imag`

Function

```
const srmatrix scmatrix::imag () const;
```

creates object of type `const srmatrix` as imaginary part of calling matrix. Please note that, unlike `cvvector::imag`, this function creates new object *not sharing* memory with imaginary part of calling matrix, i.e. matrix returned is *not l-value*. Function is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `srmatrix`, `scmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
scmatrix m((std::complex<double>*) a, 2);  
std::cout << m << std::endl << m.imag();
```

prints

```
(1,2) (5,6)  
(3,4) (7,8)
```

```
2 6  
4 8
```

**2.9.17 operator = (const *scmatrix*&)**

Operator

```
scmatrix& scmatrix::operator = (const scmatrix& m)
throw (cvmexception);
```

sets every element of calling matrix to value of appropriate element of matrix *m* and returns reference to the matrix changed. Operator throws *cvmexception* in case of different matrix sizes. Operator is *redefined* in the classes *scbmatrix* and *schmatrix*. See also *scmatrix*. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
    const scmatrix m1((std::complex<double>*) a, 2);
    scmatrix m2(2);

    m2 = m1;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(1,2) (5,6)
(3,4) (7,8)
```

**2.9.18 assign (const TC\*)**

Function

```
scmatrix& scmatrix::assign (const cvector& v) throw (cvmexception);  
scmatrix& scmatrix::assign (const TC* pD);
```

sets every element of calling matrix to value of appropriate element of vector *v* or array pointed to by *pD* and returns reference to the matrix changed. In first version it's assumed that vector passed is long enough to fill calling matrix. Function throws *cvmexception* otherwise. Function is *redefined* in the classes *scbmatrix* and *schmatrix*. See also *scmatrix*. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,  
             10., 11., 12., 13., 14., 15., 16., 17., 18.};  
scmatrix m(3);  
m.assign((std::complex<double>*) a);  
std::cout << m;
```

prints

```
(1,2) (7,8) (13,14)  
(3,4) (9,10) (15,16)  
(5,6) (11,12) (17,18)
```

**2.9.19 assign (int, int, const cmatrix&)**

Function

```
scmatrix& scmatrix::assign (int nRow, int nCol, const cmatrix& m)
throw (cvmexception);
```

sets sub-matrix of calling matrix beginning with *l*-based row nRow and column nCol to matrix m and returns reference to the matrix changed. Function throws *cvmexception* if nRow or nCol are not positive or matrix m doesn't fit. Function is *redefined* in the class *schmatrix*. See also *cmatrix*, *scmatrix*. Example:

```
using namespace cvm;
```

```
scmatrix m1(5);
cmatrix m2(2,3);
m1.set(std::complex<double>(1.,1.));
m2.set(std::complex<double>(2.,2.));
m1.assign(2,3,m2);
std::cout << m1;
```

prints

```
(1,1) (1,1) (1,1) (1,1) (1,1)
(1,1) (1,1) (2,2) (2,2) (2,2)
(1,1) (1,1) (2,2) (2,2) (2,2)
(1,1) (1,1) (1,1) (1,1) (1,1)
(1,1) (1,1) (1,1) (1,1) (1,1)
```

**2.9.20 set (TC)**

Function

```
scmatrix& scmatrix::set (TC x);
```

sets every element of calling matrix to value of parameter *x* and returns reference to the matrix changed. Use **vanish** to set every element of calling matrix to be equal to zero. Function is *redefined* in the classes **scbmatrix** and *not applicable* to objects of the class **srsmatrix** (i.e. **cvmexception** would be thrown in case of using it for objects of this class). See also **scmatrix**. Example:

```
using namespace cvm;
```

```
scmatrix m(3);  
m.set(std::complex<double>(1.,2.));  
std::cout << m;
```

prints

```
(1,2) (1,2) (1,2)  
(1,2) (1,2) (1,2)  
(1,2) (1,2) (1,2)
```

### 2.9.21 `assign_real`

Function

```
scmatrix& scmatrix::assign_real (const srmatrix& mRe)  
throw (cvmexception);
```

sets real part of every element of calling matrix to value of appropriate element of matrix `mRe` and returns reference to the matrix changed. Function throws `cvmexception` in case of different sizes of the operands. Function is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix` and `srmatrix`. Example:

```
using namespace cvm;  
scmatrix mc(3);  
srmatrix m(3);  
m.set(1.);  
mc.assign_real(m);  
std::cout << mc;
```

prints

```
(1,0) (1,0) (1,0)  
(1,0) (1,0) (1,0)  
(1,0) (1,0) (1,0)
```

### 2.9.22 `assign_imag`

Function

```
scmatrix& scmatrix::assign_imag (const srmatrix& mIm)  
throw (cvmexception);
```

sets imaginary part of every element of calling matrix to value of appropriate element of matrix `mIm` and returns reference to the matrix changed. Function throws `cvmexception` in case of different sizes of the operands. Function is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix` and `srmatrix`. Example:

```
using namespace cvm;  
scmatrix mc(3);  
srmatrix m(3);  
m.set(1.);  
mc.assign_imag(m);  
std::cout << mc;
```

prints

```
(0,1) (0,1) (0,1)  
(0,1) (0,1) (0,1)  
(0,1) (0,1) (0,1)
```

### 2.9.23 `set_real`

Function

```
scmatrix& scmatrix::set_real (TR d);
```

sets real part of every element of calling matrix to value of parameter `d` and returns reference to the matrix changed. Function is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix`. Example:

```
using namespace cvm;
```

```
scmatrix m(3);  
m.set_real(1.);  
std::cout << m;
```

prints

```
(1,0) (1,0) (1,0)  
(1,0) (1,0) (1,0)  
(1,0) (1,0) (1,0)
```



### 2.9.24 `set_imag`

Function

```
scmatrix& scmatrix::set_imag (TR d);
```

sets imaginary part of every element of calling matrix to value of parameter `d` and returns reference to the matrix changed. Function is *redefined* in the class `scbmatrix` and *not allowed* in `schmatrix`. See also `scmatrix`. Example:

```
using namespace cvm;
```

```
scmatrix m(3);  
m.set_imag(1.);  
std::cout << m;
```

prints

```
(0,1) (0,1) (0,1)  
(0,1) (0,1) (0,1)  
(0,1) (0,1) (0,1)
```

### 2.9.25 **resize**

Function

```
scmatrix& scmatrix::resize (int nNewMN);  
throw (cvmexception);
```

changes size of calling matrix to *nNewMN* by *nNewMN* and returns reference to the matrix changed. In case of increasing of its size, the matrix is filled up with zeroes. Function throws **cvmexception** in case of negative size passed or memory allocation failure. Function is *redefined* in the classes **scbmatrix** and **schmatrix**. See also **scmatrix**. Example:

```
using namespace cvm;
```

```
try {  
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
    scmatrix m((std::complex<double>*) a, 2);  
  
    std::cout << m << std::endl;  
    m.resize (3);  
    std::cout << m;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
(1,2) (5,6)  
(3,4) (7,8)  
  
(1,2) (5,6) (0,0)  
(3,4) (7,8) (0,0)  
(0,0) (0,0) (0,0)
```

**2.9.26 operator <<**

Operator

```
scmatrix& scmatrix::operator << (const scmatrix& m)
throw (cvmexception);
```

destroys calling matrix, creates new one as a copy of *m* and returns reference to the matrix changed. Operator throws **cvmexception** in case of memory allocation failure. Operator is *redefined* in the classes **scbmatrix** and **schmatrix**. See also **scmatrix**. Example:

```
using namespace cvm;
```

```
try {
    scmatrix m(3);
    scmatrix mc(1);
    m(1,2) = 1.;
    m(2,3) = 2.;
    std::cout << m << std::endl << mc << std::endl;

    mc << m;
    std::cout << mc;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(0,0) (1,0) (0,0)
(0,0) (0,0) (2,0)
(0,0) (0,0) (0,0)
```

```
(0,0)
```

```
(0,0) (1,0) (0,0)
(0,0) (0,0) (2,0)
(0,0) (0,0) (0,0)
```

**2.9.27 operator +**

Operator

```
scmatrix scmatrix::operator + (const scmatrix& m) const
throw (cvmexception);
```

creates object of type `scmatrix` as sum of calling matrix and matrix `m`. Operator throws `cvmexception` in case of different sizes of the operands. Operator is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix::sum`, `scmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
    double b[] = {10., 20., 30., 40., 50., 60., 70., 80.};
    scmatrix m1((std::complex<double>*) a, 2);
    scmatrix m2((std::complex<double>*) b, 2);

    std::cout << m1 + m2 << std::endl << m1 + m1;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

(11,22) (55,66)

(33,44) (77,88)

(2,4) (10,12)

(6,8) (14,16)

**2.9.28 operator -**

Operator

```
scmatrix scmatrix::operator - (const scmatrix& m) const
throw (cvmexception);
```

creates object of type `scmatrix` as difference of calling matrix and matrix `m`. It throws `cvmexception` in case of different sizes of the operands. Operator is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix::diff`, `scmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
    double b[] = {10., 20., 30., 40., 50., 60., 70., 80.};
    scmatrix m1((std::complex<double>*) a, 2);
    scmatrix m2((std::complex<double>*) b, 2);

    std::cout << m2 - m1 << std::endl << m1 - m1;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(9,18) (45,54)
(27,36) (63,72)

(0,0) (0,0)
(0,0) (0,0)
```

**2.9.29 sum**

Function

```
scmatrix& scmatrix::sum (const scmatrix& m1, const scmatrix& m2)
throw (cvmexception);
```

assigns result of addition of matrices *m1* and *m2* to calling matrix and returns reference to the matrix changed. It throws **cvmexception** in case of different sizes of the operands. Function is *redefined* in the classes **schmatrix** and **schmatrix**. See also **scmatrix::operator +**, **scmatrix**. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
                  10., 11., 12., 13., 14., 15., 16., 17., 18.};
    const scmatrix m1((std::complex<double>*)a, 3);
    scmatrix m2(3);
    scmatrix m(3);
    m2.set(std::complex<double>(1.,1.));

    std::cout << m.sum(m1, m2) << std::endl;
    std::cout << m.sum(m, m2);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(2,3) (8,9) (14,15)
(4,5) (10,11) (16,17)
(6,7) (12,13) (18,19)

(3,4) (9,10) (15,16)
(5,6) (11,12) (17,18)
(7,8) (13,14) (19,20)
```

**2.9.30 diff**

Function

```
scmatrix& scmatrix::diff (const scmatrix& m1, const scmatrix& m2)
throw (cvmexception);
```

assigns result of subtraction of matrices *m1* and *m2* to calling matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. Function is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix::operator -`, `scmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
                 10., 11., 12., 13., 14., 15., 16., 17., 18.};
    const scmatrix m1((std::complex<double>*)a, 3);
    scmatrix m2(3);
    scmatrix m(3);
    m2.set(std::complex<double>(1.,1.));

    std::cout << m.diff(m1, m2) << std::endl;
    std::cout << m.diff(m, m2);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(0,1) (6,7) (12,13)
(2,3) (8,9) (14,15)
(4,5) (10,11) (16,17)

(-1,0) (5,6) (11,12)
(1,2) (7,8) (13,14)
(3,4) (9,10) (15,16)
```

### 2.9.31 operator +=

Operator

```
scmatrix& scmatrix::operator += (const scmatrix& m) throw (cvmexception);
```

adds matrix *m* to calling matrix and returns reference to the matrix changed. It throws *cvmexception* in case of different sizes of the operands. Operator is *redefined* in the classes *scbmatrix* and *scmatrix*. See also *scmatrix::operator +* , *scmatrix::sum*, *scmatrix*. Example:

```
using namespace cvm;
```

```
try {
    scmatrix m1(3);
    scmatrix m2(3);
    m1.set(std::complex<double>(1.,2.));
    m2.set(std::complex<double>(3.,4.));

    m1 += m2;
    std::cout << m1 << std::endl;

    // well, you can do this too, but temporary object would be created
    m2 += m2;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(4,6) (4,6) (4,6)
(4,6) (4,6) (4,6)
(4,6) (4,6) (4,6)

(6,8) (6,8) (6,8)
(6,8) (6,8) (6,8)
(6,8) (6,8) (6,8)
```



**2.9.32 operator -=**

Operator

```
scmatrix& scmatrix::operator -= (const scmatrix& m) throw (cvmexception);
```

subtracts matrix *m* from calling matrix and returns reference to the matrix changed. It throws **cvmexception** in case of different sizes of the operands. Operator is *redefined* in the classes **scbmatrix** and **schmatrix**. See also **scmatrix::operator -** , **scmatrix::diff**, **scmatrix**. Example:

```
using namespace cvm;
```

```
try {
    scmatrix m1(3);
    scmatrix m2(3);
    m1.set(std::complex<double>(1.,2.));
    m2.set(std::complex<double>(3.,4.));

    m1 -= m2;
    std::cout << m1 << std::endl;

    // well, you can do this too, but temporary object would be created
    m2 -= m2;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(-2,-2) (-2,-2) (-2,-2)
(-2,-2) (-2,-2) (-2,-2)
(-2,-2) (-2,-2) (-2,-2)
```

```
(0,0) (0,0) (0,0)
(0,0) (0,0) (0,0)
(0,0) (0,0) (0,0)
```

### 2.9.33 operator - ()

Operator

```
scmatrix scmatrix::operator - () const throw (cvmexception);
```

creates object of type *scmatrix* as calling matrix multiplied by  $-1$ . Operator is *redefined* in the classes *scbmatrix* and *schmatrix*. See also *scmatrix*. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
scmatrix m((std::complex<double>*) a, 2);  
std::cout << -m;
```

prints

```
(-1,-2) (-5,-6)  
(-3,-4) (-7,-8)
```

### 2.9.34 operator ++

Operator

```
scmatrix& scmatrix::operator ++ ();  
scmatrix& scmatrix::operator ++ (int);
```

adds identity matrix to calling matrix and returns reference to the matrix changed. Operator is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix`. Example:

```
using namespace cvm;
```

```
scmatrix m(3);  
m.set(std::complex<double>(1.,1.));  
m++;  
std::cout << m << std::endl;  
std::cout << ++m;
```

prints

```
(2,1) (1,1) (1,1)  
(1,1) (2,1) (1,1)  
(1,1) (1,1) (2,1)
```

```
(3,1) (1,1) (1,1)  
(1,1) (3,1) (1,1)  
(1,1) (1,1) (3,1)
```

### 2.9.35 operator --

Operator

```
scmatrix& scmatrix::operator -- ();  
scmatrix& scmatrix::operator -- (int);
```

subtracts identity matrix from calling matrix and returns reference to the matrix changed. Operator is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix`. Example:

```
using namespace cvm;
```

```
scmatrix m(3);  
m.set(std::complex<double>(1.,1.));  
m--;  
std::cout << m << std::endl;  
std::cout << --m;
```

prints

```
(0,1) (1,1) (1,1)  
(1,1) (0,1) (1,1)  
(1,1) (1,1) (0,1)
```

```
(-1,1) (1,1) (1,1)  
(1,1) (-1,1) (1,1)  
(1,1) (1,1) (-1,1)
```

### 2.9.36 operator \* (TR)

Operator

```
scmatrix scmatrix::operator * (TR d) const;
```

creates object of type *scmatrix* as product of calling matrix and real number *d*. Operator is *redefined* in the classes *scbmatrix* and *schmatrix*. See also *scmatrix::operator \*=* , *scmatrix*. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,  
              10., 11., 12., 13., 14., 15., 16., 17., 18.};  
scmatrix m((std::complex<double>*) a, 3);  
std::cout << m * 5.;
```

prints

```
(5,10) (35,40) (65,70)  
(15,20) (45,50) (75,80)  
(25,30) (55,60) (85,90)
```

### 2.9.37 operator / (TR)

Operator

```
scmatrix scmatrix::operator / (TR d) const throw (cvmexception);
```

creates object of type *scmatrix* as quotient of calling matrix and real number *d*. It throws *cvmexception* if *d* has absolute value equal or less than the *smallest normalized positive number*. Operator is *redefined* in the classes *scbmatrix* and *schmatrix*. See also *scmatrix::operator /=* , *scmatrix*. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,  
             10., 11., 12., 13., 14., 15., 16., 17., 18.};  
scmatrix m((std::complex<double>*) a, 3);  
std::cout << m / 2.;
```

prints

```
(0.5,1) (3.5,4) (6.5,7)  
(1.5,2) (4.5,5) (7.5,8)  
(2.5,3) (5.5,6) (8.5,9)
```

### 2.9.38 operator \* (TC)

Operator

```
scmatrix scmatrix::operator * (TC z) const;
```

creates object of type `scmatrix` as product of calling matrix and complex number `z`. Operator is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix::operator *=`, `scmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,  
              10., 11., 12., 13., 14., 15., 16., 17., 18.};  
scmatrix m((std::complex<double>*) a, 3);  
std::cout << m * std::complex<double>(1.,1.);
```

prints

```
(-1,3) (-1,15) (-1,27)  
(-1,7) (-1,19) (-1,31)  
(-1,11) (-1,23) (-1,35)
```

### 2.9.39 operator / (TC)

Operator

```
scmatrix scmatrix::operator / (TC z) const throw (cvmexception);
```

creates object of type `scmatrix` as quotient of calling matrix and complex number `z`. It throws `cvmexception` if `z` has absolute value equal or less than the **smallest normalized positive number**. Operator is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix::operator /=`, `scmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
              10., 11., 12., 13., 14., 15., 16., 17., 18.};
scmatrix m((std::complex<double>*) a, 3);
std::cout << m / std::complex<double>(1.,1.);
```

prints

```
(1.5,0.5) (7.5,0.5) (13.5,0.5)
(3.5,0.5) (9.5,0.5) (15.5,0.5)
(5.5,0.5) (11.5,0.5) (17.5,0.5)
```



**2.9.40 operator \*= (TR)**

Operator

```
scmatrix& scmatrix::operator *= (TR d);
```

multiplies calling matrix by real number *d* and returns reference to the matrix changed. It's *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix::operator *`, `scmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,  
             10., 11., 12., 13., 14., 15., 16., 17., 18.};  
scmatrix m((std::complex<double>*) a, 3);  
m *= 5.;  
std::cout << m;
```

prints

```
(5,10) (35,40) (65,70)  
(15,20) (45,50) (75,80)  
(25,30) (55,60) (85,90)
```

### 2.9.41 operator /= (TR)

Operator

```
scmatrix& scmatrix::operator /= (TR d) throw (cvmexception);
```

divides calling matrix by real number *d* and returns reference to the matrix changed. It throws **cvmexception** if *d* has absolute value equal or less than the **smallest normalized positive number**. Operator is *redefined* in the classes **scbmatrix** and **schmatrix**. See also **scmatrix::operator /** , **scmatrix**. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,  
             10., 11., 12., 13., 14., 15., 16., 17., 18.};
```

```
scmatrix m((std::complex<double>*) a, 3);
```

```
m /= 2.;
```

```
std::cout << m;
```

prints

```
(0.5,1) (3.5,4) (6.5,7)
```

```
(1.5,2) (4.5,5) (7.5,8)
```

```
(2.5,3) (5.5,6) (8.5,9)
```

**2.9.42 operator \*= (TC)**

Operator

```
scmatrix& scmatrix::operator *= (TC z);
```

multiplies calling matrix by complex number *z* and returns reference to the matrix changed. It's *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix`, `scmatrix::operator *`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
              10., 11., 12., 13., 14., 15., 16., 17., 18.};
scmatrix m((std::complex<double>*) a, 3);
m *= std::complex<double>(2.,1.);
std::cout << m;
```

prints

```
(0,5) (6,23) (12,41)
(2,11) (8,29) (14,47)
(4,17) (10,35) (16,53)
```

**2.9.43 operator /= (TC)**

Operator

```
scmatrix& scmatrix::operator /= (TC z) throw (cvmexception);
```

divides calling matrix by complex number *z* and returns reference to the matrix changed. It throws **cvmexception** if *z* has absolute value equal or less than the **smallest normalized positive number**. Operator is *redefined* in the classes **scbmatrix** and **schmatrix**. See also **scmatrix::operator /** , **scmatrix**. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
              10., 11., 12., 13., 14., 15., 16., 17., 18.};
scmatrix m((std::complex<double>*) a, 3);
m /= std::complex<double>(2.,1.);
std::cout << m;
```

prints

```
(0.8,0.6) (4.4,1.8) (8,3)
(2,1) (5.6,2.2) (9.2,3.4)
(3.2,1.4) (6.8,2.6) (10.4,3.8)
```

### 2.9.44 **normalize**

Function

```
scmatrix& scmatrix::normalize ();
```

normalizes calling matrix so its **Euclidean norm** becomes equal to 1 if it was greater than the **smallest normalized positive number** before the call (otherwise function does nothing). Function is *redefined* in the classes **scbmatrix** and **schmatrix**. See also **scmatrix**. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (5);
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
scmatrix m((std::complex<double>*) a, 2);  
m.normalize();  
std::cout << m << m.norm() << std::endl;
```

prints

```
(7.00140e-002,1.40028e-001) (3.50070e-001,4.20084e-001)  
(2.10042e-001,2.80056e-001) (4.90098e-001,5.60112e-001)  
1.00000e+000
```

**2.9.45 conjugation**

Operator and functions

```
scmatrix scmatrix::operator ~ () const throw (cvmexception);
scmatrix& scmatrix::conj (const scmatrix& m) throw (cvmexception);
scmatrix& scmatrix::conj () throw (cvmexception);
```

implement complex matrix conjugation. First operator creates object of type `scmatrix` as conjugated calling matrix (it throws `cvmexception` in case of memory allocation failure). Second function sets calling matrix to be equal to matrix `m` conjugated (it throws `cvmexception` in case of not appropriate sizes of the operands), third one makes it to be equal to conjugated itself (it also throws `cvmexception` in case of memory allocation failure). Functions are *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix`. Example:

```
using namespace cvm;

double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
              10., 11., 12., 13., 14., 15., 16., 17., 18.};
scmatrix m((std::complex<double>*) a, 3);
scmatrix mc(3);
std::cout << m << std::endl << ~m << std::endl ;
mc.conj(m);
std::cout << mc << std::endl;
mc.conj();
std::cout << mc;
```

prints

```
(1,2) (7,8) (13,14)
(3,4) (9,10) (15,16)
(5,6) (11,12) (17,18)

(1,-2) (3,-4) (5,-6)
(7,-8) (9,-10) (11,-12)
(13,-14) (15,-16) (17,-18)

(1,-2) (3,-4) (5,-6)
(7,-8) (9,-10) (11,-12)
(13,-14) (15,-16) (17,-18)

(1,2) (7,8) (13,14)
(3,4) (9,10) (15,16)
(5,6) (11,12) (17,18)
```

**2.9.46 transposition**

Operator and functions

```
scmatrix scmatrix::operator ! () const throw (cvmexception);
scmatrix& scmatrix::transpose (const scmatrix& m) throw (cvmexception);
scmatrix& scmatrix::transpose () throw (cvmexception);
```

implement complex matrix transposition (*not* conjugation). First operator creates object of type `scmatrix` as transposed calling matrix (it throws `cvmexception` in case of memory allocation failure). Second function sets calling matrix to be equal to matrix `m` transposed (it throws `cvmexception` in case of not appropriate sizes of the operands), third one makes it to be equal to transposed itself (it also throws `cvmexception` in case of memory allocation failure). Functions are *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix`. Example:

```
using namespace cvm;

double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
              10., 11., 12., 13., 14., 15., 16., 17., 18.};
scmatrix m((std::complex<double>*) a, 3);
scmatrix mc(3);
std::cout << m << std::endl << !m << std::endl ;
mc.transpose(m);
std::cout << mc << std::endl;
mc.transpose();
std::cout << mc;
```

prints

```
(1,2) (7,8) (13,14)
(3,4) (9,10) (15,16)
(5,6) (11,12) (17,18)

(1,2) (3,4) (5,6)
(7,8) (9,10) (11,12)
(13,14) (15,16) (17,18)

(1,2) (3,4) (5,6)
(7,8) (9,10) (11,12)
(13,14) (15,16) (17,18)

(1,2) (7,8) (13,14)
(3,4) (9,10) (15,16)
(5,6) (11,12) (17,18)
```

**2.9.47 operator \* (const cvector&)**

Operator

```
cvector scmatrix::operator * (const cvector& v) const  
throw (cvmexception);
```

creates object of type `cvector` as product of calling matrix and vector `v`. It throws `cvmexception` if number of columns of calling matrix differs from size of vector `v`. Use `cvector::mult` in order to avoid new object creation. Function is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix`, `cvector`. Example:

```
using namespace cvm;
```

```
try {  
    scmatrix m(3);  
    cvector v(3);  
    m.set(std::complex<double>(1.,1.));  
    v.set(std::complex<double>(1.,1.));  
    std::cout << m * v;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
(0,6) (0,6) (0,6)
```



**2.9.48 operator \* (const cmatrix&)**

Operator

```
cmatrix scmatrix::operator * (const cmatrix& m) const  
throw (cvmexception);
```

creates object of type `cmatrix` as product of calling matrix and matrix `m`. It throws `cvmexception` if number of columns of calling matrix differs from number of rows of matrix `m`. Use `cmatrix::mult` in order to avoid new object creation. Operator is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `cmatrix`, `scmatrix`. Example:

```
using namespace cvm;
```

```
try {  
    scmatrix ms(3);  
    cmatrix m(3,2);  
    ms.set(std::complex<double>(1.,1.));  
    m.set(std::complex<double>(1.,1.));  
    std::cout << ms * m;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
(0,6) (0,6)  
(0,6) (0,6)  
(0,6) (0,6)
```

### 2.9.49 operator \* (const *scmatrix*&)

Operator

```
scmatrix scmatrix::operator * (const scmatrix& m) const  
throw (cvmexception);
```

creates object of type *scmatrix* as product of calling matrix and matrix *m*. It throws *cvmexception* if the operands have different sizes. Use *cmatrix::mult* in order to avoid new object creation. Operator is *inherited* in the class *scbmatrix* and *redefined* in *schmatrix*. See also *scmatrix*. Example:

```
using namespace cvm;
```

```
try {  
    scmatrix m1(3), m2(3);  
    m1.set(std::complex<double>(1.,1.));  
    m2.set(std::complex<double>(1.,1.));  
    std::cout << m1 * m2;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
(0,6) (0,6) (0,6)  
(0,6) (0,6) (0,6)  
(0,6) (0,6) (0,6)
```

**2.9.50 operator \*= (const scmatrix&)**

Operator

```
scmatrix& scmatrix::operator *= (const scmatrix& m)
throw (cvmexception);
```

sets calling matrix to be equal to product of itself by matrix *m* and returns reference to the object it changes. Operator throws **cvmexception** in case of different sizes of the operands. Operator is *inherited* in the class **scbmatrix** and *redefined* in **schmatrix**. See also **scmatrix**. Example:

```
using namespace cvm;
```

```
try {
    scmatrix m1(3), m2(3);
    m1.set(std::complex<double>(1.,2.));
    m2.set(std::complex<double>(2.,1.));
    m1 *= m2;
    m2 *= m2;
    std::cout << m1 << std::endl << m2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(0,15) (0,15) (0,15)
(0,15) (0,15) (0,15)
(0,15) (0,15) (0,15)

(9,12) (9,12) (9,12)
(9,12) (9,12) (9,12)
(9,12) (9,12) (9,12)
```

**2.9.51 swap\_rows**

Function

```
scmatrix& scmatrix::swap_rows (int n1, int n2) throw (cvmexception);
```

swaps two rows of calling matrix and returns reference to the matrix changed. *n1* and *n2* are numbers of rows to be swapped, both are *l*-based). Function throws **cvmexception** if one of parameters is outside of the range [1,msize()]. Function is *not applicable* to objects of the classes **scbmatrix** and **schmatrix** (i.e. **cvmexception** would be thrown in case of using it for objects of those classes). See also **scmatrix**. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
                 10., 11., 12., 13., 14., 15., 16., 17., 18.};
    scmatrix m ((std::complex<double>*)a, 3);

    std::cout << m << std::endl;
    std::cout << m.swap_rows(2,3);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(1,2) (7,8) (13,14)
(3,4) (9,10) (15,16)
(5,6) (11,12) (17,18)

(1,2) (7,8) (13,14)
(5,6) (11,12) (17,18)
(3,4) (9,10) (15,16)
```

**2.9.52 swap\_cols**

Function

```
scmatrix& scmatrix::swap_cols (int n1, int n2) throw (cvmexception);
```

swaps two columns of calling matrix and returns reference to the matrix changed. *n1* and *n2* are numbers of columns to be swapped, both are *ℓ-based*). Function throws **cvmexception** if one of parameters is outside of the range [1, *nsize()*]. Function is *not applicable* to objects of the classes **scbmatrix** and **schmatrix** (i.e. **cvmexception** would be thrown in case of using it for objects of those classes). See also **scmatrix**. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
                 10., 11., 12., 13., 14., 15., 16., 17., 18.};
    scmatrix m ((std::complex<double>*)a, 3);

    std::cout << m << std::endl;
    std::cout << m.swap_cols(2,3);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(1,2) (7,8) (13,14)
(3,4) (9,10) (15,16)
(5,6) (11,12) (17,18)

(1,2) (13,14) (7,8)
(3,4) (15,16) (9,10)
(5,6) (17,18) (11,12)
```

**2.9.53 solve**

Functions

```

cvector scmatrix::solve (const cvector& vB) const throw (cvmexception);
cmatrix scmatrix::solve (const cmatrix& mB) const throw (cvmexception);
cvector scmatrix::solve (const cvector& vB, TR& dErr) const
throw (cvmexception);
cmatrix scmatrix::solve (const cmatrix& mB, TR& dErr) const
throw (cvmexception);

```

return solution of linear equation of kind  $A * x = b$  or  $A * X = B$  where  $A$  is calling matrix. The first and third versions solve the equation  $A * x = b$  where vector  $b$  is passed in parameter  $vB$  and the second and fourth versions solve the equation  $A * X = B$  where matrix  $B$  is passed in parameter  $mB$ . The last two versions also set output parameter  $dErr$  to be equal to the norm of computation error. Functions throw `cvmexception` in case of inappropriate sizes of the operands. Functions are *inherited* in the classes `scbmatrix` and `schmatrix`. See also `cvector::solve`, `cmatrix::solve`, `scmatrix`. Example:

```

using namespace cvm;

std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (2);
try {
    double re[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
    double im[] = {-1., 2., -3., -4., 5., -6., 7., -8., 9.};
    scmatrix ma(re, im, 3);
    cmatrix  mb(3,2);
    cmatrix  mx(3,2);
    double dErr;

    mb(1).set(std::complex<double>(1.,1.));
    mb(1,2) = std::complex<double>(1.,1.);
    mb(2,2) = std::complex<double>(2.,2.);
    mb(3,2) = std::complex<double>(3.,3.);

    mx.solve (ma, mb, dErr);
    std::cout << mx << dErr
              << std::endl << ma * mx - mb << std::endl;

    cvector vb(3), vx(3);
    vb = mb(2);
    vx = ma.solve (vb, dErr);
    std::cout << vx << dErr << std::endl << ma * vx - vb;
}

```

```
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}  
  
prints  
  
(-6.25e-002,-2.71e-001) (-1.25e-001,+1.13e+000)  
(+1.25e-001,+2.08e-001) (+2.50e-001,-2.50e-001)  
(+6.25e-002,+6.25e-002) (+1.25e-001,+1.25e-001)  
+1.45e-014  
(+0.00e+000,+0.00e+000) (+0.00e+000,+0.00e+000)  
(-2.22e-016,+4.44e-016) (+0.00e+000,+0.00e+000)  
(+2.22e-016,+0.00e+000) (+0.00e+000,+0.00e+000)  
  
(-1.25e-001,+1.13e+000) (+2.50e-001,-2.50e-001) (+1.25e-001,+1.25e-001)  
+7.41e-015  
(+0.00e+000,+0.00e+000) (+0.00e+000,+0.00e+000) (+0.00e+000,+0.00e+000)
```

**2.9.54 solve\_tran**

Functions

```

cvector scmatrix::solve_tran (const cvector& vB)
const throw (cvmexception);
cmatrix scmatrix::solve_tran (const cmatrix& mB)
const throw (cvmexception);
cvector scmatrix::solve_tran (const cvector& vB, TR& dErr) const
throw (cvmexception);
cmatrix scmatrix::solve_tran (const cmatrix& mB, TR& dErr) const
throw (cvmexception);

```

return solution of linear equation of kind  $A^T * x = b$  or  $A^T * X = B$  (equivalent to  $x * A = b$  and  $X * A = B$ ) where  $A$  is calling matrix. The first and third versions solve equation  $A^T * x = b$  where vector  $b$  is passed in parameter  $vB$  and the second and fourth versions solve equation  $A^T * X = B$  where matrix  $B$  is passed in parameter  $mB$ . The last two versions also set output parameter  $dErr$  to be equal to the norm of computation error. Functions throw `cvmexception` in case of inappropriate sizes of the operands. Function is *inherited* in the classes `scbmatrix` and `schmatrix`. See also `cvector::solve`, `cmatrix::solve`, `scmatrix::solve`, `scmatrix`. Example:

```

using namespace cvm;

std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (2);
try {
    double re[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
    double im[] = {-1., 2., -3., -4., 5., -6., 7., -8., 9.};
    scmatrix ma(re, im, 3);
    cmatrix  mb(3,2);
    cmatrix  mx(3,2);
    double dErr;

    mb(1).set(std::complex<double>(1.,1.));
    mb(1,2) = std::complex<double>(1.,1.);
    mb(2,2) = std::complex<double>(2.,2.);
    mb(3,2) = std::complex<double>(3.,3.);

    mx.solve_tran (ma, mb, dErr);
    std::cout << mx << dErr
              << std::endl << !ma * mx - mb
              << std::endl << !mx * ma - !mb << std::endl;
}

```



```

    cvector vb(3), vx(3);
    vb = mb(2);
    vx = ma.solve_tran (vb, dErr);
    std::cout << vx << dErr << std::endl << vx * ma - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
}

prints

(+3.13e-002,-5.31e-001) (+9.38e-002,-9.38e-002)
(+6.25e-002,+6.25e-002) (+1.88e-001,+1.88e-001)
(+3.13e-002,+4.69e-001) (+9.38e-002,+2.40e-001)
+1.02e-014
(-2.22e-016,-2.22e-016) (+0.00e+000,+0.00e+000)
(-4.44e-016,-4.44e-016) (+0.00e+000,+0.00e+000)
(+8.88e-016,-8.88e-016) (+0.00e+000,+0.00e+000)

(-2.22e-016,-2.22e-016) (-4.44e-016,-4.44e-016) (+8.88e-016,-8.88e-016)
(+0.00e+000,+0.00e+000) (+0.00e+000,+0.00e+000) (+0.00e+000,+0.00e+000)

(+9.38e-002,-9.38e-002) (+1.88e-001,+1.88e-001) (+9.38e-002,+2.40e-001)
+1.02e-014
(+0.00e+000,+0.00e+000) (+0.00e+000,+0.00e+000) (+0.00e+000,+0.00e+000)

```

**2.9.55 solve\_conj**

Functions

```

cvector scmatrix::solve_conj (const cvector& vB)
const throw (cvmexception);
cmatrix scmatrix::solve_conj (const cmatrix& mB)
const throw (cvmexception);
cvector scmatrix::solve_conj (const cvector& vB, TR& dErr) const
throw (cvmexception);
cmatrix scmatrix::solve_conj (const cmatrix& mB, TR& dErr) const
throw (cvmexception);

```

return solution of linear equation of kind  $A^H * x = b$  or  $A^H * X = B$  (which is equivalent to  $X^H * A = B^H$ ), where  $A^H$  is conjugated  $A$ . Here  $A$  is calling matrix. The first and third versions solve equation  $A^H * x = b$  where vector  $b$  is passed in parameter  $vB$  and the second and fourth versions solve equation  $A^H * X = B$  where matrix  $B$  is passed in parameter  $mB$ . The last two versions also set output parameter  $dErr$  to be equal to the norm of computation error. Functions throw `cvmexception` in case of inappropriate sizes of the operands. Function is *inherited* in the classes `scbmatrix` and `schmatrix`. See also `cvector::solve`, `cmatrix::solve`, `scmatrix::solve`, `scmatrix`. Example:

```

using namespace cvm;

std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (2);
try {
    double re[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
    double im[] = {-1., 2., -3., -4., 5., -6., 7., -8., 9.};
    scmatrix ma(re, im, 3);
    cmatrix  mb(3,2);
    cmatrix  mx(3,2);
    double dErr;

    mb(1).set(std::complex<double>(1.,1.));
    mb(1,2) = std::complex<double>(1.,1.);
    mb(2,2) = std::complex<double>(2.,2.);
    mb(3,2) = std::complex<double>(3.,3.);

    mx.solve_conj (ma, mb, dErr);
    std::cout << mx << dErr
              << std::endl << ~ma * mx - mb
              << std::endl << ~mx * ma - ~mb << std::endl;
}

```

```

    cvector vb(3), vx(3);
    vb = mb(2);
    vx = ma.solve_conj (vb, dErr);
    std::cout << vx << dErr << std::endl << ~ma * vx - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

```

prints

```

(-5.31e-001,+3.13e-002) (-9.38e-002,+9.38e-002)
(+6.25e-002,+6.25e-002) (+1.88e-001,+1.88e-001)
(+4.69e-001,+3.13e-002) (+2.40e-001,+9.38e-002)
+1.02e-014
(-2.22e-016,-2.22e-016) (+0.00e+000,+0.00e+000)
(-4.44e-016,-4.44e-016) (+0.00e+000,+0.00e+000)
(-8.88e-016,+8.88e-016) (+0.00e+000,+0.00e+000)

(-2.22e-016,+2.22e-016) (-4.44e-016,+4.44e-016) (-8.88e-016,-8.88e-016)
(+0.00e+000,+0.00e+000) (+0.00e+000,+0.00e+000) (+0.00e+000,+0.00e+000)

(-9.38e-002,+9.38e-002) (+1.88e-001,+1.88e-001) (+2.40e-001,+9.38e-002)
+1.02e-014
(+0.00e+000,+0.00e+000) (+0.00e+000,+0.00e+000) (+0.00e+000,+0.00e+000)

```

**2.9.56 operator % (const cvector&)**

Operator

```
cvector operator % (const cvector& vB) const throw (cvmexception);
```

returns solution  $x$  of linear equation  $A^T * x = b$  (which is equivalent to  $x * A = b$ ) where calling matrix is square matrix  $A$  and vector  $b$  is passed in parameter  $vB$ . This operator throws exception of type `cvmexception` in case of inappropriate sizes of the objects or when matrix  $A$  is close to cingular. See also `cvector::solve_tran`, `scmatrix::solve_tran`, `cvector.operator /`, `cvector`, `scmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (7);

try {
    double m[] = {1., -1., 1., 2., -2., 1., 3., -3.};
    double b[] = {1., 2., 5., -3.};
    scmatrix ma((std::complex<double>*) m, 2);
    cvector vb((std::complex<double>*) b, 2);
    cvector vx(2);

    vx = ma % vb;

    std::cout << vx * ma - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

(+0.00000000e+000,+0.00000000e+000) (+0.00000000e+000,+4.4408921e-016)
```

**2.9.57 operator / (const cvector&)**

Operator

```
cvector operator / (const cvector& vB) const throw (cvmexception);
```

returns solution  $x$  of linear equation  $A * x = b$  where calling matrix is square matrix  $A$  and vector  $b$  is passed in parameter  $vB$ . This operator throws exception of type `cvmexception` in case of inappropriate sizes of the objects or when matrix  $A$  is close to cingular. It's *redefined* in class `scmatrix`. See also `cvector::solve`, `scmatrix::solve`, `cvector.operator %`, `cvector`, `scmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (7);

try {
    double m[] = {1., -1., 1., 2., -2., 1., 3., -3.};
    double b[] = {1., 2., 5., -3.};
    scmatrix ma((std::complex<double>*) m, 2);
    cvector vb((std::complex<double>*) b, 2);
    cvector vx(2);

    vx = ma / vb;

    std::cout << ma * vx - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
(-6.6613381e-016,+4.4408921e-016) (+0.0000000e+000,+0.0000000e+000)
```

**2.9.58 solve\_lu**

Functions

```

cvector
scmatrix::solve_lu (const scmatrix& mLU, const int* pPivots,
                   const cvector& vB, TR& dErr) throw (cvmexception);

cvector
scmatrix::solve_lu (const scmatrix& mLU, const int* pPivots,
                   const cvector& vB) throw (cvmexception);

cmatrix
scmatrix::solve_lu (const scmatrix& mLU, const int* pPivots,
                   const cmatrix& mB, TR& dErr) throw (cvmexception);

cmatrix
scmatrix::solve_lu (const scmatrix& mLU, const int* pPivots,
                   const cmatrix& mB) throw (cvmexception);

```

create object of type `cvector` or `cmatrix` as solution  $x$  or  $X$  of matrix linear equation  $A * x = b$  or  $A * X = B$  respectively. Here  $A$  is calling matrix, parameter `mLU` is [LU factorization](#) of the matrix  $A$ , parameter `pPivots` is an array of pivot numbers created while factorizing the matrix  $A$  and parameters `vB` and `mB` are the vector  $b$  and matrix  $B$  respectively. The first and third version also set output parameter `dErr` to be equal to the norm of computation error. These functions are useful when you need to solve few linear equations of kind  $A * x = b$  or  $A * X = B$  with the same matrix  $A$  and different vectors  $b$  or matrices  $B$ . In such case you save on matrix  $A$  factorization since it's needed to be performed just one time. Functions throw `cvmexception` in case of inappropriate sizes of the operands or when the matrix  $A$  is close to singular. Function is *inherited* in the classes [scbmatrix](#) and [schmatrix](#). See also [cvector::solve](#), [cmatrix](#), [scmatrix](#). Example:

```

using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    scmatrix ma(3);
    scmatrix mLU(3);
    cmatrix mb1(3,2); cvector vb1(3);
    cmatrix mb2(3,2); cvector vb2(3);
    cmatrix mx1(3,2); cvector vx1(3);
    cmatrix mx2(3,2); cvector vx2(3);
    iarray nPivots(3);
    double dErr = 0.;
    ma.randomize_real(-1.1,3.); ma.randomize_imag(-3.7,3.);
    mb1.randomize_real(-1.,3.); mb1.randomize_imag(-1.,3.);
    vb1.randomize_real(-2.,3.); vb1.randomize_imag(-3.,1.);

```

```

mb2.randomize_real(-5.,1.); mb2.randomize_imag(-4.,1.);
vb2.randomize_real(-1.,6.); vb1.randomize_imag(-4.,4.);
mLU.low_up(ma, nPivots);
mx1 = ma.solve_lu (mLU, nPivots, mb1, dErr);
std::cout << mx1 << dErr << std::endl;
mx2 = ma.solve_lu (mLU, nPivots, mb2);
std::cout << mx2 << std::endl;;
std::cout << ma * mx1 - mb1 << std::endl << ma * mx2 - mb2;
vx1 = ma.solve_lu (mLU, nPivots, vb1, dErr);
std::cout << vx1 << dErr << std::endl;
vx2 = ma.solve_lu (mLU, nPivots, vb2);
std::cout << vx2 << std::endl;;
std::cout << ma * vx1 - vb1 << std::endl << ma * vx2 - vb2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
}

prints

(-3.91e-001,-1.62e-001) (-3.17e-001,2.39e-001)
(-3.00e-001,-7.91e-001) (4.71e-001,-9.13e-001)
(-2.34e-001,1.09e+000) (1.10e-001,2.72e-001)
2.78e-015
(-2.60e-001,-5.48e-001) (-3.09e-002,-9.62e-001)
(8.77e-001,8.41e-001) (-6.02e-001,1.87e+000)
(4.20e-003,-9.72e-001) (6.18e-001,-5.64e-001)

(0.00e+000,-2.64e-016) (-1.11e-016,1.11e-016)
(0.00e+000,-2.22e-016) (2.22e-016,-4.44e-016)
(0.00e+000,0.00e+000) (-5.55e-017,0.00e+000)

(6.66e-016,-2.22e-016) (6.18e-016,0.00e+000)
(0.00e+000,1.11e-016) (0.00e+000,0.00e+000)
(0.00e+000,0.00e+000) (-4.44e-016,0.00e+000)
(2.61e-001,2.97e-001) (1.95e+000,-1.07e-001) (-5.51e-001,-1.03e-001)
1.96e-015
(1.26e-001,4.07e-001) (-4.82e-001,-1.14e-002) (2.59e-001,1.60e-001)

(1.11e-016,0.00e+000) (5.55e-017,0.00e+000) (1.11e-016,-2.22e-016)

(-1.11e-016,-7.61e-017) (-2.22e-016,-8.94e-017) (0.00e+000,4.07e-017)

```

### 2.9.59 det

Function

```
TC scmatrix::det () const throw (cvmexception);
```

returns determinant of calling matrix. It uses the [LU factorization](#) inside and may throw the same exceptions as the factorizer. Function is *inherited* in the classes [scbmatrix](#) and [schmatrix](#). See also [scmatrix](#). Example:

```
using namespace cvm;
```

```
try {  
    double re[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};  
    double im[] = {-1., 2., -3., -4., 5., -6., 7., -8., 9.};  
    const scmatrix m(re, im, 3);  
  
    std::cout << m << std::endl << m.det() << std::endl;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
(1,-1) (4,-4) (7,7)
```

```
(2,2) (5,5) (8,-8)
```

```
(3,-3) (6,-6) (9,9)
```

```
(-192,-192)
```



**2.9.60 low\_up**

Functions

```
scmatrix& scmatrix::low_up (const scmatrix& m, int* nPivots)
throw (cvmexception);
```

```
scmatrix scmatrix::low_up (int* nPivots) const
throw (cvmexception);
```

compute LU factorization of calling matrix as

$$A = PLU$$

where P is permutation matrix, L is lower triangular matrix with unit diagonal elements and U is upper triangular matrix. All functions store result as the matrix L without main diagonal combined with U. All functions return pivot indices as an array of integers (it should support at least `msize()` elements) pointed to by `nPivots` so *i*-th row was interchanged with `nPivots[i]`-th row. The first version sets calling matrix to be equal to the *m*'s LU factorization and the second one creates object of type `scmatrix` as calling matrix LU factorization. Functions throw `cvmexception` in case of inappropriate sizes of the operands or when the matrix to be factorized is close to singular. It is recommended to use `iarray` for pivot values. Function is *redefined* in the class `scbmatrix` and *inherited* in `schmatrix`. See also `scmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double re[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
    double im[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.1};
    scmatrix m(re, im, 3);
    scmatrix mLU(3), mLo(3), mUp(3);
    iarray naPivots(3);

    mLU.low_up (m, naPivots);

    mLo.identity ();
    mLo.diag(-2) = mLU.diag(-2);
    mLo.diag(-1) = mLU.diag(-1);
    mUp.diag(0) = mLU.diag(0);
    mUp.diag(1) = mLU.diag(1);
    mUp.diag(2) = mLU.diag(2);

    std::cout << mLo << std::endl << mUp
                << std::endl << naPivots << std::endl;
```

```

    mLU = mLo * mUp;
    for (int i = 3; i >= 1; i--) {
        mLU.swap_rows (i, naPivots[i]);
    }
    std::cout << mLU << std::endl << m - mLU;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

```

prints

```

(1,0) (0,0) (0,0)
(0.333333,0) (1,0) (0,0)
(0.666667,0) (0.5,0) (1,0)

(3,3) (6,6) (9,9.1)
(0,0) (2,2) (4,3.96667)
(0,0) (0,0) (-1.11022e-016,-0.05)

```

3 3 3

```

(1,1) (4,4) (7,7)
(2,2) (5,5) (8,8)
(3,3) (6,6) (9,9.1)

(0,0) (0,0) (0,0)
(0,0) (0,0) (0,0)
(0,0) (0,0) (0,0)

```

**2.9.61 cond**

Function

TR `scmatrix::cond () const throw (cvmexception);`returns condition number reciprocal of calling matrix *A* in the **infinity-norm**:

$$\kappa_{\infty} = \|A\|_{\infty} \|A^{-1}\|_{\infty}.$$

Less value returned means that matrix *A* is closer to singular. Zero value returned means estimation underflow or that matrix *A* is singular. The condition number is used for error analysis of systems of linear equations. Function throws **cvmexception** in case of LAPACK subroutines failure. Function is *inherited* in the classes **scbmatrix** and **schmatrix**. See also **scmatrix::solve**, **scmatrix**. Example:

```
using namespace cvm;

try {
    double re[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
    double im[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};

    scmatrix m(re, im, 3);
    std::cout << m.cond() << std::endl
              << m.det() << std::endl << std::endl;

    m(3,3) = std::complex<double>(9.,10.);
    std::cout << m.cond() << std::endl << m.det() << std::endl;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints

1.54198e-018
(1.33227e-015,-1.33227e-015)

0.0050679
(6,-1.33227e-015)
```

**2.9.62 inv**

Functions

```
scmatrix& scmatrix::inv (const scmatrix& m) throw (cvmexception);
scmatrix scmatrix::inv () const throw (cvmexception);
```

implement matrix inversion. The first version sets calling matrix to be equal to *m* inverted and the second one creates object of type *scmatrix* as inverted calling matrix. Functions throw *cvmexception* in case of inappropriate sizes of the operands or when the matrix to be inverted is close to singular. Function is *redefined* in the class *schmatrix* and *inherited* in *scbmatrix*. See also *scmatrix*. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    double re[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
    double im[] = {1., 2., 3., 4., 5., 6., 7., 8., 10.};
    scmatrix m(re, im, 3);
    scmatrix mi(3);

    mi.inv (m);
    std::cout << mi << std::endl << mi * m - eye_complex(3);
    std::cout << std::endl << mi.inv() * mi - eye_complex(3);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(-8.33e-001,-1.67e-001) (6.67e-001,1.33e+000) (4.81e-016,-1.00e+000)
(3.33e-001,1.67e+000) (-1.67e-001,-3.83e+000) (-5.37e-016,2.00e+000)
(2.22e-016,-1.00e+000) (-4.44e-016,2.00e+000) (2.22e-016,-1.00e+000)

(-1.11e-016,1.11e-016) (0.00e+000,2.22e-016) (8.33e-017,1.26e-015)
(4.44e-016,-5.00e-016) (0.00e+000,7.77e-016) (7.22e-016,-1.15e-015)
(-1.11e-016,1.11e-016) (2.22e-016,-6.66e-016) (0.00e+000,1.11e-016)

(4.44e-016,4.44e-016) (-1.18e-015,1.33e-015) (6.66e-016,-7.77e-016)
(-6.85e-016,8.88e-016) (1.33e-015,3.16e-030) (-8.33e-016,-1.77e-030)
(-5.09e-016,6.66e-016) (7.96e-016,-1.78e-015) (-4.44e-016,1.11e-016)
```

**2.9.63 exp**

Functions

```
scmatrix& scmatrix::exp (const scmatrix& m, TR tol = cvmMachSp ())
throw (cvmexception);
scmatrix scmatrix::exp (TR tol = cvmMachSp ()) const
throw (cvmexception);
```

compute exponent of calling matrix using Padé approximation defined as

$$R_{pq}(z) = D_{pq}(z)^{-1} N_{pq}(z) = 1 + z + \dots + z^p/p!,$$

where

$$N_{pq}(z) = \sum_{k=0}^p \frac{(p+q-k)!p!}{(p+q)!k!(q-k)!} z^k,$$

$$D_{pq}(z) = \sum_{k=0}^q \frac{(p+q-k)!p!}{(p+q)!k!(q-k)!} (-z)^k$$

along with the matrix normalizing as described in [2], p. 572. Functions use ZMEXP (or CMEXP for float version) FORTRAN subroutine implementing the algorithm. The first version sets calling matrix to be equal to exponent of *m* and returns reference to the matrix changed. The second version creates object of type *scmatrix* as exponent of calling matrix. The algorithm uses parameter *tol* as  $\varepsilon(p, q)$  in order to choose constants *p* and *q* so that

$$\varepsilon(p, q) \geq 2^{3-(p+q)} \frac{p!q!}{(p+q)!(p+q+1)!}.$$

This parameter is equal to the **largest relative spacing** by default. Functions throw **cvmexception** in case of inappropriate sizes of the operands or when LAPACK subroutine fails. Functions are *inherited* in the classes **scbmatrix** and **schmatrix**. The second version is *redefined* in **scbmatrix**. See also **scmatrix**. Example:

```
using namespace cvm;

std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (15);
try {
    scmatrix m(2);
    m(1,1) = std::complex<double>(-49.,1.);
    m(1,2) = std::complex<double>(24.,1.);
    m(2,1) = std::complex<double>(-64.,1.);
    m(2,2) = std::complex<double>(31.,1.);
```

```
m = m.exp();
std::cout << m(1,1) << std::endl << "  "
          << m(1,2) << std::endl;
std::cout << m(2,1) << std::endl << "  "
          << m(2,2) << std::endl;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(-4.508497580070061e-001,7.900659666739228e-001)
(3.199576050798058e-001,-6.081804753524478e-001)
(-7.584316151932173e-001,1.666747485117903e+000)
(5.295040786048336e-001,-1.278050361026397e+000)
```

MATLAB output:

Column 1

```
-4.508497580070262e-001 +7.900659666739607e-001i
-7.584316151932523e-001 +1.666747485117982e+000i
```

Column 2

```
3.199576050798204e-001 -6.081804753524764e-001i
5.295040786048589e-001 -1.278050361026457e+000i
```

**2.9.64 polynomial**

Functions

```
scmatrix& scmatrix::polynom (const scmatrix& m, const cvector& v)
throw (cvmexception);
```

```
scmatrix scmatrix::polynom (const cvector& v) const
throw (cvmexception);
```

compute matrix polynomial defined as

$$p(A) = b_0I + b_1A + \dots + b_qA^q$$

using the Horner's rule:

$$p(A) = \sum_{k=0}^r B_k(A^s)^k, \quad s = \text{floor}(\sqrt{q}), \quad r = \text{floor}(q/s)$$

where

$$B_k = \begin{cases} \sum_{i=0}^{s-1} b_{sk+i} A^i, & k = 0, 1, \dots, r-1 \\ \sum_{i=0}^{q-sr} b_{sr+i} A^i, & k = r. \end{cases}$$

See also [2], p. 568. The coefficients  $b_0, b_1, \dots, b_q$  are passed in parameter  $v$ , where  $q$  is equal to  $v.size()-1$ , so functions compute matrix polynomial equal to

$$v[1] * I + v[2] * m + \dots + v[v.size()] * m^{v.size()-1}$$

The first version sets calling matrix to be equal to the polynomial of  $m$  and the second one creates object of type *scmatrix* as the polynomial of calling matrix. Functions use ZPOLY (or CPOLY for float version) FORTRAN subroutine implementing the Horner's algorithm. Functions throw *cvmexception* in case of inappropriate sizes of the operands. Functions are *inherited* in the class *scbmatrix* and *redefined* in *schmatrix*. See also *scmatrix*. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (15);
try {
    const double re[] = {2.2, 1.3, 1.1, -0.9, 0.2,
                        -0.45, 45., -30., 10., 3., 0.};
    const double im[] = {0.5, -2, 0, 1, 3,
                        -3., 30., 0., -9., 0., 1.};
```

```

const cvector v(re, im, 11);
scmatrix m(2), mp(2);
m(1,1) = std::complex<double>(0.1, -0.2);
m(1,2) = std::complex<double>(0.1, -0.2);
m(2,1) = std::complex<double>(0.5, -0.6);
m(2,2) = std::complex<double>(0.3, -0.4);

mp.polynom (m, v);
std::cout << mp(1,1) << std::endl << "  "
          << mp(1,2) << std::endl;
std::cout << mp(2,1) << std::endl << "  "
          << mp(2,2) << std::endl;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

```

prints

```

(2.485652665600000e+000,3.791263308800001e+000)
(2.8177861760000004e-001,2.301942860800001e+000)
(-8.835069888000001e-001,8.052028620800002e+000)
(1.903009862400001e+000,6.666306188800002e+000)

```

MATLAB output:

Column 1

```

2.485652665600000e+000 +3.791263308800001e+000i
-8.835069887999991e-001 +8.052028620800002e+000i

```

Column 2

```

2.817786176000000e-001 +2.301942860800001e+000i
1.903009862399999e+000 +6.666306188800003e+000i

```



**2.9.65 eig**

Functions

```
cvector scmatrix::eig (scmatrix& mEigVect, bool bRightVect = true) const
throw (cvmexception);
```

```
cvector scmatrix::eig () const throw (cvmexception);
```

solve the **nonsymmetric eigenvalue problem** and return complex vector with eigenvalues of calling matrix. The first version sets output parameter `mEigVect` to be equal to square matrix containing right (if parameter `bRightVect` is true, which is default value) or left (if parameter `bRightVect` is false) eigenvectors as columns. All functions throw **cvmexception** in case of inappropriate sizes of the operands or in case of convergence error. Functions are *inherited* in the class **scbmatrix** and *redefined* in **schmatrix**. See also **cvector**, **scmatrix**. Example:

```
using namespace cvm;
try {
    double re[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
    double im[] = {1., 2., 3., 4., 5., 6., 7., 8., 10.};
    scmatrix m(re, im, 3);
    scmatrix me(3);
    cvector vl(3);

    vl = m.eig (me);
    std::cout << vl << std::endl;
    std::cout.setf (std::ios::scientific | std::ios::left);
    std::cout.precision (2);
    std::cout << m * me(1) - me(1) * vl(1);
    std::cout << m * me(2) - me(2) * vl(2);
    std::cout << m * me(3) - me(3) * vl(3);
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
```

prints

```
(16.1096,16.7004) (-1.09351,-0.88358) (-0.0161248,0.183218)
(-9.44e-016,-3.55e-015) (-1.11e-016,-1.07e-014) (2.66e-015,-1.42e-014)
(-5.55e-016,-4.44e-016) (-1.80e-015,9.44e-016) (-2.00e-015,7.22e-016)
(9.92e-016,1.24e-015) (1.05e-015,2.78e-017) (1.64e-015,9.30e-016)
```

**2.9.66 Cholesky**

Function

```
scmatrix& scmatrix::cholesky (const scmatrix& m)
throw (cvmexception);
```

forms the Cholesky factorization of hermitian positive-definite matrix  $A$  defined as

$$A = U^T U,$$

where  $U$  is upper triangular matrix. It utilizes one of ?POTRF routines of the [LAPACK library](#). Function sets calling matrix to be equal to the factorization of hermitian positive-definite matrix  $m$ . Function throws `cvmexception` in case of inappropriate sizes of the operands or in case of convergence error. See also `scmatrix` and `schmatrix`. Example:

```
using namespace cvm;

try {
    double r[] = {1., 2., 1., 2., 15., -1., 1., -1., 20.};
    double i[] = {0., -1., 2., 1., 0., 3., -2., -3., 0.};
    const scmatrix m(r, i, 3);
    scmatrix c(3);

    c.cholesky(m);

    std::cout << c << std::endl;
    std::cout << ~c * c - m;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
}
```

prints

```
(1,0) (2,1) (1,-2)
(0,-0) (3.16228,0) (-0.316228,0.632456)
(0,0) (-0,0) (3.80789,0)

(0,0) (0,0) (0,0)
(0,0) (1.77636e-015,0) (0,0)
(0,0) (0,0) (0,0)
```

### 2.9.67 Bunch-Kaufman

Function

```
scmatrix& scmatrix::bunch_kaufman (const schmatrix& m, int* pivots)  
throw (cvmexception);
```

forms the Bunch-Kaufman factorization of hermitian matrix (cited from the MKL library documentation):

$$A = PUDU^T P^T,$$

where  $A$  is the input matrix passed in parameter  $m$ ,  $P$  is permutation matrix,  $U$  and  $L$  are upper and lower triangular matrices with unit diagonal, and  $D$  is symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks.  $U$  and  $L$  have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of  $D$ . It utilizes one of ?SYTRF routines of the [LAPACK library](#). Function sets calling matrix to be equal to the factorization of hermitian positive-definite matrix  $m$ . Function throws `cvmexception` in case of inappropriate sizes of the operands or in case of convergence error. See also `scmatrix` and `schmatrix`. Function is mostly designed to be used for subsequent calls of ?SYTRS, ?SYCON and ?SYTRI routines of the [LAPACK library](#). Currently it's used internally in `scmatrix::det` flow when argument is hermitian but not positive-definite matrix.

**2.9.68 qr**

Function

```
void scmatrix::qr (scmatrix& mQ, scmatrix& mR) const throw (cvmexception);
```

computes QR factorization as

$$M = QR$$

where  $M$  is calling square matrix, unitary matrix  $Q$  and upper triangular matrix  $R$  are  $mQ$  and  $mR$  respectively. Function throws `cvmexception` in case of inappropriate sizes of the operands. See also `cmatrix`, `cmatrix::qr`, `scmatrix`. Example:

```
using namespace cvm;
```

```
treal ar[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
treal ai[] = {1., -1., 2., -2., 3., -3., 4., -4., 5.};
const cvm::scmatrix m(ar, ai, 3);
cvm::scmatrix q(3), r(3);
```

```
m.qr(q,r);
std::cout << (eye_complex(3) - ~q * q).norm()
           << " " << (m - q * r).norm() << std::endl;
```

prints

```
+3.7957075e-16 +1.9641850e-15
```

**2.9.69 lq**

Function

```
void scmatrix::lq (scmatrix& mL, scmatrix& mQ) const throw (cvmexception);
```

computes LQ factorization as

$$M = LQ$$

where  $M$  is calling square matrix, lower triangular matrix  $L$  and unitary matrix  $Q$  are  $mL$  and  $mQ$  respectively. Function throws `cvmexception` in case of inappropriate sizes of the operands. See also `cmatrix`, `cmatrix::lq`, `scmatrix`. Example:

```
using namespace cvm;
```

```
treal ar[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
treal ai[] = {1., -1., 2., -2., 3., -3., 4., -4., 5.};
const cvm::scmatrix m(ar, ai, 3);
cvm::scmatrix l(3), q(3);
```

```
m.lq(l,q);
std::cout << (eye_complex(3) - ~q * q).norm()
           << " " << (m - l * q).norm() << std::endl;
```

prints

```
+3.5331153e-016 +3.0526086e-015
```

**2.9.70** `rq`

Function

```
void scmatrix::rq (scmatrix& mR, scmatrix& mQ) const throw (cvmexception);
```

computes RQ factorization as

$$M = RQ$$

where  $M$  is calling square matrix, upper triangular matrix  $R$  and unitary matrix  $Q$  are `mR` and `mQ` respectively. Function throws `cvmexception` in case if inappropriate sizes of the operands passed. See also `cmatrix`, `cmatrix::rq`, `srmatrix`. Example:

```
using namespace cvm;
```

```
treal ar[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
treal ai[] = {1., -1., 2., -2., 3., -3., 4., -4., 5.};
const cvm::scmatrix m(ar, ai, 3);
cvm::scmatrix r(3), q(3);
```

```
m.rq(r,q);
std::cout << (eye_complex(3) - ~q * q).norm()
           << " " << (m - r * q).norm() << std::endl;
```

prints

```
+3.3357248e-016 +3.4488467e-015
```

**2.9.71 ql**

Function

```
void scmatrix::ql (scmatrix& mQ, scmatrix& mL) const throw (cvmexception);
```

computes QL factorization as

$$M = QL$$

where  $M$  is calling square matrix, unitary matrix  $Q$  and lower triangular matrix  $L$  are  $mQ$  and  $mL$  respectively. Function throws **cvmexception** in case if inappropriate sizes of the operands passed. See also **cmatrix**, **cmatrix::ql**, **scmatrix**. Example:

```
using namespace cvm;
```

```
treal ar[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.};
treal ai[] = {1., -1., 2., -2., 3., -3., 4., -4., 5.};
const cvm::scmatrix m(ar, ai, 3);
cvm::scmatrix q(3), l(3);
```

```
m.ql(q,l);
std::cout << (eye_complex(3) - ~q * q).norm()
           << " " << (m - q * l).norm() << std::endl;
```

prints

```
+8.8904536e-016 +4.5789129e-015
```

### 2.9.72 **identity**

Function

```
scmatrix& scmatrix::identity();
```

sets calling matrix to be equal to identity matrix and returns reference to the matrix changed. Function is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix`. Example:

```
using namespace cvm;
```

```
scmatrix m(3);  
m.randomize_real(0.,3.);  
m.randomize_imag(-1.,2.);
```

```
std::cout << m << std::endl;  
std::cout << m.identity();
```

prints

```
(1.31162,-0.52501) (2.8612,-0.531144) (1.31849,0.547838)  
(1.19929,1.48253) (0.535417,0.41316) (0.459883,1.7019)  
(0.415937,-0.491134) (2.0969,-0.218024) (0.545305,1.17866)  
  
(1,0) (0,0) (0,0)  
(0,0) (1,0) (0,0)  
(0,0) (0,0) (1,0)
```



### 2.9.73 **vanish**

Function

```
scmatrix& scmatrix::vanish();
```

sets every element of calling matrix to be equal to zero and returns reference to the matrix changed. This function is faster than `scmatrix::set(TC)` with zero operand passed. Function is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix`. Example:

```
using namespace cvm;
```

```
scmatrix m(3);  
m.randomize_real(0.,3.);  
m.randomize_imag(-1.,2.);
```

```
std::cout << m << std::endl;  
std::cout << m.vanish();
```

prints

```
(1.34834,-0.758385) (0.837825,-0.225532) (0.367687,0.791833)  
(2.23698,-0.183142) (2.6878,0.741111) (0.495865,0.698904)  
(0.584124,0.00491348) (1.31574,0.687643) (0.482131,1.66482)  
  
(0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0)
```

### 2.9.74 `randomize_real`

Function

```
scmatrix& scmatrix::randomize_real (TR dFrom, TR dTo);
```

fills real part of calling matrix with pseudo-random numbers distributed between `dFrom` and `dTo`. Function returns reference to the matrix changed. Function is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix`. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);  
scmatrix m(3);  
m.randomize_real(0.,3.);  
std::cout << m;
```

prints

```
(1.56e+000,0.00e+000) (2.39e+000,0.00e+000) (2.41e+000,0.00e+000)  
(3.73e-002,0.00e+000) (2.61e+000,0.00e+000) (1.36e+000,0.00e+000)  
(2.71e+000,0.00e+000) (1.69e+000,0.00e+000) (2.68e+000,0.00e+000)
```

### 2.9.75 `randomize_imag`

Function

```
scmatrix& scmatrix::randomize_imag (TR dFrom, TR dTo);
```

fills imaginary part of calling matrix with pseudo-random numbers distributed between `dFrom` and `dTo`. Function returns reference to the matrix changed. Function is *redefined* in the classes `scbmatrix` and `schmatrix`. See also `scmatrix`. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
```

```
std::cout.precision (2);
```

```
scmatrix m(3);
```

```
m.randomize_imag(0.,3.);
```

```
std::cout << m;
```

prints

```
(0.00e+000,1.58e+000) (0.00e+000,2.38e+000) (0.00e+000,2.64e+000)
```

```
(0.00e+000,1.62e-002) (0.00e+000,4.26e-002) (0.00e+000,2.21e+000)
```

```
(0.00e+000,2.39e+000) (0.00e+000,6.95e-001) (0.00e+000,4.30e-001)
```

## 2.10 BandMatrix

This base class contains member functions common for band matrices. This class is not designed to be instantiated.

```
template <typename TR, typename TC>
class BandMatrix {
public:
    int lsize () const;
    int usize () const;
};
```

### 2.10.1 lsize

Function

```
int BandMatrix<TR,TC>::lsize () const;
```

returns number of lower sub-diagonals of calling matrix. Function is *inherited* in the classes `srbmatrix` and `scbmatrix`. See also `BandMatrix`. Example:

```
using namespace cvm;
```

```
srbmatrix m(4,2,1);  
m.set(1.);  
std::cout << m;  
std::cout << m.msize() << " " << m.nsize() << " " << m.size();  
std::cout << " " << m.lsize() << " " << m.usize() << std::endl;
```

prints

```
1 1 0 0  
1 1 1 0  
1 1 1 1  
0 1 1 1  
4 4 16 2 1
```

### 2.10.2 `usize`

Function

```
int BandMatrix<TR,TC>::usize () const;
```

returns number of upper sub-diagonals of calling matrix. Function is *inherited* in the classes `srbmatrix` and `scbmatrix`. See also `BandMatrix`. Example:

```
using namespace cvm;
```

```
srbmatrix m(4,2,1);  
m.set(1.);  
std::cout << m;  
std::cout << m.msize() << " " << m.nsize() << " " << m.size() ;  
std::cout << " " << m.lsize() << " " << m.usize() << std::endl;
```

prints

```
1 1 0 0  
1 1 1 0  
1 1 1 1  
0 1 1 1  
4 4 16 2 1
```

## 2.11 srbmatrix

This is end-user class encapsulating square band matrix of real numbers. This class utilizes [band storage](#) for its elements.

```
template <typename TR>
class srbmatrix : public srmatrix <TR>, public BandMatrix <TR,TR> {
public:
    srbmatrix ();
    explicit srbmatrix (int nMN);
    srbmatrix (int nMN);
    srbmatrix (TR* pD, int nMN, int nKL, int nKU);
    srbmatrix (const TR* pD, int nMN, int nKL, int nKU);
    srbmatrix (const srbmatrix& m);
    srbmatrix (const rmatrix& m, int nKL, int nKU);
    explicit srbmatrix (const rvector& v);
    TR& operator () (int im, int in) throw (cvmexception);
    TR operator () (int im, int in) const throw (cvmexception);
    const rvector operator () (int i) const throw (cvmexception);
    const rvector operator [] (int i) const throw (cvmexception);
    srbmatrix& operator = (const srbmatrix& m) throw (cvmexception);
    srbmatrix& assign (const rvector& v) throw (cvmexception);
    srbmatrix& assign (const TR* pD);
    srbmatrix& set (TR x);
    srbmatrix& resize (int nNewMN) throw (cvmexception);
    srbmatrix& resize_lu (int nNewKL, int nNewKU) throw (cvmexception);
    bool operator == (const srbmatrix& v) const;
    bool operator != (const srbmatrix& v) const;
    srbmatrix& operator << (const srbmatrix& m) throw (cvmexception);
    srbmatrix operator + (const srbmatrix& m) const
        throw (cvmexception);
    srbmatrix operator - (const srbmatrix& m) const
        throw (cvmexception);
    srbmatrix& sum (const srbmatrix& m1,
        const srbmatrix& m2) throw (cvmexception);
    srbmatrix& diff (const srbmatrix& m1,
        const srbmatrix& m2) throw (cvmexception);
    srbmatrix& operator += (const srbmatrix& m) throw (cvmexception);
    srbmatrix& operator -= (const srbmatrix& m) throw (cvmexception);
    srbmatrix operator - () const;
    srbmatrix& operator ++ ();
    srbmatrix& operator ++ (int);
    srbmatrix& operator -- ();
```

```
srbmatrix& operator -- (int);
srbmatrix operator * (TR d) const;
srbmatrix operator / (TR d) const throw (cvmexception);
srbmatrix& operator *= (TR d);
srbmatrix& operator /= (TR d) throw (cvmexception);
srbmatrix& normalize ();
srbmatrix operator ~ () const throw (cvmexception);
srbmatrix& transpose (const srbmatrix& m) throw (cvmexception);
srbmatrix& transpose () throw (cvmexception);
rvector operator * (const rvector& v) const throw (cvmexception);
rmatrix operator * (const rmatrix& m) const throw (cvmexception);
srmatrix operator * (const srmatrix& m) const throw (cvmexception);
srbmatrix operator * (const srbmatrix& m) const throw (cvmexception);
srbmatrix& low_up (const srbmatrix& m,
                  int* nPivots) throw (cvmexception);
srbmatrix low_up (int* nPivots) const throw (cvmexception);
rvector operator / (const rvector& vB) const throw (cvmexception);
srbmatrix& identity ();
srbmatrix& vanish ();
srbmatrix& randomize (TR dFrom, TR dTo);
};
```



### 2.11.1 srbmatrix ()

Constructor

```
srbmatrix::srbmatrix ();
```

creates empty srbmatrix object. See also [srbmatrix](#). Example:

```
using namespace cvm;
```

```
srbmatrix m;  
std::cout << m.msize() << " " << m.nsize() << " " << m.size();  
std::cout << " " << m.lsize() << " " << m.usize() << std::endl;
```

```
m.resize (3);  
m.resize_lu(1,0);  
m.set(1.);  
std::cout << m;
```

prints

```
0 0 0 0 0  
1 0 0  
1 1 0  
0 1 1
```

### 2.11.2 srbmatrix (int)

Constructor

```
explicit srbmatrix::srbmatrix (int nMN);
```

creates  $n \times n$  srbmatrix object where  $n$  is passed in `nMN` parameter. The matrix created is diagonal, i.e.  $k_l = k_u = 0$ . Constructor throws `cvmexception` in case of non-positive size passed or memory allocation failure. See also `srbmatrix`. Example:

```
using namespace cvm;
```

```
srbmatrix m(4);  
std::cout << m.msize() << " " << m.nsize() << " " << m.size();  
std::cout << " " << m.lsize() << " " << m.usize() << std::endl;
```

```
m.set(1.);  
std::cout << m;
```

prints

```
4 4 4 0 0  
1 0 0 0  
0 1 0 0  
0 0 1 0  
0 0 0 1
```

**2.11.3 srbmatrix (int,int,int)**

Constructor

```
srbmatrix::srbmatrix (int nMN, int nKL, int nKU);
```

creates  $n \times n$  srbmatrix object where  $n$  is passed in nMN parameter. The matrix created has nKL sub-diagonals and nKU super-diagonals. Constructor throws `cvmexception` in case of non-positive size or negative number of sub-diagonals or super-diagonals passed or in case of memory allocation failure. See also `srbmatrix`. Example:

```
using namespace cvm;
```

```
srbmatrix m(5,1,1);
m.set(1.);
std::cout << m << std::endl
           << m.msize() << " " << m.nsize() << " " << m.size()
           << " " << m.lsize() << " " << m.usize() << std::endl;
```

prints

```
1 1 0 0 0
1 1 1 0 0
0 1 1 1 0
0 0 1 1 1
0 0 0 1 1

5 5 15 1 1
```

**2.11.4 srbmatrix (TR\*,int,int,int)**

Constructor

```
srbmatrix::srbmatrix (TR* pD, int nMN, int nKL, int nKU);
```

creates  $n \times n$  srbmatrix object where  $n$  is passed in nMN parameter. The matrix created has  $k_l=nKL$  sub-diagonals and  $k_u=nKU$  super-diagonals. Unlike others, this constructor *does not allocate memory*. It just shares memory with array pointed to by pD. Please note that this array must contain at least  $(k_l + k_u + 1) * n$  elements. Constructor throws `cvmexception` in case of non-positive size or negative number of sub- or super-diagonals passed. See also `srbmatrix`, `srbmatrix (const TR*,int,int,int)`. Example:

```
using namespace cvm;
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
srbmatrix ml(a,4,1,0);
srbmatrix mu(a,4,0,1);
ml(2,1) = 9.;
std::cout << ml << std::endl << mu << std::endl;
std::cout << a[0] << " " << a[1] << " " << a[2] << " "
          << a[3] << " " << a[4] << " " << a[5] << " " << std::endl;
```

prints

```
1 0 0 0
9 3 0 0
0 4 5 0
0 0 6 7

9 3 0 0
0 4 5 0
0 0 6 7
0 0 0 8

1 9 3 4 5 6
```

**2.11.5 srbmatrix (const TR\*,int,int,int)**

Constructor

```
srbmatrix::srbmatrix (const TR* pD, int nMN, int nKL, int nKU);
```

creates  $n \times n$  srbmatrix object where  $n$  is passed in nMN parameter. The matrix created has  $k_l=nKL$  sub-diagonals and  $k_u=nKU$  super-diagonals. Then constructor copies  $(k_l + k_u + 1) * n$  elements of array pD to the matrix according to **band storage**. Constructor throws **cvmexception** in case of non-positive size or negative number of sub- or super-diagonals passed. See also **srbmatrix**, **srbmatrix (TR\*,int,int,int)**. Example:

```
using namespace cvm;
const double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
srbmatrix ml(a,4,1,0);
srbmatrix mu(a,4,0,1);
ml(2,1) = 9.;
std::cout << ml << std::endl << mu << std::endl;
std::cout << a[0] << " " << a[1] << " " << a[2] << " "
          << a[3] << " " << a[4] << " " << a[5] << " " << std::endl;
```

prints

```
1 0 0 0
9 3 0 0
0 4 5 0
0 0 6 7
```

```
2 3 0 0
0 4 5 0
0 0 6 7
0 0 0 8
```

```
1 2 3 4 5 6
```

### 2.11.6 srbmatrix (const srbmatrix&)

Copy constructor

```
srbmatrix::srbmatrix (const srbmatrix& m);
```

creates srbmatrix object as a copy of m. Constructor throws `cvmexception` in case of memory allocation failure. See also `srbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
srbmatrix m(a,4,1,0);  
srbmatrix mc(m);  
m(1,1) = 7.77;  
std::cout << m << std::endl << mc;
```

prints

```
7.77 0 0 0  
2 3 0 0  
0 4 5 0  
0 0 6 7
```

```
1 0 0 0  
2 3 0 0  
0 4 5 0  
0 0 6 7
```

### 2.11.7 srbmatrix (const rmatrix&,int,int)

Constructor

```
srbmatrix::srbmatrix (const rmatrix& m, int nKL, int nKU);
```

creates srbmatrix object as a copy of “sliced” matrix *m*, i.e. it copies main diagonal, *nKL* sub-diagonals and *nKU* super-diagonals of matrix *m*. It’s assumed that  $m \times n$  matrix *m* must have equal sizes, i.e.  $m = n$  is satisfied. Constructor throws `cvmexception` if this is not true or in case of memory allocation failure. See also `srbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,  
             10., 11., 12., 13., 14., 15., 16.};
```

```
srmatrix m(a,4);
```

```
srbmatrix mb(m,1,2);
```

```
std::cout << m << std::endl << mb;
```

prints

```
1 5 9 13  
2 6 10 14  
3 7 11 15  
4 8 12 16
```

```
1 5 9 0  
2 6 10 14  
0 7 11 15  
0 0 12 16
```

### 2.11.8 srbmatrix (const rvector&)

Constructor

```
explicit srbmatrix::srbmatrix (const rvector& v);
```

creates srbmatrix object of size `v.size()` by `v.size()` and assigns vector `v` to its main diagonal. Constructor throws `cvmexception` in case of memory allocation failure. See also `srbmatrix`, `rvector`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5.};  
rvector v(a, 5);  
srbmatrix m(v);  
std::cout << m;
```

prints

```
1 0 0 0 0  
0 2 0 0 0  
0 0 3 0 0  
0 0 0 4 0  
0 0 0 0 5
```



### 2.11.9 operator (,)

Indexing operators

```
TR& srbmatrix::operator () (int im, int in) throw (cvmexception);
TR srbmatrix::operator () (int im, int in) const throw (cvmexception);
```

provide access to a particular element of calling band matrix. The first version is applicable to non-constant objects. This version returns *l-value* in order to make possible write access to an element. Only elements located on main diagonal or on non-zero sub- or super-diagonals are l-values. All other values located outside this area are not writable. Both operators are *l-based*. Operators throw `cvmexception` if some of parameters passed is outside of `[1,msize()]` range or in case of attempt to write to a non-writable element<sup>9</sup>. See also `srbmatrix`, `BandMatrix::lsize()` and `BandMatrix::usize()`. Example:

```
using namespace cvm;

std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
    srbmatrix m (a,4,1,0);

    m(2,2) = 7.77;
    std::cout << m << std::endl;
    std::cout << m(3,2) << " " << m(1,4) << std::endl;

    m(1,3) = 7.77;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
1.00e+000 0.00e+000 0.00e+000 0.00e+000
2.00e+000 7.77e+000 0.00e+000 0.00e+000
0.00e+000 4.00e+000 5.00e+000 0.00e+000
0.00e+000 0.00e+000 6.00e+000 7.00e+000
```

```
4.00e+000 0.00e+000
```

```
Exception: Attempt to change a read-only element
```

<sup>9</sup>Here I use `type_proxy<T>` class originally described in [4], p. 217.

### 2.11.10 operator ()

Indexing operator

```
const rvector srbmatrix::operator () (int i) const throw (cvmexception);
```

provides access to *i*-th column of calling band matrix. Unlike `srmatrix::operator ()`, this operator creates only *copy* of a column and therefore it returns *not l-value*. Operator is *l-based*. It throws `cvmexception` if parameter *i* is outside of `[1, nsize()]` range. See also `srbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
srbmatrix m (a,4,1,0);
```

```
std::cout << m << std::endl;
```

```
std::cout << m(2);
```

prints

```
1 0 0 0
```

```
2 3 0 0
```

```
0 4 5 0
```

```
0 0 6 7
```

```
0 3 4 0
```

### 2.11.11 operator []

Indexing operator

```
const rvector srbmatrix::operator [] (int i) const throw (cvmexception);
```

provides access to *i*-th row of calling band matrix. Unlike `srmatrix::operator []`, this operator creates only *copy* of a column and therefore it returns *not l-value*. Operator is *l-based*. It throws `cvmexception` if parameter *i* is outside of `[1, nsize()]` range. See also `srbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
srbmatrix m (a,4,1,0);
```

```
std::cout << m << std::endl;
```

```
std::cout << m[2];
```

prints

```
1 0 0 0
```

```
2 3 0 0
```

```
0 4 5 0
```

```
0 0 6 7
```

```
2 3 0 0
```

### 2.11.12 operator = (const srbmatrix&)

Operator

```
srbmatrix& srbmatrix::operator = (const srbmatrix& m)
throw (cvmexception);
```

sets every element of calling band matrix to value of appropriate element of band matrix `m` and returns reference to the matrix changed. Operator throws `cvmexception` in case of different matrix sizes or in case of different numbers of sub- or super-diagonals. See also `srbmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
    const srbmatrix m1(a,4,1,0);
    srbmatrix m2(4,1,0);

    m2 = m1;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
1 0 0 0
2 3 0 0
0 4 5 0
0 0 6 7
```

### 2.11.13 assign (const TR\*)

Function

```
srbmatrix& srbmatrix::assign (const rvector& v) throw (cvmexception);  
srbmatrix& srbmatrix::assign (const TR* pD);
```

sets every element of calling band matrix to value of appropriate element of vector *v* or array pointed to by *pD* and returns reference to the matrix changed. In first version it's assumed that vector passed is long enough to fill calling matrix. Function throws `cvmexception` otherwise. In other words this array must contain at least  $(k_l + k_u + 1)n$  elements. See also `srbmatrix`. Example:

```
using namespace cvm;
```

```
const double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
srbmatrix m(4,0,1);
```

```
m.assign(a);  
std::cout << m;
```

prints

```
2 3 0 0  
0 4 5 0  
0 0 6 7  
0 0 0 8
```

### 2.11.14 set (TR)

Function

```
srbmatrix& srbmatrix::set (TR x);
```

sets every element of calling band matrix to value of parameter `x` and returns reference to the matrix changed. Use `vanish` to set every element of calling matrix to be equal to zero. See also `srbmatrix`. Example:

```
using namespace cvm;
```

```
srbmatrix m(8,2,1);  
m.set(3.);  
std::cout << m;
```

prints

```
3 3 0 0 0 0 0 0  
3 3 3 0 0 0 0 0  
3 3 3 3 0 0 0 0  
0 3 3 3 3 0 0 0  
0 0 3 3 3 3 0 0  
0 0 0 3 3 3 3 0  
0 0 0 0 3 3 3 3  
0 0 0 0 0 3 3 3
```

**2.11.15 resize**

Function

```
srbmatrix& srbmatrix::resize (int nNewMN) throw (cvmexception);
```

changes size of calling band matrix to nNewMN by nNewMN and returns reference to the matrix changed. In case of increasing of its size, calling matrix is filled up with zeroes. This function doesn't change number of sub- or super-diagonals. Like any band matrix class member function, this function doesn't change **non-referred elements**. See number 8 appearing after resize in example below. Function throws **cvmexception** in case of non-positive size passed or memory allocation failure. See also **srbmatrix.resize\_lu**, **srbmatrix**. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
    srbmatrix m(a,4,1,0);
    std::cout << m << std::endl;
    m.resize (5);
    std::cout << m;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
1 0 0 0
2 3 0 0
0 4 5 0
0 0 6 7

1 0 0 0 0
2 3 0 0 0
0 4 5 0 0
0 0 6 7 0
0 0 0 8 0
```

### 2.11.16 `resize_lu`

Function

```
srbmatrix& srbmatrix::resize_lu (int nNewKL, int nNewKU)
throw (cvmexception);
```

changes number of sub- and super-diagonals of calling band matrix to `nNewKL` and `nNewKU` respectively and returns reference to the matrix changed. In case of increasing of the numbers, calling matrix is filled up with zeroes. Function throws `cvmexception` in case of negative number passed or memory allocation failure. See also `srbmatrix::resize`, `srbmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
    srbmatrix m(a,4,1,0);
    std::cout << m << std::endl;
    m.resize_lu (0,1);
    m.diag(1).set(9.);
    std::cout << m;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
1 0 0 0
2 3 0 0
0 4 5 0
0 0 6 7
```

```
1 9 0 0
0 3 9 0
0 0 5 9
0 0 0 7
```



### 2.11.17 operator ==

Operator

```
bool srbmatrix::operator == (const srbmatrix& m) const;
```

compares calling band matrix with band matrix `m` and returns true if they have the same sizes, the same numbers of sub- and super-diagonals and their appropriate elements differ by not more than the **smallest normalized positive number**. Returns false otherwise. See also `srbmatrix`. Example:

```
using namespace cvm;

double a[] = {1., 2., 3., 4.};
srbmatrix m1(a,2,1,0);
srbmatrix m2(2,1,0);

std::cout << m1 << std::endl;

m2(1,1) = 1.;
m2(2,1) = 2.; m2(2,2) = 3.;

std::cout << (m1 == m2) << std::endl;

prints

1 0
2 3

1
```

**2.11.18 operator !=**

Operator

```
bool srbmatrix::operator != (const srbmatrix& m) const;
```

compares calling band matrix with band matrix `m` and returns true if they have different sizes, different numbers of sub- or super-diagonals or at least one of their appropriate elements differs by more than the **smallest normalized positive number**. Returns false otherwise. See also `srbmatrix`. Example:

```
using namespace cvm;

double a[] = {1., 2., 3., 4.};
srbmatrix m1(a,2,1,0);
srbmatrix m2(2,1,0);

std::cout << m1 << std::endl;

m2(1,1) = 1.;
m2(2,1) = 2.; m2(2,2) = 3.;

std::cout << (m1 != m2) << std::endl;

prints

1 0
2 3

0
```

### 2.11.19 operator <<

Operator

```
srbmatrix& srbmatrix::operator << (const srbmatrix& m)
throw (cvmexception);
```

destroys calling band matrix, creates new one as a copy of m and returns reference to the matrix changed. Operator throws **cvmexception** in case of memory allocation failure. See also **srbmatrix**. Example:

```
using namespace cvm;

try {
    srbmatrix m(3,1,0);
    srmatrix mc(1);
    m(2,1) = 1.;
    m(2,2) = 2.;
    std::cout << m << std::endl << mc << std::endl;

    mc << m;
    std::cout << mc;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}

prints

0 0 0
1 2 0
0 0 0

0

0 0 0
1 2 0
0 0 0
```

**2.11.20 operator +**

Operator

```
srbmatrix srbmatrix::operator + (const srbmatrix& m) const
throw (cvmexception);
```

creates object of type `srbmatrix` as sum of calling band matrix and band matrix `m`. It throws `cvmexception` in case of different sizes or different numbers of sub- or super-diagonals of the operands. See also `srbmatrix::sum`, `srbmatrix`. Example:

```
using namespace cvm;

std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (1);
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
    double b[] = {10., 20., 30., 40., 50., 60., 70., 80.};
    srbmatrix m1(a,4,1,0);
    srbmatrix m2(b,4,1,0);

    std::cout << m1 + m2 << std::endl << m1 + m1;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
1.1e+01 0.0e+00 0.0e+00 0.0e+00
2.2e+01 3.3e+01 0.0e+00 0.0e+00
0.0e+00 4.4e+01 5.5e+01 0.0e+00
0.0e+00 0.0e+00 6.6e+01 7.7e+01

2.0e+00 0.0e+00 0.0e+00 0.0e+00
4.0e+00 6.0e+00 0.0e+00 0.0e+00
0.0e+00 8.0e+00 1.0e+01 0.0e+00
0.0e+00 0.0e+00 1.2e+01 1.4e+01
```

**2.11.21 operator -**

Operator

```
srbmatrix srbmatrix::operator - (const srbmatrix& m) const
throw (cvmexception);
```

creates object of type `srbmatrix` as difference of calling band matrix and band matrix `m`. It throws `cvmexception` in case of different sizes or different numbers of sub- or super-diagonals of the operands. See also `srbmatrix::diff`, `srbmatrix`. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (1);
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
    double b[] = {10., 20., 30., 40., 50., 60., 70., 80.};
    srbmatrix m1(a,4,1,0);
    srbmatrix m2(b,4,1,0);

    std::cout << m2 - m1 << std::endl << m1 - m1;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
9.0e+00 0.0e+00 0.0e+00 0.0e+00
1.8e+01 2.7e+01 0.0e+00 0.0e+00
0.0e+00 3.6e+01 4.5e+01 0.0e+00
0.0e+00 0.0e+00 5.4e+01 6.3e+01

0.0e+00 0.0e+00 0.0e+00 0.0e+00
0.0e+00 0.0e+00 0.0e+00 0.0e+00
0.0e+00 0.0e+00 0.0e+00 0.0e+00
0.0e+00 0.0e+00 0.0e+00 0.0e+00
```

**2.11.22 sum**

Function

```
srbmatrix& srbmatrix::sum (const srbmatrix& m1, const srbmatrix& m2)
throw (cvmexception);
```

assigns result of addition of band matrices m1 and m2 to calling band matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes or different numbers of sub- or super-diagonals of the operands. See also `srbmatrix::operator +`, `srbmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6.};
    const srbmatrix m1(a,3,1,0);
    srbmatrix m2(3,1,0);
    srbmatrix m(3,1,0);
    m2.set(1.);
    std::cout << m1 << std::endl << m2 << std::endl;
    std::cout << m.sum(m1, m2) << std::endl;
    std::cout << m.sum(m, m2);
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
1 0 0
2 3 0
0 4 5
```

```
1 0 0
1 1 0
0 1 1
```

```
2 0 0
3 4 0
0 5 6
```

```
3 0 0
4 5 0
0 6 7
```

**2.11.23 diff**

Function

```
srbmatrix& srbmatrix::diff (const srbmatrix& m1, const srbmatrix& m2)
throw (cvmexception);
```

assigns result of subtraction of band matrices m1 and m2 to calling band matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes or different numbers of sub- or super-diagonals of the operands. See also `srbmatrix::operator -`, `srbmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6.};
    const srbmatrix m1(a,3,1,0);
    srbmatrix m2(3,1,0);
    srbmatrix m(3,1,0);
    m2.set(1.);
    std::cout << m1 << std::endl << m2 << std::endl;
    std::cout << m.diff(m1, m2) << std::endl;
    std::cout << m.diff(m, m2);
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
1 0 0
2 3 0
0 4 5
```

```
1 0 0
1 1 0
0 1 1
```

```
0 0 0
1 2 0
0 3 4
```

```
-1 0 0
0 1 0
0 2 3
```

**2.11.24 operator +=**

Operator

```
srbmatrix& srbmatrix::operator += (const srbmatrix& m)
throw (cvmexception);
```

adds band matrix `m` to calling band matrix and returns reference to the matrix changed. Operator throws `cvmexception` in case of different sizes or different numbers of sub- or super-diagonals of the operands. See also `srbmatrix::operator +` , `srbmatrix::sum`, `srbmatrix`. Example:

```
using namespace cvm;
```

```
try {
    srbmatrix m1(4,0,1);
    srbmatrix m2(4,0,1);
    m1.set(1.);
    m2.set(2.);

    m1 += m2;
    std::cout << m1 << std::endl;

    // well, you can do this too, but temporary object would be created
    m2 += m2;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
3 3 0 0
0 3 3 0
0 0 3 3
0 0 0 3
```

```
4 4 0 0
0 4 4 0
0 0 4 4
0 0 0 4
```



**2.11.25 operator -=**

Operator

```
srbmatrix& srbmatrix::operator -= (const srbmatrix& m)
throw (cvmexception);
```

subtracts band matrix `m` from calling band matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes or different numbers of sub- or super-diagonals of the operands. See also `srbmatrix::operator -`, `srbmatrix::diff`, `srbmatrix`. Example:

```
using namespace cvm;
```

```
try {
    srbmatrix m1(4,0,1);
    srbmatrix m2(4,0,1);
    m1.set(1.);
    m2.set(4.);

    m2 -= m1;
    std::cout << m2 << std::endl;

    // well, you can do this too, but temporary object would be created
    m2 -= m2;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
3 3 0 0
0 3 3 0
0 0 3 3
0 0 0 3

0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

### 2.11.26 operator - ()

Operator

```
srbmatrix srbmatrix::operator - () const throw (cvmexception);
```

creates object of type srbmatrix as calling band matrix multiplied by  $-1$ . See also [srbmatrix](#). Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific |  
                std::ios::left |  
                std::ios::showpos);
```

```
std::cout.precision (1);
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
```

```
srbmatrix m(a,4,1,0);
```

```
std::cout << -m;
```

prints

```
-1.0e+00 +0.0e+00 +0.0e+00 +0.0e+00  
-2.0e+00 -3.0e+00 +0.0e+00 +0.0e+00  
+0.0e+00 -4.0e+00 -5.0e+00 +0.0e+00  
+0.0e+00 +0.0e+00 -6.0e+00 -7.0e+00
```

### 2.11.27 operator ++

Operator

```
srbmatrix& srbmatrix::operator ++ ();  
srbmatrix& srbmatrix::operator ++ (int);
```

adds identity matrix to calling band matrix and returns reference to the matrix changed. See also [srbmatrix](#). Example:

```
using namespace cvm;  
  
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
srbmatrix m (a,4,1,0);  
  
m++;  
std::cout << m << std::endl;  
std::cout << ++m;  
  
prints  
  
2 0 0 0  
2 4 0 0  
0 4 6 0  
0 0 6 8  
  
3 0 0 0  
2 5 0 0  
0 4 7 0  
0 0 6 9
```

### 2.11.28 operator --

Operator

```
srbmatrix& srbmatrix::operator -- ();  
srbmatrix& srbmatrix::operator -- (int);
```

subtracts identity matrix from calling band matrix and returns reference to the matrix changed. See also [srbmatrix](#). Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
srbmatrix m (a,4,1,0);
```

```
m--;  
std::cout << m << std::endl;  
std::cout << --m;
```

prints

```
0 0 0 0  
2 2 0 0  
0 4 4 0  
0 0 6 6
```

```
-1 0 0 0  
2 1 0 0  
0 4 3 0  
0 0 6 5
```

**2.11.29 operator \* (TR)**

Operator

```
srbmatrix srbmatrix::operator * (TR d) const;
```

creates object of type srbmatrix as product of calling band matrix and number d. See also [srbmatrix::operator \\*=](#) , [srbmatrix](#). Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
srbmatrix m (a,4,1,0);  
std::cout << m * 5. << std::endl;
```

prints

```
5 0 0 0  
10 15 0 0  
0 20 25 0  
0 0 30 35
```

### 2.11.30 operator / (TR)

Operator

```
srbmatrix srbmatrix::operator / (TR d) const throw (cvmexception);
```

creates object of type srbmatrix as quotient of calling band matrix and number d. It throws **cvmexception** if d has absolute value equal or less than the **smallest normalized positive number**. See also **srbmatrix::operator /=** , **srbmatrix**. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
srbmatrix m (a,4,1,0);  
std::cout << m / 2. << std::endl;
```

prints

```
0.5 0 0 0  
1 1.5 0 0  
0 2 2.5 0  
0 0 3 3.5
```

### 2.11.31 operator \*= (TR)

Operator

```
srbmatrix& srbmatrix::operator *= (TR d);
```

multiplies calling band matrix by number d and returns reference to the matrix changed. See also `srbmatrix::operator *`, `srbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
srbmatrix m (a,4,1,0);  
m *= 2.;  
std::cout << m;
```

prints

```
2 0 0 0  
4 6 0 0  
0 8 10 0  
0 0 12 14
```

### 2.11.32 operator /= (TR)

Operator

```
srbmatrix& srbmatrix::operator /= (TR d) throw (cvmexception);
```

divides calling matrix by number d and returns reference to the matrix changed. It throws **cvmexception** if d has absolute value equal or less than the **smallest normalized positive number**. See also **srbmatrix::operator /** , **srbmatrix**. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
srbmatrix m (a,4,1,0);  
m /= 2.;  
std::cout << m;
```

prints

```
0.5 0 0 0  
1 1.5 0 0  
0 2 2.5 0  
0 0 3 3.5
```



### 2.11.33 normalize

Function

```
srbmatrix& srbmatrix::normalize ();
```

normalizes calling band matrix so its **Euclidean norm** becomes equal to 1 if it was greater than the **smallest normalized positive number** before the call (otherwise function does nothing). See also **srbmatrix**. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (3);  
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
srbmatrix m (a,4,1,0);
```

```
m.normalize();  
std::cout << m << m.norm() << std::endl;
```

prints

```
8.452e-02 0.000e+00 0.000e+00 0.000e+00  
1.690e-01 2.535e-01 0.000e+00 0.000e+00  
0.000e+00 3.381e-01 4.226e-01 0.000e+00  
0.000e+00 0.000e+00 5.071e-01 5.916e-01  
1.000e+00
```

**2.11.34 transposition**

Operator and functions

```
srbmatrix srbmatrix::operator ~ () const throw (cvmexception);
srbmatrix& srbmatrix::transpose (const srbmatrix& m) throw (cvmexception);
srbmatrix& srbmatrix::transpose () throw (cvmexception);
```

implement band matrix transposition. First operator creates object of type srbmatrix as transposed calling band matrix (it throws `cvmexception` in case of memory allocation failure). Second function sets calling band matrix to be equal to band matrix `m` transposed (it throws `cvmexception` in case of not appropriate sizes of the operands or in case of memory allocation failure), third one makes it to be equal to transposed itself. See also `srbmatrix`. Example:

```
using namespace cvm;

double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
srbmatrix m (a,4,1,0);
srbmatrix mt(4,0,1);
std::cout << ~m << std::endl ;
mt.transpose(m);
std::cout << mt << std::endl;
mt.transpose();
std::cout << mt;
```

prints

```
1 2 0 0
0 3 4 0
0 0 5 6
0 0 0 7
```

```
1 2 0 0
0 3 4 0
0 0 5 6
0 0 0 7
```

```
1 0 0 0
2 3 0 0
0 4 5 0
0 0 6 7
```

### 2.11.35 operator \* (const rvector&)

Operator

```
rvector srbmatrix::operator * (const rvector& v) const  
throw (cvmexception);
```

creates object of type rvector as product of calling band matrix and vector v. It throws **cvmexception** if number of columns of calling matrix differs from size of vector v. Use **rvector::mult** in order to avoid new object creation. See also **srbmatrix** and **rvector**.  
Example:

```
using namespace cvm;
```

```
try {  
    srbmatrix m (4,1,0);  
    rvector v(4);  
    m.set(1.);  
    v.set(1.);  
  
    std::cout << m * v;  
}  
catch (std::exception& e) {  
    std::cout << "Exception: " << e.what () << std::endl;  
}
```

prints

```
1 2 2 2
```

### 2.11.36 operator \* (const rmatrix&)

Operator

```
rmatrix srbmatrix::operator * (const rmatrix& m) const  
throw (cvmexception);
```

creates object of type `rmatrix` as product of calling band matrix and matrix `m`. It throws `cvmexception` if number of columns of calling matrix differs from number of rows of matrix `m`. Use `rmatrix::mult` in order to avoid new object creation. See also `rmatrix` and `srbmatrix`. Example:

```
using namespace cvm;
```

```
try {  
    srbmatrix mb (4,1,0);  
    rmatrix m(4,2);  
    mb.set(1.);  
    m.set(1.);  
  
    std::cout << mb * m;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
1 1  
2 2  
2 2  
2 2
```

### 2.11.37 operator \* (const srmatrix&)

Operator

```
srmatrix srbmatrix::operator * (const srmatrix& m) const  
throw (cvmexception);
```

creates object of type srmatrix as product of calling band matrix and matrix m. It throws `cvmexception` if the operands have different sizes. Use `rmatrix::mult` in order to avoid new object creation. See also `srmatrix` and `srbmatrix`. Example:

```
using namespace cvm;
```

```
try {  
    srbmatrix mb(4,1,0);  
    srmatrix ms(4);  
    mb.set(1.);  
    ms.set(1.);  
  
    std::cout << mb * ms;  
}  
catch (std::exception& e) {  
    std::cout << "Exception " << e.what () << std::endl;  
}
```

prints

```
1 1 1 1  
2 2 2 2  
2 2 2 2  
2 2 2 2
```

### 2.11.38 operator \* (const srbmatrix&)

Operator

```
srbmatrix srbmatrix::operator * (const srbmatrix& m) const  
throw (cvmexception);
```

creates object of type srbmatrix as product of calling band matrix and band matrix m. It throws `cvmexception` if the operands have different sizes. Use `rmatrix::mult` in order to avoid new object creation. See also `srbmatrix`. Example:

```
using namespace cvm;
```

```
try {  
    srbmatrix m1(7,1,0);  
    srbmatrix m2(7,1,1);  
    m1.set(1.);  
    m2.set(1.);  
  
    std::cout << m1 * m2;  
}  
catch (std::exception& e) {  
    std::cout << "Exception: " << e.what () << std::endl;  
}
```

prints

```
1 1 0 0 0 0 0  
2 2 1 0 0 0 0  
1 2 2 1 0 0 0  
0 1 2 2 1 0 0  
0 0 1 2 2 1 0  
0 0 0 1 2 2 1  
0 0 0 0 1 2 2
```

**2.11.39 low\_up**

Functions

```
srbmatrix&
srbmatrix::low_up (const srbmatrix& m, int* nPivots) throw (cvmexception);
srbmatrix
srbmatrix::low_up (int* nPivots) const throw (cvmexception);
```

compute LU factorization of calling band matrix as

$$A = PLU$$

where P is permutation matrix, L is lower triangular matrix with unit diagonal elements and U is upper triangular matrix. All functions store result as matrix L without main diagonal combined with U. All functions return pivot indices as array of integers (it should support at least `m.size()` elements) pointed to by `nPivots` so *i*-th row was interchanged with `nPivots[i]`-th row. The first version sets calling matrix to be equal to the *m*'s LU factorization and the second one creates object of type `srbmatrix` as calling band matrix's LU factorization. Functions throw `cvmexception` in case of inappropriate sizes of the operands or when matrix to be factorized is close to singular. The first version also changes number of super-diagonals to be equal to  $k_l + k_u$  in order to keep the result of factorization. It is recommended to use `iarray` for pivot values. This function is provided mostly for solving multiple systems of linear equations using `srbmatrix::solve_lu` function, See also `srbmatrix`. Example:

```
using namespace cvm;

std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (4);
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
    srbmatrix ma(a,4,1,0);
    srbmatrix mLU(4,1,0);
    rmatrix mb1(4,2); rvector vb1(4);
    rmatrix mb2(4,2); rvector vb2(4);
    rmatrix mx1(4,2); rvector vx1(4);
    rmatrix mx2(4,2); rvector vx2(4);
    iarray nPivots(4);
    double dErr = 0.;
    mb1.randomize(-1.,3.); vb1.randomize(-2.,4.);
    mb2.randomize(-2.,5.); vb2.randomize(-3.,1.);

    mLU.low_up(ma, nPivots);
    mx1 = ma.solve_lu (mLU, nPivots, mb1, dErr);
```

```

std::cout << mx1 << dErr << std::endl << std::endl;
mx2 = ma.solve_lu (mLU, nPivots, mb2);
std::cout << mx2 << std::endl;;
std::cout << ma * mx1 - mb1 << std::endl << ma * mx2 - mb2;

vx1 = ma.solve_lu (mLU, nPivots, vb1, dErr);
std::cout << vx1 << dErr << std::endl;
vx2 = ma.solve_lu (mLU, nPivots, vb2);
std::cout << vx2 << std::endl;;
std::cout << ma * vx1 - vb1 << std::endl << ma * vx2 - vb2;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
prints
7.6327e-02 -4.7386e-01
-2.9523e-01 9.7577e-01
1.7288e-01 -3.5093e-01
1.0595e-01 4.7363e-01
1.1832e-15

3.1963e+00 4.8622e+00
-4.9904e-01 -2.6575e+00
8.2183e-01 2.3294e+00
-6.1693e-01 -1.8015e+00

0.0000e+00 0.0000e+00
0.0000e+00 -2.2204e-16
0.0000e+00 -4.4409e-16
0.0000e+00 0.0000e+00

0.0000e+00 0.0000e+00
8.8818e-16 0.0000e+00
0.0000e+00 -4.4409e-16
-4.4409e-16 4.4409e-16
7.8933e-01 7.0543e-01 -1.6338e-02 -2.6206e-01
1.4832e-15
-1.5505e+00 5.8987e-01 -8.4977e-01 7.3059e-01

0.0000e+00 0.0000e+00 0.0000e+00 0.0000e+00

-2.2204e-16 0.0000e+00 0.0000e+00 4.4409e-16

```



### 2.11.40 operator / (const rvector&)

Operator

```
rvector operator / (const rvector& vB) const throw (cvmexception);
```

returns solution  $x$  of linear equation  $A * x = b$  where calling matrix is square band matrix  $A$  and vector  $b$  is passed in parameter  $vB$ . This operator throws exception of type `cvmexception` in case of inappropriate sizes of the objects or when matrix  $A$  is close to singular. See also `rvector::solve`, `srmatrix::solve`, `rvector.operator %`, `rvector`, `srmatrix`, `srbmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (5);

try {
    srbmatrix ma(4,2,1);
    rvector vb(4);
    rvector vx(4);
    ma.randomize(-1.,1.);
    vb.randomize(-2.,2.);

    vx = ma / vb;

    std::cout << ma * vx - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
+0.000000e+000 +2.22045e-016 +2.22045e-016 +0.000000e+000
```

### 2.11.41 identity

Function

```
srbmatrix& srbmatrix::identity();
```

sets calling band matrix to be equal to identity matrix and returns reference to the matrix changed. Function doesn't change numbers of sub- and super-diagonals. See also [srbmatrix](#). Example:

```
using namespace cvm;
```

```
srbmatrix m(4);  
m.randomize(0.,1.);  
std::cout << m << std::endl;  
std::cout << m.identity();
```

prints

```
0.327372 0 0 0  
0 0.955718 0 0  
0 0 0.0960723 0  
0 0 0 0.291818
```

```
1 0 0 0  
0 1 0 0  
0 0 1 0  
0 0 0 1
```

### 2.11.42 vanish

Function

```
srbmatrix& srbmatrix::vanish();
```

sets every element of calling band matrix to be equal to zero and returns reference to the matrix changed. This function is faster than `srbmatrix::set(TR)` with zero operand passed. See also `srbmatrix`. Example:

```
using namespace cvm;
```

```
srbmatrix m(4,1,0);  
m.randomize(0.,1.);  
std::cout << m << std::endl;  
std::cout << m.vanish();
```

prints

```
0.337138 0 0 0  
0.101199 0.522843 0 0  
0 0.258522 0.123447 0  
0 0 0.591723 0.661489
```

```
0 0 0 0  
0 0 0 0  
0 0 0 0  
0 0 0 0
```

### 2.11.43 `randomize`

Function

```
srbmatrix& srbmatrix::randomize (TR dFrom, TR dTo);
```

fills calling band matrix with pseudo-random numbers distributed between dFrom and dTo. Function returns reference to the matrix changed. See also [srbmatrix](#). Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::showpos);  
std::cout.precision (7);  
srbmatrix m(4,1,0);  
m.randomize(0.,1.);  
std::cout << m << std::endl;
```

prints

```
+3.4281442e-01 +0.0000000e+00 +0.0000000e+00 +0.0000000e+00  
+7.9808954e-01 +5.9761345e-01 +0.0000000e+00 +0.0000000e+00  
+0.0000000e+00 +1.1670278e-01 +6.5645314e-02 +0.0000000e+00  
+0.0000000e+00 +0.0000000e+00 +4.2225410e-01 +7.5563829e-02
```

## 2.12 scbmatrix

This is end-user class encapsulating square band matrix of complex numbers. This class utilizes [band storage](#) for its elements.

```
template <typename TR, typename TC>
class scbmatrix : public scmatrix <TR,TC>, public BandMatrix <TR,TC> {
public:
    scbmatrix ();
    explicit scbmatrix (int nMN);
    scbmatrix (int nMN);
    scbmatrix (TC* pD, int nMN, int nKL, int nKU);
    scbmatrix (const TC* pD, int nMN, int nKL, int nKU);
    scbmatrix (const scbmatrix& m);
    scbmatrix (const cmatrix& m, int nKL, int nKU);
    explicit scbmatrix (const cvector& v);
    explicit scbmatrix (const srbmatrix& m, bool bRealPart = true);
    scbmatrix (const srbmatrix& mRe, const srbmatrix& mIm);
    TC& operator () (int im, int in) throw (cvmexception);
    TC operator () (int im, int in) const throw (cvmexception);
    const cvector operator () (int i) const throw (cvmexception);
    const cvector operator [] (int i) const throw (cvmexception);
    const srbmatrix real () const;
    const srbmatrix imag () const;
    scbmatrix& operator = (const scbmatrix& m) throw (cvmexception);
    scbmatrix& assign (const cvector& v) throw (cvmexception);
    scbmatrix& assign (const TC* pD);
    scbmatrix& set (TC z);
    scbmatrix& assign_real (const srbmatrix& mRe) throw (cvmexception);
    scbmatrix& assign_imag (const srbmatrix& mIm) throw (cvmexception);
    scbmatrix& set_real (TR d);
    scbmatrix& set_imag (TR d);
    scbmatrix& resize (int nNewMN) throw (cvmexception);
    scbmatrix& resize_lu (int nNewKL, int nNewKU) throw (cvmexception);
    bool operator == (const scbmatrix& v) const;
    bool operator != (const scbmatrix& v) const;
    scbmatrix& operator << (const scbmatrix& m) throw (cvmexception);
    scbmatrix operator + (const scbmatrix& m) const
        throw (cvmexception);
    scbmatrix operator - (const scbmatrix& m) const
        throw (cvmexception);
    scbmatrix& sum (const scbmatrix& m1,
        const scbmatrix& m2) throw (cvmexception);
```

```

scbmatrix& diff (const scbmatrix& m1,
                const scbmatrix& m2) throw (cvmexception);
scbmatrix& operator += (const scbmatrix& m) throw (cvmexception);
scbmatrix& operator -= (const scbmatrix& m) throw (cvmexception);
scbmatrix operator - () const;
scbmatrix& operator ++ ();
scbmatrix& operator ++ (int);
scbmatrix& operator -- ();
scbmatrix& operator -- (int);
scbmatrix operator * (TR d) const;
scbmatrix operator / (TR d) const throw (cvmexception);
scbmatrix operator * (TC z) const;
scbmatrix operator / (TC z) const throw (cvmexception);
scbmatrix& operator *= (TR d);
scbmatrix& operator /= (TR d) throw (cvmexception);
scbmatrix& operator *= (TC z);
scbmatrix& operator /= (TC z) throw (cvmexception);
scbmatrix& normalize ();
scbmatrix operator ~ () const;
scbmatrix operator ! () const;
scbmatrix& conj (const scbmatrix& m) throw (cvmexception);
scbmatrix& conj ();
scbmatrix& transpose (const scbmatrix& m) throw (cvmexception);
scbmatrix& transpose ();
cvector operator * (const cvector& v) const throw (cvmexception);
cmatrix operator * (const cmatrix& m) const throw (cvmexception);
scmatrix operator * (const scmatrix& m) const throw (cvmexception);
scbmatrix operator * (const scbmatrix& m) const throw (cvmexception);
scbmatrix& low_up (const scbmatrix& m,
                 int* nPivots) throw (cvmexception);
scbmatrix low_up (int* nPivots) const throw (cvmexception);
cvector operator / (const cvector& vB) const throw (cvmexception);
scbmatrix& identity ();
scbmatrix& vanish ();
scbmatrix& randomize_real (TR dFrom, TR dTo);
scbmatrix& randomize_imag (TR dFrom, TR dTo);
};

```

### 2.12.1 scbmatrix ()

Constructor

```
scbmatrix::scbmatrix ();
```

creates empty scbmatrix object. See also [scbmatrix](#). Example:

```
using namespace cvm;
```

```
scbmatrix m;  
std::cout << m.msize() << " " << m.nsize() << " " << m.size() ;  
std::cout << " " << m.lsize() << " " << m.usize() << std::endl;  
m.resize(3);  
m.resize_lu(1,0);  
m.set(std::complex<double>(1.,2.));  
std::cout << m;
```

prints

```
0 0 0 0 0  
(1,2) (0,0) (0,0)  
(1,2) (1,2) (0,0)  
(0,0) (1,2) (1,2)
```

**2.12.2 scbmatrix (int)**

Constructor

```
explicit scbmatrix::scbmatrix (int nMN);
```

creates  $n \times n$  scbmatrix object where  $n$  is passed in `nMN` parameter. The matrix created is diagonal, i.e.  $k_l = k_u = 0$ . Constructor throws `cvmexception` in case of non-positive size passed or memory allocation failure. See also `scbmatrix`. Example:

```
using namespace cvm;
```

```
scbmatrix m(4);
std::cout << m.msize() << " " << m.nsize() << " " << m.size() ;
std::cout << " " << m.lsize() << " " << m.usize() << std::endl;
m.set(std::complex<double>(1.,2.));
std::cout << m;
```

prints

```
4 4 4 0 0
(1,2) (0,0) (0,0) (0,0)
(0,0) (1,2) (0,0) (0,0)
(0,0) (0,0) (1,2) (0,0)
(0,0) (0,0) (0,0) (1,2)
```



**2.12.3 scbmatrix (int,int,int)**

Constructor

```
scbmatrix::scbmatrix (int nMN, int nKL, int nKU);
```

creates  $n \times n$  scbmatrix object where  $n$  is passed in nMN parameter. The matrix created has nKL sub-diagonals and nKU super-diagonals. Constructor throws `cvmexception` in case of non-positive size or negative number of sub-diagonals or super-diagonals passed or in case of memory allocation failure. See also `scbmatrix`. Example:

```
using namespace cvm;
```

```
scbmatrix m(4,1,1);
m.set(std::complex<double>(1.,2.));
std::cout << m << std::endl
          << m.msize() << " " << m.nsize() << " " << m.size()
          << " " << m.lsize() << " " << m.usize() << std::endl;
```

prints

```
(1,2) (1,2) (0,0) (0,0)
(1,2) (1,2) (1,2) (0,0)
(0,0) (1,2) (1,2) (1,2)
(0,0) (0,0) (1,2) (1,2)
```

```
4 4 12 1 1
```

**2.12.4 scbmatrix (TC\*,int,int,int)**

Constructor

```
scbmatrix::scbmatrix (TC* pD, int nMN, int nKL, int nKU);
```

creates  $n \times n$  scbmatrix object where  $n$  is passed in  $nMN$  parameter. The matrix created has  $k_l=nKL$  sub-diagonals and  $k_u=nKU$  super-diagonals. Unlike others, this constructor *does not allocate memory*. It just shares memory with array pointed to by  $pD$ . Please note that this array must contain at least  $(k_l + k_u + 1) * n$  elements. Constructor throws `cvmexception` in case of non-positive size or negative number of sub- or super-diagonals passed. See also `scbmatrix`, `scbmatrix (const TC*,int,int,int)`. Example:

```
using namespace cvm;
double a[] = {1., 1., 1., 1., 1., 1., 1., 1.,
              1., 1., 1., 1., 1., 1., 1., 1.};
scbmatrix m1 ((std::complex<double>*)a,4,1,0);
scbmatrix mu ((std::complex<double>*)a,4,0,1);
m1(2,1) = std::complex<double>(5.,5.);
std::cout << m1 << std::endl << mu << std::endl;
std::cout << a[0] << " " << a[1] << " " << a[2] << " "
          << a[3] << " " << a[4] << " " << a[5] << " " << std::endl;
```

prints

```
(1,1) (0,0) (0,0) (0,0)
(5,5) (1,1) (0,0) (0,0)
(0,0) (1,1) (1,1) (0,0)
(0,0) (0,0) (1,1) (1,1)
```

```
(5,5) (1,1) (0,0) (0,0)
(0,0) (1,1) (1,1) (0,0)
(0,0) (0,0) (1,1) (1,1)
(0,0) (0,0) (0,0) (1,1)
```

```
1 1 5 5 1 1
```

**2.12.5 scbmatrix (const TC\*,int,int,int)**

Constructor

```
scbmatrix::scbmatrix (const TC* pD, int nMN, int nKL, int nKU);
```

creates  $n \times n$  scbmatrix object where  $n$  is passed in  $nMN$  parameter. The matrix created has  $k_l=nKL$  sub-diagonals and  $k_u=nKU$  super-diagonals. Then constructor copies  $(k_l + k_u + 1) * n$  elements of array  $pD$  to the matrix according to **band storage**. Constructor throws **cvmexception** in case of non-positive size or negative number of sub- or super-diagonals passed. See also **scbmatrix**, **scbmatrix (TC\*,int,int,int)**. Example:

```
using namespace cvm;
const double a[] = {1., 1., 1., 1., 1., 1., 1., 1.,
                   1., 1., 1., 1., 1., 1., 1., 1.};
scbmatrix m1 ((const std::complex<double>*)a,4,1,0);
scbmatrix mu ((const std::complex<double>*)a,4,0,1);
m1(2,1) = std::complex<double>(5.,5.);
std::cout << m1 << std::endl << mu << std::endl;
std::cout << a[0] << " " << a[1] << " " << a[2] << " "
          << a[3] << " " << a[4] << " " << a[5] << " " << std::endl;
```

prints

```
(1,1) (0,0) (0,0) (0,0)
(5,5) (1,1) (0,0) (0,0)
(0,0) (1,1) (1,1) (0,0)
(0,0) (0,0) (1,1) (1,1)
```

```
(1,1) (1,1) (0,0) (0,0)
(0,0) (1,1) (1,1) (0,0)
(0,0) (0,0) (1,1) (1,1)
(0,0) (0,0) (0,0) (1,1)
```

```
1 1 1 1 1 1
```

**2.12.6 scbmatrix (const scbmatrix&)**

Copy constructor

```
scbmatrix::scbmatrix (const scbmatrix& m);
```

creates scbmatrix object as a copy of m. Constructor throws `cvmexception` in case of memory allocation failure. See also `scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
              10., 11., 12., 13., 14., 15., 16.};
scbmatrix m ((std::complex<double>*)a,4,1,0);
scbmatrix mc(m);
m(1,1) = 7.77;
std::cout << m << std::endl << mc;
```

prints

```
(7.77,0) (0,0) (0,0) (0,0)
(3,4) (5,6) (0,0) (0,0)
(0,0) (7,8) (9,10) (0,0)
(0,0) (0,0) (11,12) (13,14)

(1,2) (0,0) (0,0) (0,0)
(3,4) (5,6) (0,0) (0,0)
(0,0) (7,8) (9,10) (0,0)
(0,0) (0,0) (11,12) (13,14)
```

**2.12.7 scbmatrix (const cmatrix&,int,int)**

Constructor

```
scbmatrix::scbmatrix (const cmatrix& m, int nKL, int nKU);
```

creates scbmatrix object as a copy of “sliced” matrix m, i.e. it copies main diagonal, nKL sub-diagonals and nKU super-diagonals of matrix m. It’s assumed that  $m \times n$  matrix m must have equal sizes, i.e.  $m = n$  is satisfied. Constructor throws **cvmexception** if this is not true or in case of memory allocation failure. See also **scbmatrix**. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
              10., 11., 12., 13., 14., 15., 16., 17., 18.};
scmatrix m((std::complex<double>*)a,3);
scbmatrix mb(m,1,0);
std::cout << m << std::endl << mb;
```

prints

```
(1,2) (7,8) (13,14)
(3,4) (9,10) (15,16)
(5,6) (11,12) (17,18)

(1,2) (0,0) (0,0)
(3,4) (9,10) (0,0)
(0,0) (11,12) (17,18)
```

### 2.12.8 scbmatrix (const cvector&)

Constructor

```
explicit scbmatrix::scbmatrix (const cvector& v);
```

creates scbmatrix object of size `v.size()` by `v.size()` and assigns vector `v` to its main diagonal. Constructor throws `cvmexception` in case of memory allocation failure. See also `scbmatrix`, `cvector`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6.};  
cvector v((std::complex<double>*)a,3);  
scbmatrix m(v);  
std::cout << m;
```

prints

```
(1,2) (0,0) (0,0)  
(0,0) (3,4) (0,0)  
(0,0) (0,0) (5,6)
```

**2.12.9 scbmatrix (const srbmatrix&,bool)**

Constructor

```
explicit scbmatrix::scbmatrix (const srbmatrix& m, bool bRealPart = true);
```

creates scbmatrix object having the same dimension and the same numbers of sub- and super-diagonals as real matrix `m` and copies the matrix `m` to its real part if `bRealPart` is true or to its imaginary part otherwise. See also [scbmatrix](#), [srbmatrix](#). Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
const srbmatrix m(a,4,1,0);
scbmatrix mr(m), mi(m, false);
std::cout << mr << std::endl << mi;
```

prints

```
(1,0) (0,0) (0,0) (0,0)
(2,0) (3,0) (0,0) (0,0)
(0,0) (4,0) (5,0) (0,0)
(0,0) (0,0) (6,0) (7,0)
```

```
(0,1) (0,0) (0,0) (0,0)
(0,2) (0,3) (0,0) (0,0)
(0,0) (0,4) (0,5) (0,0)
(0,0) (0,0) (0,6) (0,7)
```

**2.12.10 scbmatrix (const srbmatrix&, const srbmatrix&)**

Constructor

```
scbmatrix::scbmatrix (const srbmatrix& mRe, const srbmatrix& mIm);
```

creates scbmatrix object of the same size as mRe and mIm has (it throws `cvmexception` if mRe and mIm have different sizes or different numbers of sub- or super-diagonals) and copies matrices mRe and mIm to real and imaginary part of the matrix created respectively. Constructor throws `cvmexception` in case of memory allocation failure. See also `scbmatrix`, `srbmatrix`. Example:

```
using namespace cvm;
```

```
srbmatrix mr(4,1,0), mi(4,1,0);  
mr.set(1.);  
mi.set(2.);  
const scbmatrix m(mr,mi);  
std::cout << m;
```

prints

```
(1,2) (0,0) (0,0) (0,0)  
(1,2) (1,2) (0,0) (0,0)  
(0,0) (1,2) (1,2) (0,0)  
(0,0) (0,0) (1,2) (1,2)
```



**2.12.11 operator (,)**

Indexing operators

```
TC& scbmatrix::operator () (int im, int in) throw (cvmexception);
TC scbmatrix::operator () (int im, int in) const throw (cvmexception);
```

provide access to a particular element of calling band matrix. The first version of operator is applicable to non-constant object. This version returns *l-value* in order to make possible write access to an element. Only elements located on main diagonal or on non-zero sub- or super-diagonals are l-values. All other values located outside this area are not writable. Both operators are *l-based*. Operators throw `cvmexception` if some of parameters passed is outside of `[1,msize()]` range or in case of attempt to write to non-writable element<sup>10</sup>. See also `scbmatrix`, `BandMatrix::lsize()` and `BandMatrix::usize()`. Example:

```
using namespace cvm;

std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11., 12.};
    srbmatrix m (a,3,1,0);

    m(2,1) = 7.77;
    std::cout << m << std::endl;
    std::cout << m(3,2) << " " << m(1,3) << std::endl;

    m(1,3) = 7.77;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}

prints

1.00e+00 0.00e+00 0.00e+00
7.77e+00 3.00e+00 0.00e+00
0.00e+00 4.00e+00 5.00e+00

4.00e+00 0.00e+00
Exception: Attempt to change a read-only element
```

<sup>10</sup>Here I use `type_proxy<T>` class originally described in [4], p. 217.

**2.12.12 operator ()**

Indexing operator

```
const cvector scbmatrix::operator () (int i) const throw (cvmexception);
```

provides access to *i*-th column of calling band matrix. Unlike `scmatrix::operator ()`, this operator creates only *copy* of a column and therefore it returns *not l-value*. Operator is *l-based*. It throws `cvmexception` if parameter *i* is outside of `[1, nsize()]` range. See also `scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11., 12.};
scbmatrix m ((std::complex<double>*)a,3,1,0);
std::cout << m << std::endl;
std::cout << m(2);
```

prints

```
(1,2) (0,0) (0,0)
(3,4) (5,6) (0,0)
(0,0) (7,8) (9,10)

(0,0) (5,6) (7,8)
```

**2.12.13 operator []**

Indexing operator

```
const cvector scbmatrix::operator [] (int i) const throw (cvmexception);
```

provides access to *i*-th row of calling band matrix. Unlike `scmatrix::operator []`, this operator creates only *copy* of a column and therefore it returns *not l-value*. Operator is *l-based*. It throws `cvmexception` if parameter *i* is outside of `[1, nsize()]` range. See also `scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11., 12.};
scbmatrix m ((std::complex<double>*)a,3,1,0);
std::cout << m << std::endl;
std::cout << m[3];
```

prints

```
(1,2) (0,0) (0,0)
(3,4) (5,6) (0,0)
(0,0) (7,8) (9,10)

(0,0) (7,8) (9,10)
```

### 2.12.14 real

Function

```
const srbmatrix scbmatrix::real () const;
```

creates object of type `const srbmatrix` as real part of calling band matrix. Please note that, unlike `vector::real`, this function creates new object *not sharing* memory with real part of calling matrix, i.e. the matrix returned is *not l-value*. See also `srbmatrix`, `scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11., 12.};  
scbmatrix m ((std::complex<double>*)a,3,1,0);  
std::cout << m << std::endl;  
std::cout << m.real();
```

prints

```
(1,2) (0,0) (0,0)  
(3,4) (5,6) (0,0)  
(0,0) (7,8) (9,10)
```

```
1 0 0  
3 5 0  
0 7 9
```

### 2.12.15 imag

Function

```
const srbmatrix scbmatrix::imag () const;
```

creates object of type `const srbmatrix` as imaginary part of calling band matrix. Please note that, unlike `cvvector::imag`, this function creates new object *not sharing* memory with imaginary part of calling matrix, i.e. the matrix returned is *not l-value*. See also `srbmatrix`, `scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11., 12.};  
scbmatrix m ((std::complex<double>*)a,3,1,0);  
std::cout << m << std::endl;  
std::cout << m.imag();
```

prints

```
(1,2) (0,0) (0,0)  
(3,4) (5,6) (0,0)  
(0,0) (7,8) (9,10)
```

```
2 0 0  
4 6 0  
0 8 10
```

**2.12.16 operator = (const scbmatrix&)**

Operator

```
scbmatrix& scbmatrix::operator = (const scbmatrix& m)
throw (cvmexception);
```

sets every element of calling band matrix to value of appropriate element of band matrix *m* and returns reference to the matrix changed. Operator throws `cvmexception` in case of different matrix sizes or in case of different numbers of sub- or super-diagonals. See also `scbmatrix`. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11., 12.};
    scbmatrix m1((std::complex<double>*)a,3,1,0);
    scbmatrix m2(3,1,0);

    m2 = m1;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
(1.00e+00,2.00e+00) (0.00e+00,0.00e+00) (0.00e+00,0.00e+00)
(3.00e+00,4.00e+00) (5.00e+00,6.00e+00) (0.00e+00,0.00e+00)
(0.00e+00,0.00e+00) (7.00e+00,8.00e+00) (9.00e+00,1.00e+01)
```

**2.12.17 assign (const TC\*)**

Function

```
scbmatrix& scbmatrix::assign (const cvector& v) throw (cvmexception);  
scbmatrix& scbmatrix::assign (const TC* pD);
```

sets every element of calling band matrix to value of appropriate element of vector *v* or array pointed to by *pD* and returns reference to the matrix changed. In first version it's assumed that vector passed is long enough to fill calling matrix. Function throws **cvmexception** otherwise. In other words this array must contain at least  $(k_l + k_u + 1)n$  elements. See also **scbmatrix**. Example:

```
using namespace cvm;
```

```
const double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11., 12.};  
scbmatrix m(3,0,1);  
m.assign((const std::complex<double>*)a);  
std::cout << m;
```

prints

```
(3,4) (5,6) (0,0)  
(0,0) (7,8) (9,10)  
(0,0) (0,0) (11,12)
```

**2.12.18 set (TC)**

Function

```
scbmatrix& scbmatrix::set (TC z);
```

sets every element of calling band matrix to value of parameter *z* and returns reference to the matrix changed. Use **vanish** to set every element of calling matrix to be equal to zero. See also **scbmatrix**. Example:

```
using namespace cvm;
```

```
scbmatrix m(4,1,0);  
m.set(std::complex<double>(1.,2.));  
std::cout << m;
```

prints

```
(1,2) (0,0) (0,0) (0,0)  
(1,2) (1,2) (0,0) (0,0)  
(0,0) (1,2) (1,2) (0,0)  
(0,0) (0,0) (1,2) (1,2)
```



### 2.12.19 assign\_real

Function

```
scbmatrix& scbmatrix::assign_real (const srbmatrix& mRe)  
throw (cvmexception);
```

sets real part of every element of calling band matrix to value of appropriate element of band matrix mRe and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. See also `scbmatrix` and `srbmatrix`. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);  
srbmatrix m (3,0,1);  
scbmatrix mc(3,0,1);  
m.randomize (0., 1.);
```

```
mc.assign_real(m);  
std::cout << mc;
```

prints

```
(5.44e-01,0.00e+00) (5.48e-02,0.00e+00) (0.00e+00,0.00e+00)  
(0.00e+00,0.00e+00) (3.66e-01,0.00e+00) (3.49e-01,0.00e+00)  
(0.00e+00,0.00e+00) (0.00e+00,0.00e+00) (8.00e-01,0.00e+00)
```

### 2.12.20 assign\_imag

Function

```
scbmatrix& scbmatrix::assign_imag (const srbmatrix& mIm)  
throw (cvmexception);
```

sets imaginary part of every element of calling band matrix to value of appropriate element of bandmatrix mIm and returns reference to the matrix changed. Function throws `cvmexception` in case of different sizes of the operands. See also `scbmatrix` and `srbmatrix`. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);  
srbmatrix m (3,0,1);  
scbmatrix mc(3,0,1);  
m.randomize (0., 1.);
```

```
mc.assign_imag(m);  
std::cout << mc;
```

prints

```
(0.00e+00,5.53e-01) (0.00e+00,2.16e-01) (0.00e+00,0.00e+00)  
(0.00e+00,0.00e+00) (0.00e+00,1.57e-01) (0.00e+00,1.12e-01)  
(0.00e+00,0.00e+00) (0.00e+00,0.00e+00) (0.00e+00,7.03e-01)
```

### 2.12.21 set\_real

Function

```
scbmatrix& scbmatrix::set_real (TR d);
```

sets real part of every element of calling band matrix to value of parameter d and returns reference to the matrix changed. See also [scbmatrix](#). Example:

```
using namespace cvm;
```

```
scbmatrix m(4,0,1);  
m.set_real(1.);  
std::cout << m;
```

prints

```
(1,0) (1,0) (0,0) (0,0)  
(0,0) (1,0) (1,0) (0,0)  
(0,0) (0,0) (1,0) (1,0)  
(0,0) (0,0) (0,0) (1,0)
```

**2.12.22 set\_imag**

Function

```
scbmatrix& scbmatrix::set_imag (TR d);
```

sets imaginary part of every element of calling band matrix to value of parameter *d* and returns reference to the matrix changed. See also [scbmatrix](#). Example:

```
using namespace cvm;
```

```
scbmatrix m(4,0,1);  
m.set_imag(1.);  
std::cout << m;
```

prints

```
(0,1) (0,1) (0,0) (0,0)  
(0,0) (0,1) (0,1) (0,0)  
(0,0) (0,0) (0,1) (0,1)  
(0,0) (0,0) (0,0) (0,1)
```

**2.12.23 resize**

Function

```
scbmatrix& scbmatrix::resize (int nNewMN) throw (cvmexception);
```

changes size of calling band matrix to nNewMN by nNewMN and returns reference to the matrix changed. In case of increasing of its size, calling matrix is filled up with zeroes. This function doesn't change number of sub- or super-diagonals. Like any band matrix class member function, this function doesn't change **non-referred elements**. See number (11,12) appearing after resize in example below. Function throws **cvmexception** in case of non-positive size passed or memory allocation failure. See also **scbmatrix.resize\_lu**, **scbmatrix**. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11., 12.};
    scbmatrix m((std::complex<double>*)a,3,1,0);
    std::cout << m << std::endl;
    m.resize (4);
    std::cout << m;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
(1,2) (0,0) (0,0)
(3,4) (5,6) (0,0)
(0,0) (7,8) (9,10)
```

```
(1,2) (0,0) (0,0) (0,0)
(3,4) (5,6) (0,0) (0,0)
(0,0) (7,8) (9,10) (0,0)
(0,0) (0,0) (11,12) (0,0)
```

**2.12.24 resize\_lu**

Function

```
scbmatrix& scbmatrix::resize_lu (int nNewKL, int nNewKU)
throw (cvmexception);
```

changes number of sub- and super-diagonals of calling band matrix to nNewKL by nNewKU respectively and returns reference to the matrix changed. In case of increasing of the numbers, calling matrix is filled up with zeroes. Function throws `cvmexception` in case of negative number passed or memory allocation failure. See also `scbmatrix::resize`, `scbmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9., 10., 11., 12.};
    scbmatrix m((std::complex<double>*)a,3,1,0);
    std::cout << m << std::endl;
    m.resize_lu (0,1);
    m.diag(1).set(std::complex<double>(9.,9.));
    std::cout << m;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
(1,2) (0,0) (0,0)
(3,4) (5,6) (0,0)
(0,0) (7,8) (9,10)

(1,2) (9,9) (0,0)
(0,0) (5,6) (9,9)
(0,0) (0,0) (9,10)
```

### 2.12.25 operator ==

Operator

```
bool scbmatrix::operator == (const scbmatrix& m) const;
```

compares calling band matrix with band matrix `m` and returns true if they have the same sizes, the same numbers of sub- and super-diagonals and their appropriate elements differ by not more than the **smallest normalized positive number**. Returns false otherwise. See also `scbmatrix`. Example:

```
using namespace cvm;

double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};
scbmatrix m1((std::complex<double>*)a,2,1,0);
scbmatrix m2(2,1,0);
std::cout << m1 << std::endl;

m2(1,1) = std::complex<double>(1.,2.);
m2(2,1) = std::complex<double>(3.,4.);
m2(2,2) = std::complex<double>(5.,6.);

std::cout << (m1 == m2) << std::endl;

prints

(1,2) (0,0)
(3,4) (5,6)
```

1

### 2.12.26 operator !=

Operator

```
bool scbmatrix::operator != (const scbmatrix& m) const;
```

compares calling band matrix with band matrix `m` and returns true if they have different sizes, different numbers of sub- or super-diagonals or at least one of their appropriate elements differs by more than the **smallest normalized positive number**. Returns false otherwise. See also `scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.};  
scbmatrix m1((std::complex<double>*)a,2,1,0);  
scbmatrix m2(2,1,0);  
std::cout << m1 << std::endl;
```

```
m2(1,1) = std::complex<double>(1.,2.);  
m2(2,1) = std::complex<double>(3.,4.);  
m2(2,2) = std::complex<double>(5.,6.00001);
```

```
std::cout << (m1 != m2) << std::endl;
```

prints

```
(1,2) (0,0)  
(3,4) (5,6)
```

1



**2.12.27 operator <<**

Operator

```
scbmatrix& scbmatrix::operator << (const scbmatrix& m)
throw (cvmexception);
```

destroys calling band matrix, creates new one as a copy of m and returns reference to the matrix changed. Operator throws **cvmexception** in case of memory allocation failure. See also **scbmatrix**. Example:

```
using namespace cvm;

try {
    scbmatrix m(3,1,0);
    scbmatrix mc(1);
    m(2,1) = std::complex<double>(1.,2.);
    m(2,2) = std::complex<double>(3.,4.);
    std::cout << m << std::endl << mc << std::endl;

    mc << m;
    std::cout << mc;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
}
```

prints

```
(0,0) (0,0) (0,0)
(1,2) (3,4) (0,0)
(0,0) (0,0) (0,0)
```

```
(0,0)
```

```
(0,0) (0,0) (0,0)
(1,2) (3,4) (0,0)
(0,0) (0,0) (0,0)
```

**2.12.28 operator +**

Operator

```
scbmatrix scbmatrix::operator + (const scbmatrix& m) const
throw (cvmexception);
```

creates object of type `scbmatrix` as sum of calling band matrix and band matrix `m`. It throws `cvmexception` in case of different sizes or different numbers of sub- or super-diagonals of the operands. See also `scbmatrix::sum`, `scbmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8.,
                  9., 10., 11., 12.};
    double b[] = {10., 20., 30., 40., 50., 60.,
                  70., 80., 90., 100., 110., 120.};
    scbmatrix m1((std::complex<double>*)a,3,0,1);
    scbmatrix m2((std::complex<double>*)b,3,0,1);

    std::cout << m1 << std::endl << m2 << std::endl;
    std::cout << m1 + m2 << std::endl << m1 + m1;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
(3,4) (5,6) (0,0)
(0,0) (7,8) (9,10)
(0,0) (0,0) (11,12)

(30,40) (50,60) (0,0)
(0,0) (70,80) (90,100)
(0,0) (0,0) (110,120)

(33,44) (55,66) (0,0)
(0,0) (77,88) (99,110)
(0,0) (0,0) (121,132)

(6,8) (10,12) (0,0)
(0,0) (14,16) (18,20)
(0,0) (0,0) (22,24)
```

**2.12.29 operator -**

Operator

```
scbmatrix scbmatrix::operator - (const scbmatrix& m) const
throw (cvmexception);
```

creates object of type `scbmatrix` as difference of calling band matrix and band matrix `m`. It throws `cvmexception` in case of different sizes or different numbers of sub- or super-diagonals of the operands. See also `scbmatrix::diff`, `scbmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8.,
                  9., 10., 11., 12.};
    double b[] = {10., 20., 30., 40., 50., 60.,
                  70., 80., 90., 100., 110., 120.};
    scbmatrix m1((std::complex<double>*)a,3,0,1);
    scbmatrix m2((std::complex<double>*)b,3,0,1);

    std::cout << m1 << std::endl << m2 << std::endl;
    std::cout << m1 - m2 << std::endl << m1 - m1;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
(3,4) (5,6) (0,0)
(0,0) (7,8) (9,10)
(0,0) (0,0) (11,12)

(30,40) (50,60) (0,0)
(0,0) (70,80) (90,100)
(0,0) (0,0) (110,120)

(-27,-36) (-45,-54) (0,0)
(0,0) (-63,-72) (-81,-90)
(0,0) (0,0) (-99,-108)

(0,0) (0,0) (0,0)
(0,0) (0,0) (0,0)
(0,0) (0,0) (0,0)
```

**2.12.30 sum**

Function

```
scbmatrix& scbmatrix::sum (const scbmatrix& m1, const scbmatrix& m2)
throw (cvmexception);
```

assigns result of addition of band matrices `m1` and `m2` to calling band matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes or different numbers of sub- or super-diagonals of the operands. See also `scbmatrix::operator +`, `scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.,
              9., 10., 11., 12.};
const scbmatrix m1((std::complex<double>*)a,3,1,0);
scbmatrix m2(3,1,0);
scbmatrix m(3,1,0);
m2.set(std::complex<double>(1.,1.));
std::cout << m1 << std::endl << m2 << std::endl;
std::cout << m.sum(m1, m2) << std::endl;
std::cout << m.sum(m, m2);
```

prints

```
(1,2) (0,0) (0,0)
(3,4) (5,6) (0,0)
(0,0) (7,8) (9,10)

(1,1) (0,0) (0,0)
(1,1) (1,1) (0,0)
(0,0) (1,1) (1,1)

(2,3) (0,0) (0,0)
(4,5) (6,7) (0,0)
(0,0) (8,9) (10,11)

(3,4) (0,0) (0,0)
(5,6) (7,8) (0,0)
(0,0) (9,10) (11,12)
```

**2.12.31 diff**

Function

```
scbmatrix& scbmatrix::diff (const scbmatrix& m1, const scbmatrix& m2)
throw (cvmexception);
```

assigns result of subtraction of band matrices `m1` and `m2` to calling band matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes or different numbers of sub- or super-diagonals of the operands. See also `scbmatrix::operator -`, `scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.,
              9., 10., 11., 12.};
const scbmatrix m1((std::complex<double>*)a,3,1,0);
scbmatrix m2(3,1,0);
scbmatrix m(3,1,0);
m2.set(std::complex<double>(1.,1.));
std::cout << m1 << std::endl << m2 << std::endl;
std::cout << m.diff(m1, m2) << std::endl;
std::cout << m.diff(m, m2);
```

prints

```
(1,2) (0,0) (0,0)
(3,4) (5,6) (0,0)
(0,0) (7,8) (9,10)
```

```
(1,1) (0,0) (0,0)
(1,1) (1,1) (0,0)
(0,0) (1,1) (1,1)
```

```
(0,1) (0,0) (0,0)
(2,3) (4,5) (0,0)
(0,0) (6,7) (8,9)
```

```
(-1,0) (0,0) (0,0)
(1,2) (3,4) (0,0)
(0,0) (5,6) (7,8)
```

**2.12.32 operator +=**

Operator

```
scbmatrix& scbmatrix::operator += (const scbmatrix& m)
throw (cvmexception);
```

adds band matrix *m* to calling band matrix and returns reference to the matrix changed. Operator throws `cvmexception` in case of different sizes or different numbers of sub- or super-diagonals of the operands. See also `scbmatrix::operator +` , `scbmatrix::sum`, `scbmatrix`. Example:

```
using namespace cvm;
```

```
try {
    scbmatrix m1(4,0,1);
    scbmatrix m2(4,0,1);
    m1.set(std::complex<double>(1.,2.));
    m2.set(std::complex<double>(3.,4.));

    m1 += m2;
    std::cout << m1 << std::endl;

    // well, you can do this too, but temporary object would be created
    m2 += m2;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
(4,6) (4,6) (0,0) (0,0)
(0,0) (4,6) (4,6) (0,0)
(0,0) (0,0) (4,6) (4,6)
(0,0) (0,0) (0,0) (4,6)

(6,8) (6,8) (0,0) (0,0)
(0,0) (6,8) (6,8) (0,0)
(0,0) (0,0) (6,8) (6,8)
(0,0) (0,0) (0,0) (6,8)
```

**2.12.33 operator -=**

Operator

```
scbmatrix& scbmatrix::operator -= (const scbmatrix& m)
throw (cvmexception);
```

subtracts band matrix `m` from calling band matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes or different numbers of sub- or super-diagonals of the operands. See also `scbmatrix::operator -`, `scbmatrix::diff`, `scbmatrix`. Example:

```
using namespace cvm;
```

```
try {
    scbmatrix m1(4,0,1);
    scbmatrix m2(4,0,1);
    m1.set(std::complex<double>(1.,2.));
    m2.set(std::complex<double>(3.,4.));

    m1 -= m2;
    std::cout << m1 << std::endl;

    // well, you can do this too, but temporary object would be created
    m2 -= m2;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
(-2,-2) (-2,-2) (0,0) (0,0)
(0,0) (-2,-2) (-2,-2) (0,0)
(0,0) (0,0) (-2,-2) (-2,-2)
(0,0) (0,0) (0,0) (-2,-2)

(0,0) (0,0) (0,0) (0,0)
(0,0) (0,0) (0,0) (0,0)
(0,0) (0,0) (0,0) (0,0)
(0,0) (0,0) (0,0) (0,0)
```

### 2.12.34 operator - ()

Operator

```
scbmatrix scbmatrix::operator - () const throw (cvmexception);
```

creates object of type `scbmatrix` as calling band matrix multiplied by  $-1$ . See also [scbmatrix](#). Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific |  
                std::ios::left |  
                std::ios::showpos);
```

```
std::cout.precision (2);
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8.,  
              9., 10., 11., 12.};
```

```
scbmatrix m((std::complex<double>*)a,3,1,0);
```

```
std::cout << -m;
```

prints

```
(-1.00e+000,-2.00e+000) (+0.00e+000,+0.00e+000) (+0.00e+000,+0.00e+000)  
(-3.00e+000,-4.00e+000) (-5.00e+000,-6.00e+000) (+0.00e+000,+0.00e+000)  
(+0.00e+000,+0.00e+000) (-7.00e+000,-8.00e+000) (-9.00e+000,-1.00e+001)
```



### 2.12.35 operator ++

Operator

```
scbmatrix& scbmatrix::operator ++ ();  
scbmatrix& scbmatrix::operator ++ (int);
```

adds identity matrix to calling band matrix and returns reference to the matrix changed.  
See also [scbmatrix](#). Example:

```
using namespace cvm;
```

```
scbmatrix m(4,1,0);  
m.set(std::complex<double>(1.,1.));
```

```
m++;  
std::cout << m << std::endl;  
std::cout << ++m;
```

prints

```
(2,1) (0,0) (0,0) (0,0)  
(1,1) (2,1) (0,0) (0,0)  
(0,0) (1,1) (2,1) (0,0)  
(0,0) (0,0) (1,1) (2,1)
```

```
(3,1) (0,0) (0,0) (0,0)  
(1,1) (3,1) (0,0) (0,0)  
(0,0) (1,1) (3,1) (0,0)  
(0,0) (0,0) (1,1) (3,1)
```

**2.12.36 operator --**

Operator

```
scbmatrix& scbmatrix::operator -- ();
scbmatrix& scbmatrix::operator -- (int);
```

subtracts identity matrix from calling band matrix and returns reference to the matrix changed. See also [scbmatrix](#). Example:

```
using namespace cvm;
```

```
scbmatrix m(4,1,0);
m.set(std::complex<double>(1.,1.));
```

```
m--;
std::cout << m << std::endl;
std::cout << --m;
```

prints

```
(0,1) (0,0) (0,0) (0,0)
(1,1) (0,1) (0,0) (0,0)
(0,0) (1,1) (0,1) (0,0)
(0,0) (0,0) (1,1) (0,1)
```

```
(-1,1) (0,0) (0,0) (0,0)
(1,1) (-1,1) (0,0) (0,0)
(0,0) (1,1) (-1,1) (0,0)
(0,0) (0,0) (1,1) (-1,1)
```

### 2.12.37 operator \* (TR)

Operator

```
scbmatrix scbmatrix::operator * (TR d) const;
```

creates object of type `scbmatrix` as product of calling band matrix and real number `d`. See also `scbmatrix::operator *= , scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,  
             10., 11., 12.};  
scbmatrix m((std::complex<double>*)a,3,0,1);  
std::cout << m * 5.;
```

prints

```
(15,20) (25,30) (0,0)  
(0,0) (35,40) (45,50)  
(0,0) (0,0) (55,60)
```

### 2.12.38 operator / (TR)

Operator

```
scbmatrix scbmatrix::operator / (TR d) const throw (cvmexception);
```

creates object of type `scbmatrix` as quotient of calling band matrix and real number `d`. It throws `cvmexception` if `d` has absolute value equal or less than the **smallest normalized positive number**. See also `scbmatrix::operator /=` , `scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,  
             10., 11., 12.};  
scbmatrix m((std::complex<double>*)a,3,0,1);  
std::cout << m / 2.;
```

prints

```
(1.5,2) (2.5,3) (0,0)  
(0,0) (3.5,4) (4.5,5)  
(0,0) (0,0) (5.5,6)
```

### 2.12.39 operator \* (TC)

Operator

```
scbmatrix scbmatrix::operator * (TC z) const;
```

creates object of type `scbmatrix` as product of calling band matrix and complex number `z`.

See also `scbmatrix::operator *=`, `scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,  
             10., 11., 12.};
```

```
scbmatrix m((std::complex<double>*)a,3,0,1);
```

```
std::cout << m * std::complex<double>(1.,1.);
```

prints

```
(-1,7) (-1,11) (0,0)  
(0,0) (-1,15) (-1,19)  
(0,0) (0,0) (-1,23)
```

### 2.12.40 operator / (TC)

Operator

```
scbmatrix scbmatrix::operator / (TC z) const throw (cvmexception);
```

creates object of type `scbmatrix` as quotient of calling band matrix and complex number `z`. It throws `cvmexception` if `z` has absolute value equal or less than the **smallest normalized positive number**. See also `scbmatrix::operator /=` , `scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
              10., 11., 12.};
scbmatrix m((std::complex<double>*)a,3,0,1);
std::cout << m / std::complex<double>(1.,1.);
```

prints

```
(3.5,0.5) (5.5,0.5) (0,0)
(0,0) (7.5,0.5) (9.5,0.5)
(0,0) (0,0) (11.5,0.5)
```

**2.12.41 operator \*= (TR)**

Operator

```
scbmatrix& scbmatrix::operator *= (TR d);
```

multiplies calling band matrix by real number d and returns reference to the matrix changed. See also `scbmatrix::operator *`, `scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,  
             10., 11., 12.};
```

```
scbmatrix m((std::complex<double>*)a,3,0,1);
```

```
m *= 5.;
```

```
std::cout << m;
```

prints

```
(15,20) (25,30) (0,0)
```

```
(0,0) (35,40) (45,50)
```

```
(0,0) (0,0) (55,60)
```

### 2.12.42 operator /= (TR)

Operator

```
scbmatrix& scbmatrix::operator /= (TR d) throw (cvmexception);
```

divides calling band matrix by real number d and returns reference to the matrix changed. It throws `cvmexception` if d has absolute value equal or less than the **smallest normalized positive number**. See also `scbmatrix::operator /`, `scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,  
              10., 11., 12.};  
scbmatrix m((std::complex<double>*)a,3,0,1);  
m /= 2.;  
std::cout << m;
```

prints

```
(1.5,2) (2.5,3) (0,0)  
(0,0) (3.5,4) (4.5,5)  
(0,0) (0,0) (5.5,6)
```



### 2.12.43 operator \*= (TC)

Operator

```
scbmatrix& scbmatrix::operator *= (TC z);
```

multiplies calling band matrix by complex number z and returns reference to the matrix changed. See also `scbmatrix::operator *`, `scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,  
             10., 11., 12.};  
scbmatrix m((std::complex<double>*)a,3,0,1);  
m *= std::complex<double>(1.,1.);  
std::cout << m;
```

prints

```
(-1,7) (-1,11) (0,0)  
(0,0) (-1,15) (-1,19)  
(0,0) (0,0) (-1,23)
```

### 2.12.44 operator /= (TC)

Operator

```
scbmatrix& scbmatrix::operator /= (TC z) throw (cvmexception);
```

divides calling band matrix by complex number z and returns reference to the matrix changed. It throws `cvmexception` if z has absolute value equal or less than the **smallest normalized positive number**. See also `scbmatrix::operator /` , `scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,  
             10., 11., 12.};  
scbmatrix m((std::complex<double>*)a,3,0,1);  
m /= std::complex<double>(1.,1.);  
std::cout << m;
```

prints

```
(3.5,0.5) (5.5,0.5) (0,0)  
(0,0) (7.5,0.5) (9.5,0.5)  
(0,0) (0,0) (11.5,0.5)
```

### 2.12.45 normalize

Function

```
scbmatrix& scbmatrix::normalize ();
```

normalizes calling band matrix so its **Euclidean norm** becomes equal to 1 if it was greater than the **smallest normalized positive number** before the call (otherwise function does nothing). See also **scbmatrix**. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
```

```
std::cout.precision (2);
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,  
             10., 11., 12.};
```

```
scbmatrix m((std::complex<double>*)a,3,0,1);
```

```
m.normalize();
```

```
std::cout << m << m.norm() << std::endl;
```

prints

```
(1.18e-001,1.57e-001) (1.97e-001,2.36e-001) (0.00e+000,0.00e+000)  
(0.00e+000,0.00e+000) (2.76e-001,3.15e-001) (3.54e-001,3.94e-001)  
(0.00e+000,0.00e+000) (0.00e+000,0.00e+000) (4.33e-001,4.72e-001)  
1.00e+000
```

**2.12.46 conjugation**

Operator and functions

```
scbmatrix scbmatrix::operator ~ () const throw (cvmexception);
scbmatrix& scbmatrix::conj (const scbmatrix& m) throw (cvmexception);
scbmatrix& scbmatrix::conj () throw (cvmexception);
```

implement complex band matrix conjugation. First operator creates object of type `scbmatrix` as conjugated calling band matrix (it throws `cvmexception` in case of memory allocation failure). Second function sets calling matrix to be equal to matrix `m` conjugated (it throws `cvmexception` in case of not appropriate sizes or numbers of sub- or super-diagonals of the operands), third one makes it to be equal to conjugated itself (it also throws `cvmexception` in case of memory allocation failure). See also `scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
              10., 11., 12., 13., 14., 15., 16.};
scbmatrix m((std::complex<double>*)a,4,1,0);
scbmatrix mc(4,0,1);
std::cout << m << std::endl << ~m << std::endl ;
mc.conj(m);
std::cout << mc << std::endl;
mc.conj();
std::cout << mc;
```

prints

```
(1,2) (0,0) (0,0) (0,0)
(3,4) (5,6) (0,0) (0,0)
(0,0) (7,8) (9,10) (0,0)
(0,0) (0,0) (11,12) (13,14)

(1,-2) (3,-4) (0,0) (0,0)
(0,0) (5,-6) (7,-8) (0,0)
(0,0) (0,0) (9,-10) (11,-12)
(0,0) (0,0) (0,0) (13,-14)

(1,-2) (3,-4) (0,0) (0,0)
(0,0) (5,-6) (7,-8) (0,0)
(0,0) (0,0) (9,-10) (11,-12)
(0,0) (0,0) (0,0) (13,-14)

(1,2) (0,0) (0,0) (0,0)
```

(3,4) (5,6) (0,0) (0,0)  
(0,0) (7,8) (9,10) (0,0)  
(0,0) (0,0) (11,12) (13,14)

**2.12.47 transposition**

Operator and functions

```
scbmatrix scbmatrix::operator ! () const throw (cvmexception);
scbmatrix& scbmatrix::transpose (const scbmatrix& m) throw (cvmexception);
scbmatrix& scbmatrix::transpose () throw (cvmexception);
```

implement complex band matrix transposition (*not* conjugation). First operator creates object of type `scbmatrix` as transposed calling matrix (it throws `cvmexception` in case of memory allocation failure). Second function sets calling matrix to be equal to matrix `m` transposed (it throws `cvmexception` in case of not appropriate sizes of the operands), third one makes it to be equal to transposed itself (it also throws `cvmexception` in case of memory allocation failure). See also `scbmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
              10., 11., 12., 13., 14., 15., 16.};
scbmatrix m((std::complex<double>*)a,4,1,0);
scbmatrix mc(4,0,1);
std::cout << m << std::endl << !m << std::endl ;
mc.transpose(m);
std::cout << mc << std::endl;
mc.transpose();
std::cout << mc;
```

prints

```
(1,2) (0,0) (0,0) (0,0)
(3,4) (5,6) (0,0) (0,0)
(0,0) (7,8) (9,10) (0,0)
(0,0) (0,0) (11,12) (13,14)
```

```
(1,2) (3,4) (0,0) (0,0)
(0,0) (5,6) (7,8) (0,0)
(0,0) (0,0) (9,10) (11,12)
(0,0) (0,0) (0,0) (13,14)
```

```
(1,2) (3,4) (0,0) (0,0)
(0,0) (5,6) (7,8) (0,0)
(0,0) (0,0) (9,10) (11,12)
(0,0) (0,0) (0,0) (13,14)
```

```
(1,2) (0,0) (0,0) (0,0)
```

(3,4) (5,6) (0,0) (0,0)  
(0,0) (7,8) (9,10) (0,0)  
(0,0) (0,0) (11,12) (13,14)

### 2.12.48 operator \* (const cvector&)

Operator

```
cvector scbmatrix::operator * (const cvector& v) const  
throw (cvmexception);
```

creates object of type cvector as product of calling band matrix and vector v. It throws **cvmexception** if number of columns of calling matrix differs from size of vector v. Use **cvector::mult** in order to avoid new object creation. See also **scbmatrix** and **cvector**.  
Example:

```
using namespace cvm;
```

```
try {  
    scbmatrix m (4,1,0);  
    cvector v(4);  
    m.set(std::complex<double>(1.,1.));  
    v.set(std::complex<double>(1.,1.));  
  
    std::cout << m * v;  
}  
catch (std::exception& e) {  
    std::cout << "Exception: " << e.what () << std::endl;  
}
```

prints

```
(0,2) (0,4) (0,4) (0,4)
```



### 2.12.49 operator \* (const cmatrix&)

Operator

```
cmatrix scbmatrix::operator * (const cmatrix& m) const  
throw (cvmexception);
```

creates object of type `cmatrix` as product of calling band matrix and matrix `m`. It throws `cvmexception` if number of columns of calling matrix differs from number of rows of matrix `m`. Use `cmatrix::mult` in order to avoid new object creation. See also `cmatrix` and `scbmatrix`. Example:

```
using namespace cvm;
```

```
try {  
    scbmatrix mb(4,1,0);  
    cmatrix m(4,2);  
    mb.set(std::complex<double>(1.,1.));  
    m.set(std::complex<double>(1.,1.));  
  
    std::cout << mb * m;  
}  
catch (std::exception& e) {  
    std::cout << "Exception: " << e.what () << std::endl;  
}
```

prints

```
(0,2) (0,2)  
(0,4) (0,4)  
(0,4) (0,4)  
(0,4) (0,4)
```

### 2.12.50 operator \* (const scmatrix&)

Operator

```
scmatrix scbmatrix::operator * (const scmatrix& m) const  
throw (cvmexception);
```

creates object of type `scmatrix` as product of calling band matrix and matrix `m`. It throws `cvmexception` if the operands have different sizes. Use `cmatrix::mult` in order to avoid new object creation. See also `scmatrix` and `scbmatrix`. Example:

```
using namespace cvm;
```

```
try {  
    scbmatrix mb(4,1,0);  
    scmatrix m(4);  
    mb.set(std::complex<double>(1.,1.));  
    m.set(std::complex<double>(1.,1.));  
  
    std::cout << mb * m;  
}  
catch (std::exception& e) {  
    std::cout << "Exception: " << e.what () << std::endl;  
}
```

prints

```
(0,2) (0,2) (0,2) (0,2)  
(0,4) (0,4) (0,4) (0,4)  
(0,4) (0,4) (0,4) (0,4)  
(0,4) (0,4) (0,4) (0,4)
```

### 2.12.51 operator \* (const scbmatrix&)

Operator

```
scbmatrix scbmatrix::operator * (const scbmatrix& m) const  
throw (cvmexception);
```

creates object of type `scbmatrix` as product of calling band matrix and band matrix `m`. It throws `cvmexception` if the operands have different sizes. Use `cmatrix::mult` in order to avoid new object creation. See also `scbmatrix`. Example:

```
using namespace cvm;
```

```
try {  
    scbmatrix m1(5,1,0);  
    scbmatrix m2(5,1,1);  
    m1.set(std::complex<double>(1.,1.));  
    m2.set(std::complex<double>(1.,1.));  
  
    std::cout << m1 * m2;  
}  
catch (std::exception& e) {  
    std::cout << "Exception: " << e.what () << std::endl;  
}
```

prints

```
(0,2) (0,2) (0,0) (0,0) (0,0)  
(0,4) (0,4) (0,2) (0,0) (0,0)  
(0,2) (0,4) (0,4) (0,2) (0,0)  
(0,0) (0,2) (0,4) (0,4) (0,2)  
(0,0) (0,0) (0,2) (0,4) (0,4)
```

**2.12.52 low\_up**

Functions

```
scbmatrix&
scbmatrix::low_up (const scbmatrix& m, int* nPivots) throw (cvmexception);
scbmatrix
scbmatrix::low_up (int* nPivots) const throw (cvmexception);
```

compute LU factorization of calling band matrix as

$$A = PLU$$

where P is permutation matrix, L is lower triangular matrix with unit diagonal elements and U is upper triangular matrix. All functions store the result as matrix L without main diagonal combined with U. All functions return pivot indices as array of integers (it should support at least `m.size()` elements) pointed to by `nPivots` so *i*-th row was interchanged with `nPivots[i]`-th row. The first version sets calling matrix to be equal to matrix *m*'s LU factorization and the second one creates object of type `scbmatrix` as calling matrix's LU factorization. Functions throw `cvmexception` in case of inappropriate sizes of the operands or when matrix to be factorized is close to singular. The first version also changes numbers of super-diagonals to be equal to  $k_l + k_u$  in order to keep result of factorization. It is recommended to use `iarray` for pivot values. This function is provided mostly for solving multiple systems of linear equations using `scmatrix::solve_lu` function, See also `scbmatrix`. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
try {
    double a[] = {1., 2., 3., 4., 5., 6., 7., 8., 9.,
                  10., 11., 12.};
    scbmatrix ma((std::complex<double>*)a,3,1,0);
    scbmatrix mLU(3,1,0);
    cmatrix mb1(3,2); cvector vb1(3);
    cmatrix mb2(3,2); cvector vb2(3);
    cmatrix mx1(3,2); cvector vx1(3);
    cmatrix mx2(3,2); cvector vx2(3);
    iarray nPivots(3);
    double dErr = 0.;
    mb1.randomize_real(-1.,3.); mb1.randomize_imag(1.,5.);
    mb2.randomize_real(-2.,5.); mb2.randomize_imag(-3.,0.);
    vb1.randomize_real(-2.,4.); vb1.randomize_imag(-4.,1.);
    vb2.randomize_real(-3.,1.); vb2.randomize_imag(4.,5.);
```

```

    mLU.low_up(ma, nPivots);
    mx1 = ma.solve_lu (mLU, nPivots, mb1, dErr);
    std::cout << mx1 << dErr << std::endl << std::endl;
    mx2 = ma.solve_lu (mLU, nPivots, mb2);
    std::cout << mx2 << std::endl;;
    std::cout << ma * mx1 - mb1 << std::endl << ma * mx2 - mb2;

    vx1 = ma.solve_lu (mLU, nPivots, vb1, dErr);
    std::cout << vx1 << dErr << std::endl;
    vx2 = ma.solve_lu (mLU, nPivots, vb2);
    std::cout << vx2 << std::endl;;
    std::cout << ma * vx1 - vb1 << std::endl << ma * vx2 - vb2;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
}

prints

(1.20e+000,4.02e-002) (1.82e+000,1.23e+000)
(-6.55e-001,1.37e-001) (-6.41e-001,-8.72e-001)
(7.75e-001,4.70e-002) (5.35e-001,8.11e-001)
1.45e-015

(-4.52e-001,-2.68e-002) (-1.09e+000,2.01e-001)
(6.08e-001,-4.76e-001) (5.48e-001,-1.95e-001)
(-3.46e-001,1.57e-001) (-3.38e-001,-7.54e-002)

(0.00e+000,4.44e-016) (-2.22e-016,8.88e-016)
(-2.22e-016,2.22e-016) (0.00e+000,0.00e+000)
(-1.11e-016,0.00e+000) (-3.33e-016,-6.66e-016)

(0.00e+000,0.00e+000) (2.22e-016,2.22e-016)
(0.00e+000,0.00e+000) (4.44e-016,-2.22e-016)
(8.88e-016,5.55e-016) (0.00e+000,0.00e+000)
(-1.28e+000,-5.12e-001) (8.22e-001,1.59e-001) (-6.45e-001,-3.74e-001)
1.31e-015
(1.26e+000,1.50e+000) (-5.13e-001,-4.66e-001) (5.97e-001,7.01e-001)

(0.00e+000,8.88e-016) (-4.44e-016,4.44e-016) (-8.88e-016,0.00e+000)

(2.22e-016,-8.88e-016) (4.44e-016,-8.88e-016) (-2.22e-016,8.88e-016)

```

### 2.12.53 operator / (const cvector&)

Operator

```
cvector operator / (const cvector& vB) const throw (cvmexception);
```

returns solution  $x$  of linear equation  $A * x = b$  where calling matrix is square band matrix  $A$  and vector  $b$  is passed in parameter  $vB$ . This operator throws exception of type `cvmexception` in case of inappropriate sizes of the objects or when matrix  $A$  is close to singular. See also `cvector::solve`, `scmatrix::solve`, `cvector.operator %`, `cvector`, `scmatrix`, `scbmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (12);

try {
    scbmatrix ma(4,2,1);
    cvector vb(4);
    cvector vx(4);
    ma.randomize_real(-1.,1.);
    ma.randomize_imag(-1.,1.);
    vb.randomize_real(-2.,2.);
    vb.randomize_imag(-2.,2.);

    vx = ma / vb;

    std::cout << (ma * vx - vb).norm() << std::endl;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
+8.082545620881e-016
```

### 2.12.54 identity

Function

```
scbmatrix& scbmatrix::identity();
```

sets calling band matrix to be equal to identity matrix and returns reference to the matrix changed. Function doesn't change numbers of sub- and super-diagonals. See also [scbmatrix](#). Example:

```
using namespace cvm;
```

```
srbmatrix m(4);  
m.randomize(0.,1.);  
std::cout << m << std::endl;  
std::cout << m.identity();
```

prints

```
(0.576128,1.42595) (0,0) (0,0) (0,0)  
(0.956359,-0.919523) (0.869716,-0.704093) (0,0) (0,0)  
(0,0) (0.0959807,0.0616779) (0.632618,1.1793) (0,0)  
(0,0) (0,0) (0.532182,-0.870724) (0.338023,1.22892)  
  
(1,0) (0,0) (0,0) (0,0)  
(0,0) (1,0) (0,0) (0,0)  
(0,0) (0,0) (1,0) (0,0)  
(0,0) (0,0) (0,0) (1,0)
```

### 2.12.55 vanish

Function

```
scbmatrix& scbmatrix::vanish();
```

sets every element of calling band matrix to be equal to zero and returns reference to the matrix changed. This function is faster than `scbmatrix::set(TR)` with zero operand passed. See also `scbmatrix`. Example:

```
using namespace cvm;
```

```
scbmatrix m(4,1,0);  
m.randomize_real(0.,1.);  
m.randomize_imag(-1.,2.);  
std::cout << m << std::endl;  
std::cout << m.vanish();
```

prints

```
(0.584094,0.985931) (0,0) (0,0) (0,0)  
(0.197546,0.0150761) (0.483413,-0.733848) (0,0) (0,0)  
(0,0) (0.844356,1.97848) (0.814692,1.50194) (0,0)  
(0,0) (0,0) (0.118931,-0.720756) (0.936796,-0.582232)  
  
(0,0) (0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0) (0,0)
```



### 2.12.56 `randomize_real`

Function

```
scbmatrix& scbmatrix::randomize_real (TR dFrom, TR dTo);
```

fills real part of calling band matrix with pseudo-random numbers distributed between `dFrom` and `dTo`. Function returns reference to the matrix changed. See also [scbmatrix](#).  
Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
```

```
std::cout.precision (2);
```

```
scbmatrix m(3,0,1);
```

```
m.randomize_real(0.,3.);
```

```
std::cout << m;
```

prints

```
(1.78e+000,0.00e+000) (1.17e+000,0.00e+000) (0.00e+000,0.00e+000)
```

```
(0.00e+000,0.00e+000) (1.09e-002,0.00e+000) (6.05e-001,0.00e+000)
```

```
(0.00e+000,0.00e+000) (0.00e+000,0.00e+000) (2.49e+000,0.00e+000)
```

### 2.12.57 `randomize_imag`

Function

```
scbmatrix& scbmatrix::randomize_imag (TR dFrom, TR dTo);
```

fills imaginary part of calling band matrix with pseudo-random numbers distributed between `dFrom` and `dTo`. Function returns reference to the matrix changed. See also [scbmatrix](#). Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);  
scbmatrix m(3,0,1);  
m.randomize_imag(0.,3.);  
std::cout << m;
```

prints

```
(0.00e+000,1.80e+000) (0.00e+000,1.68e-001) (0.00e+000,0.00e+000)  
(0.00e+000,0.00e+000) (0.00e+000,1.05e+000) (0.00e+000,1.40e+000)  
(0.00e+000,0.00e+000) (0.00e+000,0.00e+000) (0.00e+000,1.98e+000)
```

## 2.13 srsmatrix

This is end-user class encapsulating symmetric matrix of real numbers.

```

template <typename TR>
class srsmatrix : public srmatrix <TR> {
public:
    srsmatrix ();
    explicit srsmatrix (int nMN);
    srsmatrix (TR* pD, int nMN, TR tol = cvmMachSp());
    srsmatrix (const TR* pD, int nMN, TR tol = cvmMachSp());
    srsmatrix (const srsmatrix& m);
    explicit srsmatrix (const rmatrix& m, TR tol = cvmMachSp());
    explicit srsmatrix (const rvector& v);
    srsmatrix (srsmatrix& m, int nRowCol, int nSize);
    TR operator () (int im, int in) const throw (cvmexception);
    const rvector operator () (int i) const throw (cvmexception);
    const rvector operator [] (int i) const throw (cvmexception);
    const rvector diag (int i) const throw (cvmexception);
    srsmatrix& operator = (const srsmatrix& m) throw (cvmexception);
    srsmatrix& assign (const rvector& v, TR tol = cvmMachSp())
        throw (cvmexception);
    srsmatrix& assign (const TR* pD, TR tol = cvmMachSp())
        throw (cvmexception);
    srsmatrix& assign (int nRowCol, const srsmatrix& m)
        throw (cvmexception);
    srsmatrix& set (TR x);
    srsmatrix& set (int nRow, int nCol, TR x);
    srsmatrix& set_diag (int i, const rvector& v) throw (cvmexception);
    srsmatrix& resize (int nNewMN) throw (cvmexception);
    bool operator == (const srsmatrix& m) const;
    bool operator != (const srsmatrix& m) const;
    srsmatrix& operator << (const srsmatrix& m) throw (cvmexception);
    srsmatrix operator + (const srsmatrix& m) const
        throw (cvmexception);
    srsmatrix operator - (const srsmatrix& m) const
        throw (cvmexception);
    srsmatrix& sum (const srsmatrix& m1,
        const srsmatrix& m2) throw (cvmexception);
    srsmatrix& diff (const srsmatrix& m1,
        const srsmatrix& m2) throw (cvmexception);
    srsmatrix& operator += (const srsmatrix& m) throw (cvmexception);
    srsmatrix& operator -= (const srsmatrix& m) throw (cvmexception);

```

```

srsmatrix operator - () const;
srsmatrix& operator ++ ();
srsmatrix& operator ++ (int);
srsmatrix& operator -- ();
srsmatrix& operator -- (int);
srsmatrix operator * (TR d) const;
srsmatrix operator / (TR d) const throw (cvmexception);
srsmatrix& operator *= (TR d);
srsmatrix& operator /= (TR d) throw (cvmexception);
srsmatrix& normalize ();
srsmatrix operator ~ () const throw (cvmexception);
srsmatrix& transpose (const srsmatrix& m) throw (cvmexception);
srsmatrix& transpose ();
rvector operator * (const rvector& v) const throw (cvmexception);
rmatrix operator * (const rmatrix& m) const throw (cvmexception);
srmatrix operator * (const srmatrix& m) const throw (cvmexception);
rvector operator / (const rvector& vB) const throw (cvmexception);
srsmatrix& syrk (TR alpha,
                const rvector& v, TR beta) throw (cvmexception);
srsmatrix& syrk (bool bTransp, TR alpha,
                const rmatrix& m, TR beta) throw (cvmexception);
srsmatrix& syr2k (TR alpha,
                const rvector& v1, const rvector& v2, TR beta)
                throw (cvmexception);
srsmatrix& syr2k (bool bTransp, TR alpha,
                const rmatrix& m1, const rmatrix& m2, TR beta)
                throw (cvmexception);
srsmatrix& inv (const srsmatrix& mArg) throw (cvmexception);
srsmatrix inv () const throw (cvmexception);
srsmatrix& exp (const srsmatrix& m,
                TR tol = cvmMachSp()) throw (cvmexception);
srsmatrix exp (TR tol = cvmMachSp()) const throw (cvmexception);
srsmatrix& polynom (const srsmatrix& m, const rvector& v)
                throw (cvmexception);
srsmatrix polynom (const rvector& v) const throw (cvmexception);
rvector eig (srmatrix& mEigVect) const throw (cvmexception);
rvector eig () const throw (cvmexception);
srmatrix cholesky () const throw (cvmexception);
srmatrix bunch_kaufman () const throw (cvmexception);
srsmatrix& identity ();
srsmatrix& vanish ();
srsmatrix& randomize (TR dFrom, TR dTo);
};

```

### 2.13.1 srsmatrix ()

Constructor

```
srsmatrix::srsmatrix ();
```

creates empty srsmatrix object. See also [srsmatrix](#). Example:

```
using namespace cvm;
```

```
srsmatrix m;  
std::cout << m.msize() << " " << m.nsize() << " " << m.size()  
          << std::endl << std::endl;  
m.resize (3);  
m.set(1.);  
std::cout << m;
```

prints

```
0 0 0
```

```
1 1 1
```

```
1 1 1
```

```
1 1 1
```

### 2.13.2 srsmatrix (int)

Constructor

```
explicit srsmatrix::srsmatrix (int nMN);
```

creates  $n \times n$  srsmatrix object where  $n$  is passed in `nMN` parameter. Constructor throws `cvmexception` in case of non-positive size passed or memory allocation failure. See also `srsmatrix`. Example:

```
using namespace cvm;
```

```
srsmatrix m(4);  
std::cout << m.msize() << " " << m.nsize() << " " << m.size()  
          << std::endl << std::endl;  
m.set(1.);  
std::cout << m;
```

prints

```
4 4 16
```

```
1 1 1 1  
1 1 1 1  
1 1 1 1  
1 1 1 1
```

**2.13.3 srsmatrix (TR\*,int)**

Constructor

```
srsmatrix::srsmatrix (TR* pD, int nMN, TR tol = cvmMachSp());
```

creates  $n \times n$  srsmatrix object where  $n$  is passed in nMN parameter. Unlike others, this constructor *does not allocate memory*. It just shares memory with array pointed to by pD. Constructor throws **cvexception** if the matrix created doesn't appear to be symmetric or in case of non-positive size passed. Symmetry tolerance is set by parameter tol. If subsequent application flow would change the array passed so it becomes not symmetric matrix anymore then results are not predictable. See also **srsmatrix**, **srsmatrix (const TR\*,int)**. Example:

```
using namespace cvm;
double a[] = {1., 1., 1., 1., 1., 1., 1., 1., 1.};
srsmatrix m (a, 3);
m.set(2,1,5.);
std::cout << m << std::endl;
std::cout << a[0] << " " << a[1] << " " << a[2] << " "
          << a[3] << " " << a[4] << " " << a[5] << " " << std::endl;
```

prints

```
1 5 1
5 1 1
1 1 1

1 5 1 5 1 1
```

**2.13.4 srsmatrix (const TR\*,int)**

Constructor

```
srsmatrix::srsmatrix (const TR* pD, int nMN, TR tol = cvmMachSp());
```

creates  $n \times n$  srsmatrix object where  $n$  is passed in nMN parameter and copies  $n * n$  elements of array pD to it by columns. Constructor throws `cvmexception` if the matrix created doesn't appear to be symmetric or in case of non-positive size passed. Symmetry tolerance is set by parameter tol. See also `srsmatrix`, `srsmatrix (TR*,int)`. Example:

```
using namespace cvm;
const double a[] = {1., 1., 1., 1., 1., 1., 1., 1., 1.};
srsmatrix m (a, 3);
m.set(2,1,5.);
std::cout << m << std::endl;
std::cout << a[0] << " " << a[1] << " " << a[2] << " "
          << a[3] << " " << a[4] << " " << a[5] << " " << std::endl;
```

prints

```
1 5 1
5 1 1
1 1 1

1 1 1 1 1 1
```



### 2.13.5 srsmatrix (const srsmatrix&)

Copy constructor

```
srsmatrix::srsmatrix (const srsmatrix& m);
```

creates srsmatrix object as a copy of m. Constructor throws `cvmexception` in case of memory allocation failure. See also `srsmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};  
srsmatrix m (a, 3);  
srmatrix mc(m);  
m.set(2,3,7.77);  
std::cout << m << std::endl << mc;
```

prints

```
1 2 3  
2 5 7.77  
3 7.77 9
```

```
1 2 3  
2 5 6  
3 6 9
```

### 2.13.6 srsmatrix (const rmatrix&)

Constructor

```
explicit srsmatrix::srsmatrix (const rmatrix& m, TR tol = cvmMachSp());
```

creates srsmatrix object as a copy of matrix  $m$ . It's assumed that  $m \times n$  matrix  $m$  must have equal sizes, i.e.  $m = n$  is satisfied, and must be symmetric. Symmetry tolerance is set by parameter `tol`. Constructor throws `cvmexception` if this is not true or in case of memory allocation failure. See also `srsmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};  
rmatrix m(a, 3, 3);  
srsmatrix ms(m);  
std::cout << ms;
```

prints

```
1 2 3  
2 5 6  
3 6 9
```

### 2.13.7 srsmatrix (const rvector&)

Constructor

```
explicit srsmatrix::srsmatrix (const rvector& v);
```

creates srsmatrix object of size `v.size()` by `v.size()` and assigns vector `v` to its main diagonal. Constructor throws `cvmexception` in case of memory allocation failure. See also `srsmatrix`, `rvector`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5.};  
const rvector v(a, 5);  
srsmatrix m(v);  
std::cout << m;
```

prints

```
1 0 0 0 0  
0 2 0 0 0  
0 0 3 0 0  
0 0 0 4 0  
0 0 0 0 5
```

### 2.13.8 submatrix

Submatrix constructor

```
srsmatrix::srsmatrix (srsmatrix& m, int nRowCol, int nSize);
```

creates srmatrix object as *submatrix* of symmetric matrix *m*. It means that the matrix created shares memory with some part of *m*. This part is defined by its upper left corner (parameter *nRowCol*, *l*-based) and its size (parameter *nSize*). See also [srsmatrix](#). Example:

```
using namespace cvm;
```

```
srsmatrix m(5);  
srsmatrix subm(m, 2, 2);  
subm.set(1.);  
std::cout << m;
```

prints

```
0 0 0 0 0  
0 1 1 0 0  
0 1 1 0 0  
0 0 0 0 0
```

### 2.13.9 operator (,)

Indexing operator

```
TR srsmatrix::operator () (int im, int in) const throw (cvmexception);
```

provides access to a particular element of calling matrix. Unlike indexing operators in other classes, this operator doesn't return *l-value* because this would make the matrix non-symmetric. Operator is *l-based*. It throws `cvmexception` if some of parameters passed is outside of `[1,msize()]` range. See also `srsmatrix`, `Matrix::msize()`, `Matrix::nsize()`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};  
srsmatrix m(a, 3);  
std::cout << m(1,1) << " " << m(2,3) << std::endl;
```

prints

```
1 6
```

### 2.13.10 operator ()

Indexing operator

```
const rvector srsmatrix::operator () (int i) const throw (cvmexception);
```

provides access to *i*-th column of calling matrix. Unlike indexing operators in other classes, this operator doesn't return *l-value* because this would make the matrix non-symmetric. Operator creates object of class `rvector` as *copy* of a column and therefore it's *not l-value*. Operator is *l-based*. It throws `cvmexception` if parameter *i* is outside of `[1, nsize()]` range. See also `srsmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};  
srsmatrix m(a, 3);  
std::cout << m(1) << m(2) << m(3);
```

prints

```
1 2 3  
2 5 6  
3 6 9
```

### 2.13.11 operator []

Indexing operator

```
const rvector srsmatrix::operator [] (int i) const throw (cvmexception);
```

provides access to *i*-th row of calling matrix. Unlike indexing operators in other classes, this operator doesn't return *l-value* because this would make the matrix non-symmetric. Operator creates object of class `rvector` as *copy* of a row and therefore it's *not l-value*. Operator is *l-based*. It throws `cvmexception` if parameter *i* is outside of `[1,msize()]` range. See also `srsmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};  
srsmatrix m(a, 3);  
std::cout << m[1] << m[2] << m[3];
```

prints

```
1 2 3  
2 5 6  
3 6 9
```

**2.13.12 diag**

Function

```
const rvector srsmatrix::diag (int i) const throw (cvmexception);
```

provides access to  $i$ -th diagonal of calling matrix, where  $i = 0$  for main diagonal,  $i < 0$  for lower diagonals and  $i > 0$  for upper ones. Unlike `diag` function in other classes, this one doesn't return *l-value* because this would make the matrix non-symmetric. Function creates object of class `rvector` as *copy* of a diagonal and therefore it's *not l-value*. Function is *l-based*. It throws `cvmexception` if parameter  $i$  is outside of  $[-m\text{size}()+1, n\text{size}()-1]$  range. See also `srsmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};
srsmatrix m(a, 3);
std::cout << m << std::endl
           << m.diag(-2) << m.diag(-1) << m.diag(0)
           << m.diag(1) << m.diag(2);
```

prints

```
1 2 3
2 5 6
3 6 9

3
2 6
1 5 9
2 6
3
```



### 2.13.13 operator = (const srsmatrix&)

Operator

```
srsmatrix& srsmatrix::operator = (const srsmatrix& m)
throw (cvmexception);
```

sets every element of calling symmetric matrix to value of appropriate element of symmetric matrix `m` and returns reference to the matrix changed. Operator throws `cvmexception` in case of different sizes of the operands. See also `srsmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};
const srsmatrix m1(a, 3);
srsmatrix m2(3);
```

```
m2 = m1;
std::cout << m2;
```

prints

```
1 2 3
2 5 6
3 6 9
```

### 2.13.14 assign (const TR\*)

Function

```
srsmatrix& srsmatrix::assign (const rvector& v, TR tol = cvmMachSp())  
throw (cvmexception);
```

```
srsmatrix& srsmatrix::assign (const TR* pD, TR tol = cvmMachSp())  
throw (cvmexception);
```

sets every element of calling matrix to value of appropriate element of vector *v* or array pointed to by *pD* and returns reference to the matrix changed. In first version it's assumed that vector passed is long enough to fill calling matrix. Function throws **cvmexception** otherwise. Function throws **cvmexception** if the matrix changed doesn't appear to be symmetric. Symmetry tolerance is set by parameter *tol*. See also **srsmatrix**. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};  
srsmatrix m(3);  
m.assign(a);  
std::cout << m;
```

prints

```
1 2 3  
2 5 6  
3 6 9
```

### 2.13.15 assign (int, const srsmatrix&)

Function

```
srsmatrix& srsmatrix::assign (int nRowCol, const srsmatrix& m)
throw (cvmexception);
```

sets main sub-matrix of calling symmetric matrix beginning with  $\ell$ -based row nRowCol to symmetric matrix m and returns reference to the matrix changed. Function throws `cvmexception` if nRowCol is not positive or matrix m doesn't fit. See also `srsmatrix`. Example:

```
using namespace cvm;
```

```
srsmatrix m1(5);
srsmatrix m2(2);
m1.set(1.);
m2.set(2.);
m1.assign(2,m2);
std::cout << m1;
```

prints

```
1 1 1 1 1
1 2 2 1 1
1 2 2 1 1
1 1 1 1 1
1 1 1 1 1
```

### 2.13.16 set (TR)

Function

```
srsmatrix& srsmatrix::set (TR x);
```

sets every element of calling matrix to value of parameter x and returns reference to the matrix changed. See also [srsmatrix](#). Example:

```
using namespace cvm;
```

```
srsmatrix m(3);  
m.set(3.);  
std::cout << m;
```

prints

```
3 3 3  
3 3 3  
3 3 3
```

### 2.13.17 set (int,int,TR)

Function

```
srsmatrix& srsmatrix::set (int nRow, int nCol, TR x);
```

sets both elements located on nRow's row and nCol's column and on nCol's row and nRow's column to value of parameter x and returns reference to the matrix changed (thus the matrix remains symmetric). Parameters passed are *ℓ-based*. Function throws *cvmexception* if any of the parameters passed is outside of [1,msize()] range. See also *srsmatrix*. Example:

```
using namespace cvm;
```

```
srsmatrix m(3);  
m.set(3.);  
m.set(1,3,7.);  
std::cout << m;
```

prints

```
3 3 7  
3 3 3  
7 3 3
```

### 2.13.18 set\_diag (int,rvector)

Function

```
srsmatrix& srsmatrix::set_diag (int i, const rvector& v)
throw (cvmexception);
```

assigns vector *v* to *i*-th diagonal of calling matrix, where  $i = 0$  for main diagonal,  $i < 0$  for lower diagonals and  $i > 0$  for upper ones. If  $i \neq 0$ , then function assigns the vector to both *i*-th and  $-i$ -th diagonals (thus calling matrix remains symmetric). Function returns reference to the matrix changed. Parameter *i* is *not l-based*. Function throws **cvmexception** if parameter *i* is outside of  $[-m\text{size}()+1, n\text{size}()-1]$  range or if vector *v* passed has size not equal to  $m\text{size}()-\text{abs}(i)$ . See also **srsmatrix**. Example:

```
using namespace cvm;
```

```
srsmatrix m(3);
rvector v(2);
m.set(3.);
v.set(1.);
m.set_diag(1,v);
std::cout << m;
```

prints

```
3 1 3
1 3 1
3 1 3
```

### 2.13.19 `resize`

Function

```
srsmatrix& srsmatrix::resize (int nNewMN) throw (cvmexception);
```

changes size of calling matrix to `nNewMN` by `nNewMN` and returns reference to the matrix changed. In case of increasing of its size, calling matrix is filled up with zeroes. Function throws `cvmexception` in case of negative size passed or memory allocation failure. See also `srsmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};  
srsmatrix m(a,3);  
std::cout << m << std::endl;  
m.resize(4);  
std::cout << m;
```

prints

```
1 2 3  
2 5 6  
3 6 9
```

```
1 2 3 0  
2 5 6 0  
3 6 9 0  
0 0 0 0
```

### 2.13.20 operator ==

Operator

```
bool srsmatrix::operator == (const srsmatrix& m) const;
```

compares calling matrix with symmetric matrix `m` and returns true if they have the same sizes and their appropriate elements differ by not more than the **smallest normalized positive number**. Returns false otherwise. See also `srsmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 2., 3.};  
srsmatrix m1(a, 2);  
srsmatrix m2(2);  
m2.set(1,1,1.);  
m2.set(1,2,2.);  
m2.set(2,2,3.);
```

```
std::cout << (m1 == m2) << std::endl;
```

prints

```
1
```



### 2.13.21 operator !=

Operator

```
bool srsmatrix::operator != (const srsmatrix& m) const;
```

compares calling matrix with symmetric matrix `m` and returns `true` if they have different sizes or at least one of their appropriate elements differs by more than the **smallest normalized positive number**. Returns `false` otherwise. See also `srsmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 2., 3.};  
srsmatrix m1(a, 2);  
srsmatrix m2(2);  
m2.set(1,1,1.0001);  
m2.set(1,2,2.);  
m2.set(2,2,3.);
```

```
std::cout << (m1 != m2) << std::endl;
```

prints

```
1
```

### 2.13.22 operator <<

Operator

```
srsmatrix& srsmatrix::operator << (const srsmatrix& m)
throw (cvmexception);
```

destroys calling matrix, creates new one as a copy of `m` and returns reference to the matrix changed. Operator throws `cvmexception` in case of memory allocation failure. See also `srsmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};
srsmatrix m(a, 3);
srsmatrix mc(1);
std::cout << m << std::endl << mc << std::endl;
mc << m;
std::cout << mc;
```

prints

```
1 2 3
2 5 6
3 6 9
```

```
0
```

```
1 2 3
2 5 6
3 6 9
```

### 2.13.23 operator +

Operator

```
srsmatrix srsmatrix::operator + (const srsmatrix& m) const  
throw (cvmexception);
```

creates object of type `srsmatrix` as sum of calling symmetric matrix and symmetric matrix `m`. It throws `cvmexception` in case of different sizes of the operands. See also `srsmatrix::sum`, `srsmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};  
srsmatrix m1(a, 3);  
srsmatrix m2(3);  
m2.set(1.);  
std::cout << m1 + m2 << std::endl << m1 + m1;
```

prints

```
2 3 4  
3 6 7  
4 7 10
```

```
2 4 6  
4 10 12  
6 12 18
```

### 2.13.24 operator -

Operator

```
srsmatrix srsmatrix::operator - (const srsmatrix& m) const  
throw (cvmexception);
```

creates object of type `srsmatrix` as difference of calling symmetric matrix and symmetric matrix `m`. It throws `cvmexception` in case of different sizes of the operands. See also `srsmatrix::diff`, `srsmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};  
srsmatrix m1(a, 3);  
srsmatrix m2(3);  
m2.set(1.);  
std::cout << m1 - m2 << std::endl << m1 - m1;
```

prints

```
0 1 2  
1 4 5  
2 5 8
```

```
0 0 0  
0 0 0  
0 0 0
```

### 2.13.25 sum

Function

```
srsmatrix& srsmatrix::sum (const srsmatrix& m1, const srsmatrix& m2)
throw (cvexception);
```

assigns result of addition of symmetric matrices m1 and m2 to calling symmetric matrix and returns reference to the matrix changed. It throws `cvexception` in case of different sizes of the operands. See also `srsmatrix::operator +`, `srsmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};
const srsmatrix m1(a, 3);
srsmatrix m2(3);
srsmatrix m(3);
m2.set(1.);
```

```
std::cout << m.sum(m1, m2) << std::endl;
std::cout << m.sum(m, m2);
```

prints

```
2 3 4
3 6 7
4 7 10
```

```
3 4 5
4 7 8
5 8 11
```

### 2.13.26 diff

Function

```
srsmatrix& srsmatrix::diff (const srsmatrix& m1, const srsmatrix& m2)
throw (cvmexception);
```

assigns result of subtraction of symmetric matrices m1 and m2 to calling symmetric matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. See also `srsmatrix::operator -`, `srsmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};
const srsmatrix m1(a, 3);
srsmatrix m2(3);
srsmatrix m(3);
m2.set(1.);
```

```
std::cout << m.diff(m1, m2) << std::endl;
std::cout << m.diff(m, m2);
```

prints

```
0 1 2
1 4 5
2 5 8
```

```
-1 0 1
0 3 4
1 4 7
```

### 2.13.27 operator +=

Operator

```
srsmatrix& srsmatrix::operator += (const srsmatrix& m)
throw (cvmexception);
```

adds symmetric matrix `m` to calling symmetric matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. See also `srsmatrix::operator +`, `srsmatrix::sum`, `srsmatrix`. Example:

```
using namespace cvm;
```

```
srsmatrix m1(3);
srsmatrix m2(3);
m1.set(1.);
m2.set(2.);
```

```
m1 += m2;
std::cout << m1 << std::endl;
```

```
// well, you can do this too, but temporary object would be created
m2 += m2;
std::cout << m2;
```

prints

```
3 3 3
3 3 3
3 3 3
```

```
4 4 4
4 4 4
4 4 4
```

### 2.13.28 operator -=

Operator

```
srsmatrix& srsmatrix::operator -= (const srsmatrix& m)
throw (cvmexception);
```

subtracts symmetric matrix `m` from calling symmetric matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. See also `srsmatrix::operator -`, `srsmatrix::diff`, `srsmatrix`. Example:

```
using namespace cvm;
```

```
srsmatrix m1(3);
srsmatrix m2(3);
m1.set(1.);
m2.set(2.);
```

```
m1 -= m2;
std::cout << m1 << std::endl;
```

```
// well, you can do this too, but temporary object would be created
```

```
m2 -= m2;
std::cout << m2;
```

prints

```
-1 -1 -1
-1 -1 -1
-1 -1 -1
```

```
0 0 0
0 0 0
0 0 0
```



### 2.13.29 operator - ()

Operator

```
srsmatrix srsmatrix::operator - () const throw (cvmexception);
```

creates object of type srsmatrix as calling symmetric matrix multiplied by  $-1$ . See also [srsmatrix](#). Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};  
srsmatrix m(a, 3);  
std::cout << -m;
```

prints

```
-1 -2 -3  
-2 -5 -6  
-3 -6 -9
```

### 2.13.30 operator ++

Operator

```
srsmatrix& srsmatrix::operator ++ ();  
srsmatrix& srsmatrix::operator ++ (int);
```

adds identity matrix to calling symmetric matrix and returns reference to the matrix changed. See also [srsmatrix](#). Example:

```
using namespace cvm;
```

```
srsmatrix m(4);  
m.set(4.);  
m++;  
std::cout << m << std::endl;  
std::cout << ++m;
```

prints

```
5 4 4 4  
4 5 4 4  
4 4 5 4  
4 4 4 5
```

```
6 4 4 4  
4 6 4 4  
4 4 6 4  
4 4 4 6
```

### 2.13.31 operator --

Operator

```
srsmatrix& srsmatrix::operator -- ();  
srsmatrix& srsmatrix::operator -- (int);
```

subtracts identity matrix from calling symmetric matrix and returns reference to the matrix changed. See also [srsmatrix](#). Example:

```
using namespace cvm;
```

```
srsmatrix m(4);  
m.set(4.);  
m--;  
std::cout << m << std::endl;  
std::cout << --m;
```

prints

```
3 4 4 4  
4 3 4 4  
4 4 3 4  
4 4 4 3
```

```
2 4 4 4  
4 2 4 4  
4 4 2 4  
4 4 4 2
```

### 2.13.32 operator \* (TR)

Operator

```
srsmatrix srsmatrix::operator * (TR d) const;
```

creates object of type srsmatrix as product of calling symmetric matrix and number d.  
See also `srsmatrix::operator *= , srsmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};  
srsmatrix m(a,3);  
std::cout << m * 5.;
```

prints

```
5 10 15  
10 25 30  
15 30 45
```

### 2.13.33 operator / (TR)

Operator

```
srsmatrix srsmatrix::operator / (TR d) const throw (cvmexception);
```

creates object of type `srsmatrix` as quotient of calling symmetric matrix and number `d`. It throws `cvmexception` if `d` has absolute value equal or less than the **smallest normalized positive number**. See also `srsmatrix::operator /=` , `srsmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};  
srsmatrix m(a,3);  
std::cout << m / 4.;
```

prints

```
0.25 0.5 0.75  
0.5 1.25 1.5  
0.75 1.5 2.25
```

### 2.13.34 operator \*= (TR)

Operator

```
srsmatrix& srsmatrix::operator *= (TR d);
```

multiplies calling symmetric matrix by number d and returns reference to the matrix changed. See also `srsmatrix::operator *`, `srsmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};  
srsmatrix m(a,3);  
m *= 2.;  
std::cout << m;
```

prints

```
2 4 6  
4 10 12  
6 12 18
```

### 2.13.35 operator /= (TR)

Operator

```
srsmatrix& srsmatrix::operator /= (TR d) throw (cvmexception);
```

divides calling symmetric matrix by number d and returns reference to the matrix changed. It throws **cvmexception** if d has absolute value equal or less than the **smallest normalized positive number**. See also **srsmatrix::operator /** , **srsmatrix**. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};  
srsmatrix m(a,3);  
m /= 2.;  
std::cout << m;
```

prints

```
0.5 1 1.5  
1 2.5 3  
1.5 3 4.5
```

### 2.13.36 normalize

Function

```
srsmatrix& srsmatrix::normalize ();
```

normalizes calling symmetric matrix so its **Euclidean norm** becomes equal to 1 if it was greater than the **smallest normalized positive number** before the call (otherwise function does nothing). See also **srsmatrix**. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (3);  
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};  
srsmatrix m(a,3);  
m.normalize();  
std::cout << m << m.norm() << std::endl;
```

prints

```
6.984e-002 1.397e-001 2.095e-001  
1.397e-001 3.492e-001 4.191e-001  
2.095e-001 4.191e-001 6.286e-001  
1.000e+000
```



### 2.13.37 transposition

Operator and functions

```
srsmatrix srsmatrix::operator ~ () const throw (cvmexception);  
srsmatrix& srsmatrix::transpose (const srsmatrix& m) throw (cvmexception);  
srsmatrix& srsmatrix::transpose ();
```

do nothing since calling matrix is symmetric. They are provided to reimplement similar member functions and operator of the class `srmatrix`. See also `srsmatrix`, `srmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};  
srsmatrix m(a,3);  
std::cout << m - ~m;
```

prints

```
0 0 0  
0 0 0  
0 0 0
```

### 2.13.38 operator \* (const rvector&)

Operator

```
rvector srsmatrix::operator * (const rvector& v) const  
throw (cvmexception);
```

creates object of type rvector as product of calling symmetric matrix and vector v. It throws `cvmexception` if number of columns of calling matrix differs from size of vector v. Use `rvector::mult` in order to avoid new object creation. See also `srsmatrix` and `rvector`. Example:

```
using namespace cvm;
```

```
try {  
    srsmatrix m (4);  
    rvector v(4);  
    m.set(1.);  
    v.set(1.);  
  
    std::cout << m * v;  
}  
catch (std::exception& e) {  
    std::cout << "Exception: " << e.what () << std::endl;  
}
```

prints

```
4 4 4 4
```

### 2.13.39 operator \* (const rmatrix&)

Operator

```
rmatrix srsmatrix::operator * (const rmatrix& m) const  
throw (cvmexception);
```

creates object of type `rmatrix` as product of calling symmetric matrix and matrix `m`. It throws `cvmexception` if number of columns of calling matrix differs from number of rows of matrix `m`. Use `rmatrix::mult` in order to avoid new object creation. See also `rmatrix` and `srsmatrix`. Example:

```
using namespace cvm;
```

```
try {  
    srsmatrix ms(4);  
    rmatrix m(4,2);  
    ms.set(1.);  
    m.set(2.);  
    std::cout << ms * m;  
}  
catch (std::exception& e) {  
    std::cout << "Exception: " << e.what () << std::endl;  
}
```

prints

```
8 8  
8 8  
8 8  
8 8
```

**2.13.40 operator \* (const srmatrix&)**

Operator

```
srmatrix srsmatrix::operator * (const srmatrix& m) const
throw (cvmexception);
```

creates object of type srmatrix as product of calling symmetric matrix and matrix m. It throws `cvmexception` if operands have different sizes. Use `rmatrix::mult` in order to avoid new object creation. See also `srmatrix` and `srsmatrix`. Example:

```
using namespace cvm;
```

```
try {
    srsmatrix ms(3);
    srmatrix m(3);
    ms.set(1.);
    m.set(2.);
    std::cout << ms * m << std::endl;

    double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};
    const srsmatrix ms2(a, 3);
    std::cout << ms2 * ms;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
6 6 6
6 6 6
6 6 6
```

```
6 6 6
13 13 13
18 18 18
```

**2.13.41 operator / (const rvector&)**

Operator

```
rvector operator / (const rvector& vB) const throw (cvmexception);
```

returns solution  $x$  of linear equation  $A * x = b$  where calling matrix is square symmetric matrix  $A$  and vector  $b$  is passed in parameter  $vB$ . This operator throws exception of type `cvmexception` in case of inappropriate sizes of the objects or when matrix  $A$  is close to singular. See also `rvector::solve`, `srsmatrix::solve`, `rvector.operator %`, `rvector`, `srsmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (12);

try {
    double m[] = {3., 2., 3., 2., 5., 6., 3., 6., 9.};
    double b[] = {1., 2., 3.};
    srsmatrix ma(m, 3);
    rvector vb(b, 3);
    rvector vx(3);

    vx = ma / vb;

    std::cout << ma * vx - vb;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}

prints
-1.110223024625e-016 -2.220446049250e-016 +4.440892098501e-016
```

**2.13.42 syrk**

Functions

```
srsmatrix&
srsmatrix::syrk (TR alpha, const rvector& v, TR beta)
throw (cvmexception);
```

```
srsmatrix&
srsmatrix::syrk (bool bTransp, TR alpha, const rmatrix& m, TR beta)
throw (cvmexception);
```

call one of ?SYRK routines of the [BLAS library](#) performing matrix-vector operation defined for the first version as rank-1 update operation

$$A = \alpha \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} (v_1 \ v_2 \ \cdots \ v_n) + \beta A,$$

and for the second version as

$$A = \alpha MM^T + \beta A \quad \text{or} \quad A = \alpha M^T M + \beta A.$$

Here  $\alpha$  and  $\beta$  are real numbers (parameters alpha and beta),  $M$  is real matrix (parameter m),  $A$  is calling symmetric matrix and  $v$  is real vector (parameter v). First operation for the second version is performed if bTransp passed is false and second one otherwise. Function returns reference to the matrix changed and throws `cvmexception` in case of inappropriate sizes of the operands. See also `rvector`, `rmatrix` and `srsmatrix`.

Example:

```
using namespace cvm;
const treal alpha = 2.12;
const treal beta = -3.07;
rvector v(3);
srsmatrix ms(3), ms2(3);
v.randomize(-3.,2.);
ms.randomize(-1.,2.);
ms2 = ms;
ms.syrk (alpha, v, beta);
ms2 = alpha * srsmatrix(v.rank1update(v)) + beta * ms2;
std::cout << ms - ms2;
```

prints

```
0 0 0
0 0 0
0 0 0
```

Example:

```
using namespace cvm;
const treal alpha = 2.12;
const treal beta = -3.07;
rmatrix m(3,3);
srsmatrix ms(3), ms2(3);
m.randomize(-1.,2.);
ms.randomize(-1.,2.);
ms2 = ms;
ms.syrk (false, alpha, m, beta);
ms2 = alpha * srsmatrix (m * ~m) + beta * ms2;
std::cout << ms - ms2;
```

prints

```
0 0 0
0 0 0
0 0 0
```

Example:

```
using namespace cvm;
const treal alpha = 2.12;
const treal beta = -3.07;
rmatrix m(3,3);
srsmatrix ms(3), ms2(3);
m.randomize(-1.,2.);
ms.randomize(-1.,2.);
ms2 = ms;
ms.syrk (true, alpha, m, beta);
ms2 = alpha * srsmatrix (~m * m) + beta * ms2;
std::cout << ms - ms2;
```

prints

```
0 0 0
0 0 0
0 0 0
```

**2.13.43 syr2k**

Functions

```

srsmatrix&
srsmatrix::syr2k (TR alpha, const rvector& v1,
                 const rvector& v2, TR beta) throw (cvmexception);
srsmatrix&
srsmatrix::syr2k (bool bTransp, TR alpha, const rmatrix& m1,
                 const rmatrix& m2, TR beta) throw (cvmexception);

```

call one of ?SYR2K routines of the [BLAS library](#) performing matrix-vector operation defined for the first version as rank-1 update operation

$$A = \alpha v_1 v_2' + \alpha v_2 v_1' + \beta A,$$

and for the second version as

$$A = \alpha M_1 M_2^T + \alpha M_2 M_1^T + \beta A \quad \text{or} \quad A = \alpha M_1^T M_2 + \alpha M_2^T M_1 + \beta A.$$

Here  $\alpha$  and  $\beta$  are real numbers (parameters alpha and beta),  $M_1$  and  $M_2$  are real matrices (parameters m1 and m2),  $A$  is calling symmetric matrix and  $v_1$  and  $v_2$  are real vectors (parameters v1 and v2). First operation for the second version is performed if bTransp passed is false and second one otherwise. Function returns reference to the matrix changed and throws `cvmexception` in case of inappropriate sizes of the operands. See also `rvector`, `rmatrix` and `srsmatrix`. Example:

```

using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (2);
const treal alpha = 2.12;
const treal beta = -3.07;
rvector v1(4);
rvector v2(4);
srsmatrix ms(4), ms2(4);
v1.randomize(-1.,3.);
v2.randomize(-1.,3.);
ms.randomize(-1.,3.);
ms2 = ms;
ms.syr2k (alpha, v1, v2, beta);
ms2 = alpha * srsmatrix(v1.rank1update(v2) + v2.rank1update(v1)) + beta * ms2;
std::cout << ms - ms2;

```

prints



```
+0.00e+000 -1.33e-015 +0.00e+000 +0.00e+000
-1.33e-015 +0.00e+000 +4.44e-016 +0.00e+000
+0.00e+000 +4.44e-016 -5.55e-017 +0.00e+000
+0.00e+000 +0.00e+000 +0.00e+000 -8.88e-016
```

Example:

```
const treal alpha = 2.12;
const treal beta = -3.07;
rmatrix m1(3,3), m2(3,3);
srsmatrix ms(3), ms2(3);
m1.randomize(-2.,2.);
m2.randomize(-2.,2.);
ms.randomize(-1.,2.);
ms2 = ms;
ms.syr2k (false, alpha, m1, m2, beta);
ms2 = alpha * srsmatrix (m1 * ~m2 + m2 * ~m1) + beta * ms2;
std::cout << ms - ms2;
```

prints

```
+1.11e-015 +0.00e+000 +0.00e+000
+0.00e+000 +8.88e-016 +1.78e-015
+0.00e+000 +1.78e-015 -1.78e-015
```

Example:

```
const treal alpha = 2.12;
const treal beta = -3.07;
rmatrix m1(3,3), m2(3,3);
srsmatrix ms(3), ms2(3);
m1.randomize(-2.,2.);
m2.randomize(-2.,2.);
ms.randomize(-1.,2.);
ms2 = ms;
ms.syr2k (true, alpha, m1, m2, beta);
ms2 = alpha * srsmatrix (~m1 * m2 + ~m2 * m1) + beta * ms2;
std::cout << ms - ms2;
```

prints

```
+0.00e+000 +1.78e-015 +0.00e+000
+1.78e-015 +0.00e+000 +1.33e-015
+0.00e+000 +1.33e-015 +0.00e+000
```

**2.13.44 inv**

Functions

```
srsmatrix& srsmatrix::inv (const srsmatrix& m) throw (cvmexception);
srsmatrix srsmatrix::inv () const throw (cvmexception);
```

implement symmetric matrix inversion. The first version sets calling symmetric matrix to be equal to symmetric matrix *m* inverted and the second one creates object of type *srsmatrix* as inverted calling matrix. Functions throw *cvmexception* in case of inappropriate sizes of the operands or when matrix to be inverted is close to singular. See also *srsmatrix*. Example:

```
using namespace cvm;

std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (5);

double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.05};
const srsmatrix m(a, 3);
const srsmatrix mi = m.inv();

std::cout << mi << std::endl;
std::cout << mi * m - eye_real(3);
```

prints

```
1.85000e+002 -2.00000e+000 -6.00000e+001
-2.00000e+000 1.00000e+000 0.00000e+000
-6.00000e+001 0.00000e+000 2.00000e+001

0.00000e+000 0.00000e+000 0.00000e+000
0.00000e+000 0.00000e+000 0.00000e+000
0.00000e+000 0.00000e+000 0.00000e+000
```

**2.13.45 exp**

Functions

```
srsmatrix& srsmatrix::exp (const srsmatrix& m, TR tol = cvmMachSp())
throw (cvmexception);
```

```
srsmatrix srsmatrix::exp (TR tol = cvmMachSp()) const
throw (cvmexception);
```

compute exponent of calling symmetric matrix using Padé approximation defined as

$$R_{pq}(z) = D_{pq}(z)^{-1} N_{pq}(z) = 1 + z + \dots + z^p / p!,$$

where

$$N_{pq}(z) = \sum_{k=0}^p \frac{(p+q-k)! p!}{(p+q)! k! (q-k)!} z^k,$$

$$D_{pq}(z) = \sum_{k=0}^q \frac{(p+q-k)! p!}{(p+q)! k! (q-k)!} (-z)^k$$

along with matrix normalizing as described in [2], p. 572. Functions use DMEXP (or SMEXP for float version) FORTRAN subroutine implementing the algorithm. The first version sets calling symmetric matrix to be equal to exponent of symmetric matrix *m* and returns reference to the matrix changed. The second version creates object of type *srsmatrix* as exponent of calling matrix. The algorithm uses parameter *tol* as  $\varepsilon(p, q)$  in order to choose constants *p* and *q* so that

$$\varepsilon(p, q) \geq 2^{3-(p+q)} \frac{p!q!}{(p+q)!(p+q+1)!}.$$

This parameter is equal to the **largest relative spacing** by default. Functions throw **cvmexception** in case of inappropriate sizes of the operands or when LAPACK subroutine fails. See also **srsmatrix**. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (15);
```

```
double a[] = {1., 2., 1., 2., 0., -1., 1., -1., 2.};
const srsmatrix m(a, 3);
std::cout << m.exp();
```

```
prints
```

```
9.198262499129184e+000 5.558586002658855e+000 3.852443363622591e+000
5.558586002658857e+000 5.345819135506593e+000 -1.706142639036265e+000
3.852443363622590e+000 -1.706142639036266e+000 1.090440513816545e+001
```

Matlab output:

Columns 1 through 2

```
9.198262499129212e+000 5.558586002658862e+000
5.558586002658865e+000 5.345819135506588e+000
3.852443363622600e+000 -1.706142639036258e+000
```

Column 3

```
3.852443363622601e+000
-1.706142639036260e+000
1.090440513816545e+001
```

**2.13.46 polynomial**

Functions

```
srsmatrix& srsmatrix::polynom (const srsmatrix& m, const rvector& v)
throw (cvmexception);
```

```
srsmatrix srsmatrix::polynom (const rvector& v) const
throw (cvmexception);
```

compute symmetric matrix polynomial defined as

$$p(A) = b_0I + b_1A + \dots + b_qA^q$$

using the Horner's rule:

$$p(A) = \sum_{k=0}^r B_k(A^s)^k, \quad s = \text{floor}(\sqrt{q}), \quad r = \text{floor}(q/s)$$

where

$$B_k = \begin{cases} \sum_{i=0}^{s-1} b_{sk+i}A^i, & k = 0, 1, \dots, r-1 \\ \sum_{i=0}^{q-sr} b_{sr+i}A^i, & k = r. \end{cases}$$

See also [2], p. 568. Coefficients  $b_0, b_1, \dots, b_q$  are passed in parameter  $v$ , where  $q$  is equal to  $v.size()-1$ , so functions compute matrix polynomial equal to

$$v[1] * I + v[2] * m + \dots + v[v.size()] * m^{v.size()-1}$$

The first version sets calling symmetric matrix to be equal to the polynomial of symmetric matrix  $m$  and the second one creates object of type `srsmatrix` as the polynomial of calling symmetric matrix. Functions use DPOLY (or SPOLY for float version) FORTRAN subroutine implementing the Horner's algorithm. Functions throw `cvmexception` in case of inappropriate sizes of the operands. See also `srsmatrix`. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (7);
double a[] = {1., 2., 1., 2., 0., -1., 1., -1., 2.};
double av[] = {2.2, 1.3, 1.1, -0.9, 0.2,
               -0.45, 45, -30, 10, 3, 3.2};
const rvector v(av, 11);
const srsmatrix m(a, 3);

std::cout << m.polynom (v);
```

prints

```
6.2127400e+004 2.3998000e+004 3.4100550e+004
2.3998000e+004 2.8026850e+004 1.0102550e+004
3.4100550e+004 1.0102550e+004 5.2024850e+004
```

Matlab output:

Columns 1 through 2

```
6.2127400000000001e+004    2.399800000000000e+004
2.399800000000000e+004    2.802685000000000e+004
3.410055000000000e+004    1.010255000000000e+004
```

Column 3

```
3.410055000000000e+004
1.010255000000000e+004
5.202485000000000e+004
```

**2.13.47 eig**

Functions

```
rvector srsmatrix::eig (srmatrix& mEigVect) const throw (cvmexception);
rvector srsmatrix::eig () const throw (cvmexception);
```

solve the **symmetric eigenvalue problem** and return real vector with eigenvalues of calling symmetric matrix. The first version sets output parameter `mEigVect` to be equal to square matrix containing orthogonal eigenvectors as columns. All functions throw **cvmexception** in case of inappropriate sizes of the operands or in case of convergence error. See also **rvector**, **srmatrix** and **srsmatrix**. Example:

```
using namespace cvm;

std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (10);
double a[] = {1., 2., 1., 2., 0., -1., 1., -1., 2.};
const srsmatrix m(a, 3);
srmatrix me(3);
rvector v(3);

v = m.eig(me);
std::cout << v << std::endl;

std::cout << m * me(1) - me(1) * v(1);
std::cout << m * me(2) - me(2) * v(2);
std::cout << m * me(3) - me(3) * v(3);

prints

-2.0489173395e+000 2.3568958679e+000 2.6920214716e+000

4.4408920985e-016 0.0000000000e+000 5.5511151231e-016
-1.1102230246e-016 2.2204460493e-016 2.2204460493e-016
0.0000000000e+000 -1.1102230246e-016 -4.4408920985e-016
```

**2.13.48 Cholesky**

Function

```
srmatrix srsmatrix::cholesky () const throw (cvmexception);
```

forms the Cholesky factorization of symmetric positive-definite matrix  $A$  defined as

$$A = U^T U,$$

where  $U$  is upper triangular matrix. It utilizes one of ?POTRF routines of the [LAPACK library](#). Function creates object of type `srmatrix` as factorization of calling matrix. Function throws `cvmexception` in case of convergence error. See also `srmatrix` and `srsmatrix`. Example:

```
using namespace cvm;
```

```
try {
    double a[] = {1., 2., 1., 2., 5., -1., 1., -1., 20.};
    const srsmatrix m(a, 3);

    srmatrix h = m.cholesky();
    std::cout << h << std::endl;
    std::cout << ~h * h - m;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
1 2 1
0 1 -3
0 0 3.16228

0 0 0
0 0 0
0 0 0
```



### 2.13.49 Bunch-Kaufman

Function

```
srmatrix srmatrix::bunch_kaufman () throw (cvmexception);
```

forms the Bunch-Kaufman factorization of calling symmetric matrix (cited from the MKL library documentation):

$$A = PUDU^T P^T,$$

where  $A$  is calling symmetric matrix,  $P$  is permutation matrix,  $U$  and  $L$  are upper and lower triangular matrices with unit diagonal, and  $D$  is symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks.  $U$  and  $L$  have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of  $D$ . It utilizes one of ?SYTRF routines of the [LAPACK library](#). Function creates object of type `srmatrix` as factorization of calling matrix. Function throws `cvmexception` in case of convergence error. See also `srmatrix` and `srmatrix`. Function is mostly designed to be used for subsequent calls of ?SYTRS, ?SYCON and ?SYTRI routines of the [LAPACK library](#). Currently it's used internally in `srmatrix::det` flow when argument is symmetric but not positive-definite.

### 2.13.50 identity

Function

```
srsmatrix& srsmatrix::identity();
```

sets calling symmetric matrix to be equal to identity matrix and returns reference to the matrix changed. See also [srsmatrix](#). Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (3);  
srsmatrix m(3);  
m.randomize(0.,1.);
```

```
std::cout << m << std::endl;  
std::cout << m.identity();
```

prints

```
1.329e-001 8.527e-001 3.110e-001  
8.527e-001 6.152e-001 3.247e-001  
3.110e-001 3.247e-001 9.145e-001  
  
1.000e+000 0.000e+000 0.000e+000  
0.000e+000 1.000e+000 0.000e+000  
0.000e+000 0.000e+000 1.000e+000
```

### 2.13.51 vanish

Function

```
srsmatrix& srsmatrix::vanish();
```

sets every element of calling symmetric matrix to be equal to zero and returns reference to the matrix changed. See also [srsmatrix](#). Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (3);  
srsmatrix m(3);  
m.randomize(0.,1.);
```

```
std::cout << m << std::endl;  
std::cout << m.vanish();
```

prints

```
1.422e-001 1.477e-001 1.445e-001  
1.477e-001 8.893e-001 1.669e-002  
1.445e-001 1.669e-002 7.766e-001  
  
0.000e+000 0.000e+000 0.000e+000  
0.000e+000 0.000e+000 0.000e+000  
0.000e+000 0.000e+000 0.000e+000
```

### 2.13.52 `randomize`

Function

```
srsmatrix& srsmatrix::randomize (TR dFrom, TR dTo);
```

fills calling symmetric matrix with pseudo-random numbers distributed between `dFrom` and `dTo` keeping it to be symmetric. Function returns reference to the matrix changed. See also `srsmatrix`. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (7);
```

```
srsmatrix m(3);  
m.randomize(-2.,3.);  
std::cout << m;
```

prints

```
-1.2277291e+000 3.6610004e-001 2.1380047e+000  
3.6610004e-001 1.0336924e+000 -1.8565630e+000  
2.1380047e+000 -1.8565630e+000 1.7774285e+000
```

## 2.14 schmatrix

This is end-user class encapsulating hermitian matrix of complex numbers.

```
template <typename TR, typename TC>
class schmatrix : public scmatrix <TR,TC> {
public:
    schmatrix ();
    explicit schmatrix (int nMN);
    schmatrix (TC* pD, int nMN, TR tol = cvmMachSp());
    schmatrix (const TC* pD, int nMN, TR tol = cvmMachSp());
    schmatrix (const schmatrix& m);
    explicit schmatrix (const cmatrix& m, TR tol = cvmMachSp());
    explicit schmatrix (const rvector& v);
    explicit schmatrix (const srsmatrix& m);
    schmatrix (const TR* pRe, const TR* pIm, int nMN,
               TR tol = cvmMachSp());
    schmatrix (const srmatrix& mRe, const srmatrix& mIm,
               TR tol = cvmMachSp());
    schmatrix (schmatrix& m, int nRowCol, int nSize);
    TC operator () (int im, int in) const throw (cvmexception);
    const cvector operator () (int i) const throw (cvmexception);
    const cvector operator [] (int i) const throw (cvmexception);
    const cvector diag (int i) const throw (cvmexception);
    const srsmatrix real () const;
    const srmatrix imag () const;
    schmatrix& operator = (const schmatrix& m) throw (cvmexception);
    schmatrix& assign (const cvector& v, TR tol = cvmMachSp())
                     throw (cvmexception);
    schmatrix& assign (const TC* pD, TR tol = cvmMachSp())
                     throw (cvmexception);
    schmatrix& assign (int nRowCol, const schmatrix& m)
                     throw (cvmexception);
    schmatrix& set (int nRow, int nCol, TC z);
    schmatrix& set_diag (int i, const cvector& v) throw (cvmexception);
    schmatrix& set_main_diag (const rvector& v) throw (cvmexception);
    schmatrix& assign_real (const srsmatrix& m) throw (cvmexception);
    schmatrix& set_real (TR d) throw (cvmexception);
    schmatrix& resize (int nNewMN) throw (cvmexception);
    bool operator == (const schmatrix& v) const;
    bool operator != (const schmatrix& v) const;
    schmatrix& operator << (const schmatrix& m) throw (cvmexception);
    schmatrix operator + (const schmatrix& m) const
```

```

        throw (cvmexception);
schmatrix operator - (const schmatrix& m) const
        throw (cvmexception);
schmatrix& sum (const schmatrix& m1,
               const schmatrix& m2) throw (cvmexception);
schmatrix& diff (const schmatrix& m1,
               const schmatrix& m2) throw (cvmexception);
schmatrix& operator += (const schmatrix& m) throw (cvmexception);
schmatrix& operator -= (const schmatrix& m) throw (cvmexception);
schmatrix operator - () const;
schmatrix& operator ++ ();
schmatrix& operator ++ (int);
schmatrix& operator -- ();
schmatrix& operator -- (int);
schmatrix operator * (TR d) const;
schmatrix operator / (TR d) const throw (cvmexception);
scmatrix operator * (TC z) const;
scmatrix operator / (TC z) const throw (cvmexception);
schmatrix& operator *= (TR d);
schmatrix& operator /= (TR d) throw (cvmexception);
schmatrix& normalize ();
schmatrix operator ~ () const;
schmatrix operator ! () const;
schmatrix& conj (const schmatrix& m) throw (cvmexception);
schmatrix& conj ();
schmatrix& transpose (const schmatrix& m) throw (cvmexception);
schmatrix& transpose ();
cvector operator * (const cvector& v) const throw (cvmexception);
cmatrix operator * (const cmatrix& m) const throw (cvmexception);
scmatrix operator * (const scmatrix& m) const throw (cvmexception);
cvector operator / (const cvector& vB) const throw (cvmexception);
schmatrix& herk (TR alpha,
               const cvector& v, TR beta) throw (cvmexception);
schmatrix& herk (bool bTransp, TR alpha,
               const cmatrix& m, TR beta) throw (cvmexception);
schmatrix& her2k (TC alpha,
               const cvector& v1, const cvector& v2, TR beta)
               throw (cvmexception);
schmatrix& her2k (bool bTransp, TC alpha,
               const cmatrix& m1, const cmatrix& m2, TR beta)
               throw (cvmexception);
schmatrix& inv (const schmatrix& mArg) throw (cvmexception);
schmatrix inv () const throw (cvmexception);

```

```
schmatrix& exp (const schmatrix& m,  
               TR tol = cvmMachSp ()) throw (cvmexception);  
schmatrix exp (TR tol = cvmMachSp ()) const throw (cvmexception);  
schmatrix& polynom (const schmatrix& m, const rvector& v)  
                throw (cvmexception);  
schmatrix polynom (const rvector& v) const throw (cvmexception);  
rvector eig (scmatrix& mEigVect) const throw (cvmexception);  
rvector eig () const throw (cvmexception);  
scmatrix cholesky () const throw (cvmexception);  
scmatrix bunch_kaufman () const throw (cvmexception);  
schmatrix& identity ();  
schmatrix& vanish ();  
schmatrix& randomize_real (TR dFrom, TR dTo);  
schmatrix& randomize_imag (TR dFrom, TR dTo);  
};
```

### 2.14.1 schmatrix ()

Constructor

```
schmatrix::schmatrix ();
```

creates empty schmatrix object. See also [schmatrix](#). Example:

```
using namespace cvm;
```

```
schmatrix m;  
std::cout << m.msize() << " " << m.nsize() << " "  
           << m.size() << std::endl;
```

```
m.resize (3);  
std::cout << m;
```

prints

```
0 0 0  
(0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0)
```



### 2.14.2 schmatrix (int)

Constructor

```
explicit schmatrix::schmatrix (int nMN);
```

creates  $n \times n$  schmatrix object where  $n$  is passed in `nMN` parameter. Constructor throws `cvmexception` in case of non-positive size passed or memory allocation failure. See also `schmatrix`. Example:

```
using namespace cvm;
```

```
schmatrix m (4);  
std::cout << m.msize() << std::endl  
          << m.nsize() << std::endl  
          << m.size() << std::endl << m;
```

prints

```
4  
4  
16  
(0,0) (0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0) (0,0)
```

**2.14.3 schmatrix (TC\*,int)**

Constructor

```
schmatrix::schmatrix (TC* pD, int nMN, TR tol = cvmMachSp());
```

creates  $n \times n$  schmatrix object where  $n$  is passed in nMN parameter. Unlike others, this constructor *does not allocate memory*. It just shares memory with array pointed to by pD. Constructor throws **cvmexception** if the matrix created doesn't appear to be hermitian (tolerance is set by parameter tol) or in case of non-positive size passed. If subsequent application flow would change the array passed so it becomes not hermitian matrix any-more then results are not predictable. See also **schmatrix**, **schmatrix (const TC\*,int)**. Example:

```
using namespace cvm;
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m ((std::complex<double>*)a, 3);
m.set(2,1,std::complex<double>(8.,8.));
std::cout << m << std::endl;
std::cout << a[0] << " " << a[1] << " " << a[2] << " "
          << a[3] << " " << a[4] << " " << a[5] << " " << std::endl;
```

prints

```
(1,0) (8,-8) (-1,-2)
(8,8) (2,0) (0,-3)
(-1,2) (0,3) (3,0)
```

```
1 0 8 8 -1 2
```

**2.14.4 schmatrix (const TC\*,int)**

Constructor

```
schmatrix::schmatrix (const TC* pD, int nMN, TR tol = cvmMachSp());
```

creates  $n \times n$  schmatrix object where  $n$  is passed in nMN parameter and copies  $n * n$  elements of array pD to it by `columns`. Constructor throws `cvmexception` if the matrix created doesn't appear to be hermitian (tolerance is set by parameter tol) or in case of non-positive size passed. See also `schmatrix`, `schmatrix (TC*,int)`. Example:

```
using namespace cvm;
const double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
                   0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m ((const std::complex<double>*)a, 3);
m.set(2,1,std::complex<double>(8.,8.));
std::cout << m << std::endl;
std::cout << a[0] << " " << a[1] << " " << a[2] << " "
          << a[3] << " " << a[4] << " " << a[5] << " " << std::endl;
```

prints

```
(1,0) (8,-8) (-1,-2)
(8,8) (2,0) (0,-3)
(-1,2) (0,3) (3,0)
```

```
1 0 2 1 -1 2
```

**2.14.5 schmatrix (const schmatrix&)**

Copy constructor

```
schmatrix::schmatrix (const schmatrix& m)
```

creates schmatrix object as a copy of m. Constructor throws `cvmexception` in case of memory allocation failure. See also `schmatrix`. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left |
               std::ios::showpos);
```

```
std::cout.precision (1);
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
```

```
schmatrix m ((std::complex<double>*)a, 3);
```

```
schmatrix mc(m);
```

```
m.set(1,2, std::complex<double>(7.7,7.7));
```

```
std::cout << m << std::endl << mc;
```

prints

```
(+1.0e+000,+0.0e+000) (+7.7e+000,+7.7e+000) (-1.0e+000,-2.0e+000)
(+7.7e+000,-7.7e+000) (+2.0e+000,+0.0e+000) (+0.0e+000,-3.0e+000)
(-1.0e+000,+2.0e+000) (+0.0e+000,+3.0e+000) (+3.0e+000,+0.0e+000)
```

```
(+1.0e+000,+0.0e+000) (+2.0e+000,-1.0e+000) (-1.0e+000,-2.0e+000)
(+2.0e+000,+1.0e+000) (+2.0e+000,+0.0e+000) (+0.0e+000,-3.0e+000)
(-1.0e+000,+2.0e+000) (+0.0e+000,+3.0e+000) (+3.0e+000,+0.0e+000)
```

### 2.14.6 schmatrix (const cmatrix&)

Constructor

```
explicit schmatrix::schmatrix (const cmatrix& m, TR tol = cvmMachSp())
```

creates schmatrix object as a copy of matrix m. It's assumed that  $m \times n$  matrix m must have equal sizes, i.e.  $m = n$  is satisfied and it has to be hermitian one (tolerance is set by parameter tol). Constructor throws `cvexception` if this is not true or in case of memory allocation failure. See also `schmatrix` and `cmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};
```

```
cmatrix m ((std::complex<double>*)a, 3, 3);
```

```
schmatrix mch(m);
```

```
std::cout << mch;
```

prints

```
(1,0) (2,-1) (-1,-2)
```

```
(2,1) (2,0) (0,-3)
```

```
(-1,2) (0,3) (3,0)
```

**2.14.7 schmatrix (const rvector&)**

Constructor

```
explicit schmatrix::schmatrix (const rvector& v);
```

creates schmatrix object of size `v.size()` by `v.size()` and assigns vector `v` to its main diagonal. Constructor throws `cvmexception` in case of memory allocation failure. See also `schmatrix` and `rvector`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 4., 5.};
const rvector v (a, 5);
schmatrix m(v);
std::cout << m.msize() << " " << m.nsize() << " "
           << m.size() << std::endl << m;
```

prints

```
5 5 25
(1,0) (0,0) (0,0) (0,0) (0,0)
(0,0) (2,0) (0,0) (0,0) (0,0)
(0,0) (0,0) (3,0) (0,0) (0,0)
(0,0) (0,0) (0,0) (4,0) (0,0)
(0,0) (0,0) (0,0) (0,0) (5,0)
```

### 2.14.8 schmatrix (const srsmatrix&)

Constructor

```
explicit schmatrix::schmatrix (const srsmatrix& m);
```

creates schmatrix object having the same dimension as real symmetric matrix `m` and copies matrix `m` to its real part. See also [schmatrix](#) and [srsmatrix](#). Example:

```
using namespace cvm;
```

```
double a[] = {1., 2., 3., 2., 5., 6., 3., 6., 9.};  
srsmatrix m(a, 3);  
schmatrix mch(m);  
std::cout << mch;
```

prints

```
(1,0) (2,0) (3,0)  
(2,0) (5,0) (6,0)  
(3,0) (6,0) (9,0)
```

### 2.14.9 schmatrix (const TR\*,const TR\*,int)

Constructor

```
schmatrix::schmatrix (const TR* pRe, const TR1* pIm, int nMN,  
                      TR tol = cvmMachSp());
```

creates schmatrix object of size nMN by nMN and copies every element of arrays pointed to by pRe and pIm to real and imaginary part of the matrix created respectively. Use NULL pointer to fill up appropriate part with zero values. Constructor throws [cvmexception](#) if the matrix created doesn't appear to be hermitian (tolerance is set by parameter tol) or in case of memory allocation failure. See also [schmatrix](#). Example:

```
using namespace cvm;
```

```
double re[] = {1., 2., -1., 2., 2., 0., -1., 0., 3.};  
double im[] = {0., 1., 2., -1., 0., 3., -2., -3., 0.};  
schmatrix m(re, im, 3);  
std::cout << m;
```

prints

```
(1,0) (2,-1) (-1,-2)  
(2,1) (2,0) (0,-3)  
(-1,2) (0,3) (3,0)
```



### 2.14.10 schmatrix (const srmatrix&, const srmatrix&)

Constructor

```
schmatrix::schmatrix (const srmatrix& mRe, const srmatrix& mIm,  
                     TR tol = cvmMachSp());
```

creates schmatrix object of the same size as mRe and mIm has (it throws [cvmexception](#) if mRe and mIm have different sizes) and copies matrices mRe and mIm to real and imaginary part of the matrix created respectively. Constructor throws [cvmexception](#) if the matrix created doesn't appear to be hermitian (tolerance is set by parameter tol) or in case of memory allocation failure. See also [schmatrix](#), [srmatrix](#). Example:

```
using namespace cvm;
```

```
double re[] = {1., 2., -1., 2., 2., 0., -1., 0., 3.};  
double im[] = {0., 1., 2., -1., 0., 3., -2., -3., 0.};  
srmatrix mr(re, 3);  
srmatrix mi(im, 3);  
schmatrix m(mr, mi);  
std::cout << m;
```

prints

```
(1,0) (2,-1) (-1,-2)  
(2,1) (2,0) (0,-3)  
(-1,2) (0,3) (3,0)
```

**2.14.11 submatrix**

Submatrix constructor

```
schmatrix::schmatrix (schmatrix& m, int nRowCol, int nSize);
```

creates schmatrix object as *submatrix* of m. It means that the matrix object created shares memory with some part of m. This part is defined by its upper left corner (parameter nRowCol, *l-based*) and its size (parameter nSize). See also [schmatrix](#). Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m ((std::complex<double>*)a, 3);
schmatrix m2 (m, 1, 2);
```

```
std::cout << m2 << std::endl;
m2.set_real(7.7);
std::cout << m;
```

prints

```
(1,0) (2,-1)
(2,1) (2,0)
```

```
(7.7,0) (7.7,-1) (-1,-2)
(7.7,1) (7.7,0) (0,-3)
(-1,2) (0,3) (3,0)
```

**2.14.12 operator (,)**

Indexing operator

```
TC schmatrix::operator () (int im, int in) const throw (cvmexception);
```

returns value of element of calling hermitian matrix located on im-row and in-th column. Operator is *ℓ*-based. It throws `cvmexception` if some of parameters passed is outside of `[1,msize()]` range. See also `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m ((std::complex<double>*)a, 3);
std::cout << m << std::endl;
std::cout << m(3,2) << " " << m(1,3) << std::endl;
```

prints

```
(1,0) (2,-1) (-1,-2)
(2,1) (2,0) (0,-3)
(-1,2) (0,3) (3,0)

(0,3) (-1,-2)
```

**2.14.13 operator ()**

Indexing operator

```
const cvector schmatrix::operator () (int i) const throw (cvmexception);
```

provides access to *i*-th column of calling hermitian matrix. Unlike `schmatrix::operator ()`, this operator creates only *copy* of a column and therefore it returns *not l-value*. Operator is *l-based*. It throws `cvmexception` if parameter *i* is outside of `[1, nsize()]` range. See also `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m ((std::complex<double>*)a, 3);
std::cout << m << std::endl;
std::cout << m(1) << std::endl << m(3);
```

prints

```
(1,0) (2,-1) (-1,-2)
(2,1) (2,0) (0,-3)
(-1,2) (0,3) (3,0)
```

```
(1,0) (2,1) (-1,2)
```

```
(-1,-2) (0,-3) (3,0)
```

**2.14.14 operator []**

Indexing operator

```
const cvector schmatrix::operator [] (int i) const throw (cvmexception);
```

provides access to *i*-th row of calling hermitian matrix. Unlike `schmatrix::operator []`, this operator creates only *copy* of a column and therefore it returns *not l-value*. Operator is *l-based*. It throws `cvmexception` if parameter *i* is outside of `[1, nsize()]` range. See also `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m ((std::complex<double>*)a, 3);
std::cout << m << std::endl;
std::cout << m[1] << std::endl << m[3];
```

prints

```
(1,0) (2,-1) (-1,-2)
(2,1) (2,0) (0,-3)
(-1,2) (0,3) (3,0)
```

```
(1,0) (2,-1) (-1,-2)
```

```
(-1,2) (0,3) (3,0)
```

**2.14.15 diag**

Functions

```
const cvector schmatrix::diag (int i) const throw (cvmexception);
```

provide access to *i*-th diagonal of calling matrix, where *i* = 0 for main diagonal, *i* < 0 for lower diagonals and *i* > 0 for upper ones. Unlike `cmatrix::diag`, this operator creates only *copy* of a diagonal and therefore it returns *not l-value*. Function throws `cvmexception` if parameter *i* is outside of `[-msize()+1, nsize()-1]` range. See also `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m ((std::complex<double>*)a, 3);
std::cout << m << std::endl;
std::cout << m.diag(-2)
          << m.diag(-1)
          << m.diag(0)
          << m.diag(1)
          << m.diag(2);
```

prints

```
(1,0) (2,-1) (-1,-2)
(2,1) (2,0) (0,-3)
(-1,2) (0,3) (3,0)

(-1,2)
(2,1) (0,3)
(1,0) (2,0) (3,0)
(2,-1) (0,-3)
(-1,-2)
```

**2.14.16 real**

Function

```
const srsmatrix schmatrix::real () const;
```

creates object of type `const srsmatrix` as real part of calling hermitian matrix. Please note that, unlike `cvector::real`, this function creates new object *not sharing* memory with real part of calling matrix, i.e. the matrix returned is *not l-value*. See also `srsmatrix`, `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m ((std::complex<double>*)a, 3);
std::cout << m << std::endl;
std::cout << m.real();
```

prints

```
(1,0) (2,-1) (-1,-2)
(2,1) (2,0) (0,-3)
(-1,2) (0,3) (3,0)
```

```
1 2 -1
2 2 0
-1 0 3
```

### 2.14.17 imag

Function

```
const srmatrix schmatrix::imag () const;
```

creates object of type `const srmatrix` as imaginary part of calling matrix. Please note that, unlike `cvector::imag`, this function creates new object *not sharing* memory with imaginary part of calling matrix, i.e. the matrix returned is *not l-value*. See also `srmatrix`, `schmatrix`.  
Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};
```

```
schmatrix m ((std::complex<double>*)a, 3);
```

```
std::cout << m << std::endl;
```

```
std::cout << m.imag();
```

prints

```
(1,0) (2,-1) (-1,-2)
```

```
(2,1) (2,0) (0,-3)
```

```
(-1,2) (0,3) (3,0)
```

```
0 -1 -2
```

```
1 0 -3
```

```
2 3 0
```



**2.14.18 operator = (const schmatrix&)**

Operator

```
schmatrix& schmatrix::operator = (const schmatrix& m)
throw (cvmexception);
```

sets every element of calling hermitian matrix to value of appropriate element of hermitian matrix *m* and returns reference to the matrix changed. Operator throws `cvmexception` in case of different matrix sizes. See also `schmatrix`. Example:

```
using namespace cvm;

try {
    double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
                  0., 3., -1., -2., 0., -3., 3., 0.};
    schmatrix m1((std::complex<double>*)a, 3);
    schmatrix m2(3);

    m2 = m1;
    std::cout << m2;
}
catch (std::exception& e) {
    std::cout << "Exception " << e.what () << std::endl;
}
}
```

prints

```
(1,0) (2,-1) (-1,-2)
(2,1) (2,0) (0,-3)
(-1,2) (0,3) (3,0)
```

**2.14.19 assign (const TC\*)**

Function

```
schmatrix& schmatrix::assign (const cvector& v, TR tol = cvmMachSp())
throw (cvmexception);
```

```
schmatrix& schmatrix::assign (const TC* pD, TR tol = cvmMachSp())
throw (cvmexception);
```

sets every element of calling hermitian matrix to value of appropriate element of vector *v* or array pointed to by *pD* and returns reference to the matrix changed. In first version it's assumed that vector passed is long enough to fill calling matrix. Function throws **cvmexception** otherwise. Function throws **cvmexception** if the matrix changed doesn't appear to be hermitian (tolerance is set by parameter *tol*). See also **schmatrix**. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
```

```
schmatrix m(3);
m.assign((std::complex<double>*)a);
std::cout << m;
```

prints

```
(1,0) (2,-1) (-1,-2)
(2,1) (2,0) (0,-3)
(-1,2) (0,3) (3,0)
```

**2.14.20 assign (int, int, const schmatrix&)**

Function

```
schmatrix& schmatrix::assign (int nRowCol, const schmatrix& m)
throw (cvmexception);
```

sets main sub-matrix of calling hermitian matrix beginning with  $\ell$ -based row nRowCol to hermitian matrix m and returns reference to the matrix changed. Function throws `cvmexception` if nRowCol is not positive or matrix m doesn't fit. See also `schmatrix`. Example:

```
using namespace cvm;
```

```
schmatrix m1(5);
schmatrix m2(2);
m2.set_main_diag(rvector(2,2.));
m2.set(1,2,std::complex<double>(2.,2.));
m1.assign(2,m2);
std::cout << m1;
```

prints

```
(0,0) (0,0) (0,0) (0,0) (0,0)
(0,0) (2,0) (2,2) (0,0) (0,0)
(0,0) (2,-2) (2,0) (0,0) (0,0)
(0,0) (0,0) (0,0) (0,0) (0,0)
(0,0) (0,0) (0,0) (0,0) (0,0)
```

**2.14.21 set (int,int,TC)**

Function

```
schmatrix& schmatrix::set (int nRow, int nCol, TC z) throw (cvmexception);
```

sets an element located on nRow's row (*ℓ-based*) and nCol's column (*ℓ-based*) to value of parameter z and an element located on nCol's row and nRow's column to value of parameter z conjugated (keeping calling matrix to be hermitian). Function returns reference to the matrix changed. It throws *cvmexception* if any of parameters passed is outside of [1,msize()] range. It also throws in case of assigning complex number with non-zero imaginary part to any element on the main diagonal. See also *schmatrix*. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m((std::complex<double>*)a,3);
```

```
m.set(1,2,std::complex<double>(7.7,7.7));
m.set(3,3,11.11);
std::cout << m;
```

prints

```
(1,0) (7.7,7.7) (-1,-2)
(7.7,-7.7) (2,0) (0,-3)
(-1,2) (0,3) (11.11,0)
```

**2.14.22 set\_diag**

Function

```
schmatrix& schmatrix::set_diag (int i, const cvector& v)
throw (cvmexception);
```

sets  $i$ -th diagonal of calling hermitian matrix,  $i < 0$  for lower diagonals and  $i > 0$  for upper ones, to be equal to complex vector passed in parameter  $v$ . Function also sets  $-i$ -th diagonal to be equal to vector  $v$  conjugated, thus keeping calling matrix to be hermitian. Parameter  $i$  is *not*  $\ell$ -based. Function returns reference to the matrix changed and throws `cvmexception` if parameter  $i$  passed is outside of  $[-m\text{size}()+1, n\text{size}()-1]$  range or equal to zero. It also throws if vector  $v$  passed has size not equal to  $m\text{size}()-\text{abs}(i)$ . Use `schmatrix::set_main_diag` to set main diagonal. See also `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m((std::complex<double>*)a,3);
cvector v(2);
v.set(std::complex<double>(7.7,7.7));
m.set_diag(1, v);
std::cout << m;
```

prints

```
(1,0) (7.7,7.7) (-1,-2)
(7.7,-7.7) (2,0) (7.7,7.7)
(-1,2) (7.7,-7.7) (3,0)
```

### 2.14.23 set\_main\_diag

Function

```
schmatrix& schmatrix::set_main_diag (const rvector& v)
throw (cvexception);
```

sets main diagonal of calling hermitian matrix to be equal to real vector passed in parameter v. Function returns reference to the matrix changed. It throws `cvexception` if vector v passed has size not equal to `msize()`. See also `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m((std::complex<double>*)a,3);
rvector v(3);
v.set(7.7);
m.set_main_diag(v);
std::cout << m;
```

prints

```
(7.7,0) (2,-1) (-1,-2)
(2,1) (7.7,0) (0,-3)
(-1,2) (0,3) (7.7,0)
```

### 2.14.24 assign\_real

Function

```
schmatrix& schmatrix::assign_real (const srsmatrix& m)
throw (cvmexception);
```

sets real part of calling hermitian matrix to be equal to real symmetric matrix m. Function returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. See also `schmatrix`. Example:

```
using namespace cvm;

double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m((std::complex<double>*)a,3);
srsmatrix ms(3);
ms.set(7.);
m.assign_real(ms);
std::cout << m;
```

prints

```
(7,0) (7,-1) (7,-2)
(7,1) (7,0) (7,-3)
(7,2) (7,3) (7,0)
```

### 2.14.25 set\_real

Function

```
schmatrix& schmatrix::set_real (TR d);
```

sets every element of real part of calling hermitian matrix to be equal to real number d. Function returns reference to the matrix changed. See also [schmatrix](#). Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};  
schmatrix m((std::complex<double>*)a,3);  
m.set_real(7.);  
std::cout << m;
```

prints

```
(7,0) (7,-1) (7,-2)  
(7,1) (7,0) (7,-3)  
(7,2) (7,3) (7,0)
```



**2.14.26 resize**

Function

```
schmatrix& schmatrix::resize (int nNewMN) throw (cvmexception);
```

changes size of calling hermitian matrix to nNewMN by nNewMN and returns reference to the matrix changed. In case of increasing of its size, calling matrix is filled up with zeroes. Function throws `cvmexception` in case of non-positive size passed or memory allocation failure. See also `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m((std::complex<double>*)a,3);
std::cout << m << std::endl;
m.resize(4);
std::cout << m;
```

prints

```
(1,0) (2,-1) (-1,-2)
(2,1) (2,0) (0,-3)
(-1,2) (0,3) (3,0)
```

```
(1,0) (2,-1) (-1,-2) (0,0)
(2,1) (2,0) (0,-3) (0,0)
(-1,2) (0,3) (3,0) (0,0)
(0,0) (0,0) (0,0) (0,0)
```

### 2.14.27 operator ==

Operator

```
bool schmatrix::operator == (const schmatrix& m) const;
```

compares calling hermitian matrix with hermitian matrix `m` and returns true if they have the same sizes and their appropriate elements differ by not more than the **smallest normalized positive number**. Returns false otherwise. See also `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};  
schmatrix m1((std::complex<double>*)a,3);  
schmatrix m2(3);  
m2.assign((std::complex<double>*)a);  
std::cout << (m1 == m2) << std::endl;
```

prints

```
1
```

### 2.14.28 operator !=

Operator

```
bool schmatrix::operator != (const schmatrix& m) const;
```

compares calling hermitian matrix with hermitian matrix `m` and returns true if they have different sizes or at least one of their appropriate elements differs by more than the **smallest normalized positive number**. Returns false otherwise. See also `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};  
schmatrix m1((std::complex<double>*)a,3);  
schmatrix m2(3);  
m2.assign((std::complex<double>*)a);  
m2.set(2,1,std::complex<double>(2.,1.000001));  
std::cout << (m1 != m2) << std::endl;
```

prints

```
1
```

**2.14.29 operator <<**

Operator

```
schmatrix& schmatrix::operator << (const schmatrix& m)
throw (cvmexception);
```

destroys calling hermitian matrix, creates new one as a copy of m and returns reference to the matrix changed. Operator throws `cvmexception` in case of memory allocation failure. See also `schmatrix`. Example:

```
using namespace cvm;

double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m((std::complex<double>*)a,3);
schmatrix mc(1);
std::cout << m << std::endl;
std::cout << mc << std::endl;
mc << m;
std::cout << mc;
```

prints

```
(1,0) (2,-1) (-1,-2)
(2,1) (2,0) (0,-3)
(-1,2) (0,3) (3,0)
```

```
(0,0)
```

```
(1,0) (2,-1) (-1,-2)
(2,1) (2,0) (0,-3)
(-1,2) (0,3) (3,0)
```

### 2.14.30 operator +

Operator

```
schmatrix schmatrix::operator + (const schmatrix& m) const  
throw (cvmexception);
```

creates object of type `schmatrix` as sum of calling hermitian matrix and hermitian matrix `m`. It throws `cvmexception` in case of different sizes of the operands. See also `schmatrix::sum`, `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};  
double b[] = {1., 0., 1., 1., 1., 1., 1., -1., 1., 0.,  
              1., 1., 1., -1., 1., -1., 1., 0.};  
schmatrix m1((std::complex<double>*)a,3);  
schmatrix m2((std::complex<double>*)b,3);  
std::cout << m1 + m2;
```

prints

```
(2,0) (3,-2) (0,-3)  
(3,2) (3,0) (1,-4)  
(0,3) (1,4) (4,0)
```

### 2.14.31 operator -

Operator

```
schmatrix schmatrix::operator - (const schmatrix& m) const  
throw (cvmexception);
```

creates object of type `schmatrix` as difference of calling hermitian matrix and hermitian matrix `m`. It throws `cvmexception` in case of different sizes of the operands. See also `schmatrix::diff`, `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};  
double b[] = {1., 0., 1., 1., 1., 1., 1., -1., 1., 0.,  
              1., 1., 1., -1., 1., -1., 1., 0.};  
schmatrix m1((std::complex<double>*)a,3);  
schmatrix m2((std::complex<double>*)b,3);  
std::cout << m1 - m2;
```

prints

```
(0,0) (1,0) (-2,-1)  
(1,0) (1,0) (-1,-2)  
(-2,1) (-1,2) (2,0)
```

### 2.14.32 sum

Function

```
schmatrix& schmatrix::sum (const schmatrix& m1, const schmatrix& m2)
throw (cvmexception);
```

assigns result of addition of hermitian matrices m1 and m2 to calling hermitian matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. See also `schmatrix::operator +`, `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
double b[] = {1., 0., 1., 1., 1., 1., 1., -1., 1., 0.,
              1., 1., 1., -1., 1., -1., 1., 0.};
schmatrix m1((std::complex<double>*)a,3);
schmatrix m2((std::complex<double>*)b,3);
schmatrix m(3);
std::cout << m.sum(m1, m2);
```

prints

```
(2,0) (3,-2) (0,-3)
(3,2) (3,0) (1,-4)
(0,3) (1,4) (4,0)
```

### 2.14.33 diff

Function

```
schmatrix& schmatrix::diff (const schmatrix& m1, const schmatrix& m2)
throw (cvmexception);
```

assigns result of subtraction of hermitian matrices m1 and m2 to calling hermitian matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. See also `schmatrix::operator -`, `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
double b[] = {1., 0., 1., 1., 1., 1., 1., -1., 1., 0.,
              1., 1., 1., -1., 1., -1., 1., 0.};
schmatrix m1((std::complex<double>*)a,3);
schmatrix m2((std::complex<double>*)b,3);
schmatrix m(3);
std::cout << m.diff(m1, m2);
```

prints

```
(0,0) (1,0) (-2,-1)
(1,0) (1,0) (-1,-2)
(-2,1) (-1,2) (2,0)
```



**2.14.34 operator +=**

Operator

```
schmatrix& schmatrix::operator += (const schmatrix& m)
throw (cvmexception);
```

adds hermitian matrix *m* to calling hermitian matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. See also `schmatrix::operator +`, `schmatrix::sum`, `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
double b[] = {1., 0., 1., 1., 1., 1., 1., -1., 1., 0.,
              1., 1., 1., -1., 1., -1., 1., 0.};
schmatrix m1((std::complex<double>*)a,3);
schmatrix m2((std::complex<double>*)b,3);
m1 += m2;
m2 += m2;
std::cout << m1 << std::endl;
std::cout << m2;
```

prints

```
(2,0) (3,-2) (0,-3)
(3,2) (3,0) (1,-4)
(0,3) (1,4) (4,0)
```

```
(2,0) (2,-2) (2,-2)
(2,2) (2,0) (2,-2)
(2,2) (2,2) (2,0)
```

**2.14.35 operator -=**

Operator

```
schmatrix& schmatrix::operator -= (const schmatrix& m)
throw (cvmexception);
```

subtracts hermitian matrix *m* from calling hermitian matrix and returns reference to the matrix changed. It throws `cvmexception` in case of different sizes of the operands. See also `schmatrix::operator -`, `schmatrix::diff`, `schmatrix`. Example:

```
using namespace cvm;

double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
double b[] = {1., 0., 1., 1., 1., 1., 1., -1., 1., 0.,
              1., 1., 1., -1., 1., -1., 1., 0.};
schmatrix m1((std::complex<double>*)a,3);
schmatrix m2((std::complex<double>*)b,3);
m1 -= m2;
m2 -= m2;
std::cout << m1 << std::endl;
std::cout << m2;
```

prints

```
(0,0) (1,0) (-2,-1)
(1,0) (1,0) (-1,-2)
(-2,1) (-1,2) (2,0)
```

```
(0,0) (0,0) (0,0)
(0,0) (0,0) (0,0)
(0,0) (0,0) (0,0)
```

### 2.14.36 operator - ()

Operator

```
schmatrix schmatrix::operator - () const throw (cvmexception);
```

creates object of type schmatrix as calling hermitian matrix multiplied by  $-1$ . See also [schmatrix](#). Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};
```

```
schmatrix m((std::complex<double>*)a,3);
```

```
std::cout << -m;
```

prints

```
(-1,0) (-2,1) (1,2)  
(-2,-1) (-2,0) (0,3)  
(1,-2) (0,-3) (-3,0)
```

**2.14.37 operator ++**

Operator

```
schmatrix& schmatrix::operator ++ ();
schmatrix& schmatrix::operator ++ (int);
```

adds identity matrix to calling hermitian matrix and returns reference to the matrix changed. See also [schmatrix](#). Example:

```
using namespace cvm;

double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m((std::complex<double>*)a,3);
m++;
std::cout << m << std::endl;
std::cout << ++m;
```

prints

```
(2,0) (2,-1) (-1,-2)
(2,1) (3,0) (0,-3)
(-1,2) (0,3) (4,0)

(3,0) (2,-1) (-1,-2)
(2,1) (4,0) (0,-3)
(-1,2) (0,3) (5,0)
```

### 2.14.38 operator --

Operator

```
schmatrix& schmatrix::operator -- ();  
schmatrix& schmatrix::operator -- (int);
```

subtracts identity matrix from calling hermitian matrix and returns reference to the matrix changed. See also [schmatrix](#). Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};  
schmatrix m((std::complex<double>*)a,3);  
m--;  
std::cout << m << std::endl;  
std::cout << --m;
```

prints

```
(0,0) (2,-1) (-1,-2)  
(2,1) (1,0) (0,-3)  
(-1,2) (0,3) (2,0)
```

```
(-1,0) (2,-1) (-1,-2)  
(2,1) (0,0) (0,-3)  
(-1,2) (0,3) (1,0)
```

**2.14.39 operator \* (TR)**

Operator

```
schmatrix schmatrix::operator * (TR d) const;
```

creates object of type `schmatrix` as product of calling hermitian matrix and real number `d`.  
See also `schmatrix::operator *=`, `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};  
schmatrix m((std::complex<double>*)a,3);  
std::cout << m * 5.;
```

prints

```
(5,0) (10,-5) (-5,-10)  
(10,5) (10,0) (0,-15)  
(-5,10) (0,15) (15,0)
```

### 2.14.40 operator / (TR)

Operator

```
schmatrix schmatrix::operator / (TR d) const throw (cvmexception);
```

creates object of type `schmatrix` as quotient of calling hermitian matrix and real number `d`. It throws `cvmexception` if `d` has absolute value equal or less than the **smallest normalized positive number**. See also `schmatrix::operator /=` , `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};  
schmatrix m((std::complex<double>*)a,3);  
std::cout << m / 2.;
```

prints

```
(0.5,0) (1,-0.5) (-0.5,-1)  
(1,0.5) (1,0) (0,-1.5)  
(-0.5,1) (0,1.5) (1.5,0)
```

### 2.14.41 operator \* (TC)

Operator

```
schmatrix schmatrix::operator * (TC z) const;
```

creates object of type `schmatrix` as product of calling hermitian matrix and complex number `z`. See also `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};  
schmatrix m((std::complex<double>*)a,3);  
std::cout << m * std::complex<double>(1.,1.);
```

prints

```
(1,1) (3,1) (1,-3)  
(1,3) (2,2) (3,-3)  
(-3,1) (-3,3) (3,3)
```



### 2.14.42 operator / (TC)

Operator

```
schmatrix schmatrix::operator / (TC z) const throw (cvmexception);
```

creates object of type `schmatrix` as quotient of calling hermitian matrix and complex number `z`. It throws `cvmexception` if `z` has absolute value equal or less than the **smallest normalized positive number**. See also `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};  
schmatrix m((std::complex<double>*)a,3);  
std::cout << m / std::complex<double>(1.,1.);
```

prints

```
(0.5,-0.5) (0.5,-1.5) (-1.5,-0.5)  
(1.5,-0.5) (1,-1) (-1.5,-1.5)  
(0.5,1.5) (1.5,1.5) (1.5,-1.5)
```

### 2.14.43 operator \*= (TR)

Operator

```
schmatrix& schmatrix::operator *= (TR d);
```

multiplies calling hermitian matrix by real number d and returns reference to the matrix changed. See also `schmatrix::operator *`, `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};  
schmatrix m((std::complex<double>*)a,3);  
m *= 5.;  
std::cout << m;
```

prints

```
(5,0) (10,-5) (-5,-10)  
(10,5) (10,0) (0,-15)  
(-5,10) (0,15) (15,0)
```

**2.14.44 operator /= (TR)**

Operator

```
schmatrix& schmatrix::operator /= (TR d) throw (cvmexception);
```

divides calling hermitian matrix by real number d and returns reference to the matrix changed. It throws **cvmexception** if d has absolute value equal or less than the **smallest normalized positive number**. See also **schmatrix::operator /** , **schmatrix**. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};
```

```
schmatrix m((std::complex<double>*)a,3);
```

```
m /= 2.;
```

```
std::cout << m;
```

prints

```
(0.5,0) (1,-0.5) (-0.5,-1)  
(1,0.5) (1,0) (0,-1.5)  
(-0.5,1) (0,1.5) (1.5,0)
```

### 2.14.45 normalize

Function

```
schmatrix& schmatrix::normalize ();
```

normalizes calling hermitian matrix so its **Euclidean norm** becomes equal to 1 if it was greater than the **smallest normalized positive number** before the call (otherwise function does nothing). See also **schmatrix**. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);  
std::cout.precision (2);  
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};  
schmatrix m((std::complex<double>*)a,3);
```

```
m.normalize();  
std::cout << m << m.norm() << std::endl;
```

prints

```
(1.39e-001,0.00e+000) (2.77e-001,-1.39e-001) (-1.39e-001,-2.77e-001)  
(2.77e-001,1.39e-001) (2.77e-001,0.00e+000) (0.00e+000,-4.16e-001)  
(-1.39e-001,2.77e-001) (0.00e+000,4.16e-001) (4.16e-001,0.00e+000)  
1.00e+000
```

### 2.14.46 conjugation

Operator and functions

```
schmatrix schmatrix::operator ~ () const throw (cvmexception);  
schmatrix& schmatrix::conj (const schmatrix& m) throw (cvmexception);  
schmatrix& schmatrix::conj () throw (cvmexception);
```

do nothing since matrix calling is hermitian. They are provided to reimplement similar member functions and operator of the class `schmatrix`. See also `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};  
schmatrix m((std::complex<double>*)a,3);
```

```
std::cout << m - ~m;
```

prints

```
(0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0)
```

**2.14.47 transposition**

Operator and functions

```
schmatrix schmatrix::operator ! () const throw (cvmexception);
schmatrix& schmatrix::transpose (const schmatrix& m) throw (cvmexception);
schmatrix& schmatrix::transpose () throw (cvmexception);
```

implement complex hermitian matrix transposition (*not* conjugation). First operator creates object of type `schmatrix` as transposed calling matrix (it throws `cvmexception` in case of memory allocation failure). Second function sets calling matrix to be equal to matrix `m` transposed (it throws `cvmexception` in case of not appropriate sizes of the operands), third one makes it to be equal to transposed itself (it also throws `cvmexception` in case of memory allocation failure). See also `schmatrix`. Example:

```
using namespace cvm;

double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m((std::complex<double>*)a,3);
schmatrix mt(3);
std::cout << m << std::endl << !m << std::endl ;
mt.transpose(m);
std::cout << mt << std::endl;
mt.transpose();
std::cout << mt;
```

prints

```
(1,0) (2,-1) (-1,-2)
(2,1) (2,0) (0,-3)
(-1,2) (0,3) (3,0)
```

```
(1,0) (2,1) (-1,2)
(2,-1) (2,0) (0,3)
(-1,-2) (0,-3) (3,0)
```

```
(1,0) (2,1) (-1,2)
(2,-1) (2,0) (0,3)
(-1,-2) (0,-3) (3,0)
```

```
(1,0) (2,-1) (-1,-2)
(2,1) (2,0) (0,-3)
(-1,2) (0,3) (3,0)
```

**2.14.48 operator \* (const cvector&)**

Operator

```
cvector schmatrix::operator * (const cvector& v) const  
throw (cvmexception);
```

creates object of type cvector as product of calling hermitian matrix and vector v. It throws `cvmexception` if number of columns of calling matrix differs from size of vector v. Use `cvector::mult` in order to avoid new object creation. See also `schmatrix` and `cvector`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,  
              0., 3., -1., -2., 0., -3., 3., 0.};
```

```
try {  
    schmatrix m((std::complex<double>*)a,3);  
    cvector v(3);  
    v.set(std::complex<double>(1.,1.));  
  
    std::cout << m * v;  
}  
catch (std::exception& e) {  
    std::cout << "Exception: " << e.what () << std::endl;  
}
```

prints

```
(5,-1) (6,2) (-3,7)
```

**2.14.49 operator \* (const cmatrix&)**

Operator

```
cmatrix schmatrix::operator * (const cmatrix& m) const
throw (cvmexception);
```

creates object of type `cmatrix` as product of calling hermitian matrix and matrix `m`. It throws `cvmexception` if number of columns of calling matrix differs from number of rows of matrix `m`. Use `cmatrix::mult` in order to avoid new object creation. See also `cmatrix` and `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
try {
    schmatrix ms((std::complex<double>*)a,3);
    cmatrix m(3,2);
    m.set(std::complex<double>(1.,1.));

    std::cout << ms * m;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
(5,-1) (5,-1)
(6,2) (6,2)
(-3,7) (-3,7)
```



**2.14.50 operator \* (const schmatrix&)**

Operator

```
schmatrix schmatrix::operator * (const schmatrix& m) const
throw (cvmexception);
```

creates object of type `schmatrix` as product of calling hermitian matrix and matrix `m`. It throws `cvmexception` if the operands have different sizes. Use `cmatrix::mult` in order to avoid new object creation. See also `schmatrix` and `schmatrix`. Example:

```
using namespace cvm;
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
```

```
try {
    schmatrix ms((std::complex<double>*)a,3);
    schmatrix m(3);
    m.set(std::complex<double>(1.,1.));

    std::cout << ms * m;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}
```

prints

```
(5,-1) (5,-1) (5,-1)
(6,2) (6,2) (6,2)
(-3,7) (-3,7) (-3,7)
```

**2.14.51 operator / (const cvector&)**

Operator

```
cvector operator / (const cvector& vB) const throw (cvmexception);
```

returns solution  $x$  of linear equation  $A * x = b$  where calling matrix is square hermitian matrix  $A$  and vector  $b$  is passed in parameter  $vB$ . This operator throws exception of type `cvmexception` in case of inappropriate sizes of the objects or when matrix  $A$  is close to singular. See also `cvector::solve`, `schmatrix::solve`, `cvector.operator %`, `cvector`, `schmatrix`, `schmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::showpos);
std::cout.precision (2);

double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
try {
    schmatrix ma((std::complex<double>*)a,3);
    cvector vb(3);
    cvector vx(3);
    vb.set(std::complex<double>(1.,2.));

    vx = ma / vb;

    std::cout << ma * vx - vb;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}

prints
(+0.00e+000,+1.78e-015) (-8.88e-016,+3.55e-015) (-3.55e-015,-1.78e-015)
```

**2.14.52 herk**

Functions

```
schmatrix&
schmatrix::herk (TR alpha, const cvector& v, TR beta)
                throw (cvmexception);

schmatrix&
schmatrix::herk (bool bTransp, TR alpha, const cmatrix& m, TR beta)
                throw (cvmexception);
```

call ZHERK routine of the [BLAS library](#) performing matrix-vector operation defined for the first version as rank-1 update operation

$$A = \alpha \begin{pmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{pmatrix} (v_1^* \ v_2^* \ \cdots \ v_n^*) + \beta A,$$

and for the second version as

$$A = \alpha MM^H + \beta A \quad \text{or} \quad A = \alpha M^H M + \beta A.$$

Here  $\alpha$  and  $\beta$  are real numbers (parameters alpha and beta),  $M$  is complex matrix (parameter m),  $A$  is calling hermitian matrix and  $v$  is complex vector (parameter v). First operation for the second version is performed if `bTransp` passed is `false` and second one otherwise. Function returns reference to the matrix changed and throws `cvmexception` in case of inappropriate sizes of the operands. See also `cvector`, `cmatrix` and `schmatrix` Example:

```
using namespace cvm;
const treal a[] = {1., -1., 2., 2., 3., -3.};
const cvector v((std::complex<double>*)a, 3);
const treal alpha = 2.12;
const treal beta = -3.07;
schmatrix mh(3), mh2(3);
mh.randomize_real(-1.,2.);
mh.randomize_imag(-2.,3.);
mh2 = mh;
mh.herk (alpha, v, beta);
mh2 = alpha * schmatrix(v.rank1update_c(v)) + beta * mh2;
std::cout << mh - mh2;
```

prints

```
(0,0) (0,0) (0,0)
(0,0) (0,0) (0,0)
(0,0) (0,0) (0,0)
```

Example:

```
cmatrix m(3,3);
const treal alpha = 2.12;
const treal beta = -3.07;
schmatrix mh(3), mh2(3);
m.randomize_real(-1.,2.);
m.randomize_imag(-2.,3.);
mh.randomize_real(-1.,2.);
mh.randomize_imag(-2.,3.);
mh2 = mh;
mh.herk (false, alpha, m, beta);
mh2 = alpha * schmatrix (m * ~m) + beta * mh2;
std::cout << mh - mh2;
```

prints

```
(0,0) (0,0) (0,0)
(0,0) (0,0) (0,0)
(0,0) (0,0) (0,0)
```

Example:

```
cmatrix m(3,3);
const treal alpha = 2.12;
const treal beta = -3.07;
schmatrix mh(3), mh2(3);
m.randomize_real(-1.,2.);
m.randomize_imag(-2.,3.);
mh.randomize_real(-1.,2.);
mh.randomize_imag(-2.,3.);
mh2 = mh;
mh.herk (true, alpha, m, beta);
mh2 = alpha * schmatrix (~m * m) + beta * mh2;
std::cout << mh - mh2;
```

prints

```
(3.55271e-015,0) (0,0) (0,0)
(0,0) (0,0) (0,0)
(0,0) (0,0) (1.77636e-015,0)
```

**2.14.53 her2k**

Functions

```
schmatrix&
schmatrix::her2k (TC alpha, const cvector& v1,
                 const cvector& v2, TR beta) throw (cvmexception);

schmatrix&
schmatrix::her2k (bool bTransp, TC alpha, const cmatrix& m1,
                 const cmatrix& m2, TR beta) throw (cvmexception);
```

call one of ?HER2K routines of the [BLAS library](#) performing matrix-vector operation defined for the first version as rank-1 update operation

$$A = \alpha v_1 v_2^* + \alpha^* v_2 v_1^* + \beta A,$$

and for the second version as

$$A = \alpha M_1 M_2^H + \alpha^* M_2 M_1^H + \beta A \quad \text{or} \quad A = \alpha M_1^H M_2 + \alpha^* M_2^H M_1 + \beta A.$$

Here  $\alpha$  is complex and  $\beta$  is real number (parameters alpha and beta),  $M_1$  and  $M_2$  are complex matrices (parameters m1 and m2),  $A$  is calling hermitian matrix and  $v_1$  and  $v_2$  are complex vectors (parameters v1 and v2). First operation for the second version is performed if bTransp passed is false and second one otherwise. Function returns reference to the matrix changed and throws [cvmexception](#) in case of inappropriate sizes of the operands. See also [cvector](#), [cmatrix](#) and [schmatrix](#) Example:

```
using namespace cvm;
std::cout.setf(std::ios::scientific | std::ios::showpos);
std::cout.precision(2);
const tcomplex alpha(2.12,-0.14);
const tcomplex alphac(2.12,0.14);
const treal beta = -3.07;
cvector v1(3), v2(3);
schmatrix mh(3), mh2(3);
v1.randomize_real(-1.,2.);
v1.randomize_imag(-2.,3.);
v2.randomize_real(-1.,2.);
v2.randomize_imag(-2.,3.);
mh.randomize_real(-1.,2.);
mh.randomize_imag(-2.,3.);
mh2 = mh;
mh.her2k (alpha, v1, v2, beta);
mh2 = schmatrix(alpha * v1.rank1update_c(v2) +
               alphac * v2.rank1update_c(v1)) + beta * mh2;
std::cout << mh - mh2;
```

prints

```
(+1.78e-015,+0.00e+000) (+0.00e+000,+0.00e+000) (+0.00e+000,+0.00e+000)
(+0.00e+000,+0.00e+000) (-1.78e-015,+0.00e+000) (+0.00e+000,+1.78e-015)
(+0.00e+000,+0.00e+000) (+0.00e+000,-1.78e-015) (+4.44e-016,+0.00e+000)
```

Example:

```
using namespace cvm;
std::cout.setf(std::ios::scientific | std::ios::showpos);
std::cout.precision(2);
const tcomplex alpha(2.12,-0.14);
const tcomplex alphac(2.12,0.14);
const treal beta = -3.07;
cmatrix m1(3,3), m2(3,3);
schmatrix mh(3), mh2(3);
m1.randomize_real(-1.,2.);
m1.randomize_imag(-2.,3.);
m2.randomize_real(-1.,2.);
m2.randomize_imag(-2.,3.);
mh.randomize_real(-1.,2.);
mh.randomize_imag(-2.,3.);
mh2 = mh;
mh.her2k (false, alpha, m1, m2, beta);
mh2 = schmatrix(alpha * m1 * ~m2 +
                alphac * m2 * ~m1, 1.e-14) + beta * mh2;
std::cout << mh - mh2;
```

prints

```
(+0.00e+000,+4.44e-016) (+0.00e+000,+0.00e+000) (+0.00e+000,+1.78e-015)
(+3.55e-015,+0.00e+000) (+4.44e-016,-8.88e-016) (+0.00e+000,+0.00e+000)
(+8.88e-016,+8.88e-016) (+0.00e+000,+0.00e+000) (+8.88e-016,+1.78e-015)
```

Example:

```
using namespace cvm;
std::cout.setf(std::ios::scientific | std::ios::showpos);
std::cout.precision(2);
const tcomplex alpha(2.12,-0.14);
const tcomplex alphac(2.12,0.14);
const treal beta = -3.07;
cmatrix m1(3,3), m2(3,3);
schmatrix mh(3), mh2(3);
m1.randomize_real(-1.,2.);
```

```
m1.randomize_imag(-2.,3.);
m2.randomize_real(-1.,2.);
m2.randomize_imag(-2.,3.);
mh.randomize_real(-1.,2.);
mh.randomize_imag(-2.,3.);
mh2 = mh;
mh.her2k (true, alpha, m1, m2, beta);
mh2 = schmatrix(alpha * ~m1 * m2 +
                alphac * ~m2 * m1, 1.e-14) + beta * mh2;
std::cout << mh - mh2;
```

prints

```
(+3.55e-015,-8.88e-016) (+0.00e+000,+0.00e+000) (+0.00e+000,-1.78e-015)
(-8.88e-016,+1.78e-015) (+0.00e+000,-4.44e-016) (+8.88e-016,+0.00e+000)
(+0.00e+000,+3.55e-015) (-1.33e-015,-1.78e-015) (+0.00e+000,+0.00e+000)
```

**2.14.54 inv**

Functions

```
schmatrix& schmatrix::inv (const schmatrix& m) throw (cvmexception);
schmatrix schmatrix::inv () const throw (cvmexception);
```

implement hermitian matrix inversion. The first version sets calling hermitian matrix to be equal to hermitian matrix *m* inverted and the second one creates object of type *schmatrix* as inverted calling matrix. Functions throw *cvmexception* in case of inappropriate sizes of the operands or when matrix to be inverted is close to singular. See also *schmatrix*. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
```

```
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
```

```
schmatrix m((std::complex<double>*)a,3);
const schmatrix mi = m.inv();
```

```
std::cout << mi << std::endl;
std::cout << mi * m - eye_complex(3);
```

prints

```
(-1.50e+000,0.00e+000) (-1.67e-016,8.33e-017) (-5.00e-001,-1.00e+000)
(-1.67e-016,-8.33e-017) (-1.00e+000,0.00e+000) (0.00e+000,-1.00e+000)
(-5.00e-001,1.00e+000) (0.00e+000,1.00e+000) (-1.50e+000,0.00e+000)
```

```
(2.22e-016,0.00e+000) (1.11e-016,0.00e+000) (2.78e-017,0.00e+000)
(2.78e-016,2.22e-016) (4.44e-016,0.00e+000) (0.00e+000,-4.44e-016)
(2.22e-016,-4.44e-016) (0.00e+000,-8.88e-016) (-2.22e-016,0.00e+000)
```



**2.14.55 exp**

Functions

```
schmatrix& schmatrix::exp (const schmatrix& m, TR tol = cvmMachSp ())
throw (cvmexception);
```

```
schmatrix schmatrix::exp (TR tol = cvmMachSp ()) const
throw (cvmexception);
```

compute exponent of calling hermitian matrix using Padé approximation defined as

$$R_{pq}(z) = D_{pq}(z)^{-1}N_{pq}(z) = 1 + z + \dots + z^p/p!,$$

where

$$N_{pq}(z) = \sum_{k=0}^p \frac{(p+q-k)!p!}{(p+q)!k!(q-k)!} z^k,$$

$$D_{pq}(z) = \sum_{k=0}^q \frac{(p+q-k)!p!}{(p+q)!k!(q-k)!} (-z)^k$$

along with matrix normalizing as described in [2], p. 572. Functions use ZMEXP (or CMEXP for float version) FORTRAN subroutine implementing the algorithm. The first version sets calling hermitian matrix to be equal to exponent of hermitian matrix *m* and returns reference to the matrix changed. The second version creates object of type *schmatrix* as exponent of calling matrix. The algorithm uses parameter *tol* as  $\varepsilon(p, q)$  in order to choose constants *p* and *q* so that

$$\varepsilon(p, q) \geq 2^{3-(p+q)} \frac{p!q!}{(p+q)!(p+q+1)!}.$$

This parameter is equal to the **largest relative spacing** by default. Functions throw **cvmexception** in case of inappropriate sizes of the operands or when LAPACK subroutine fails. See also *schmatrix*. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left |
                std::ios::showpos);
std::cout.precision (15);
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m((std::complex<double>*)a,3);
schmatrix me(3);
me.exp(m);
```

```

std::cout << "Column 1" << std::endl
<< me(1,1) << std::endl << me(2,1) << std::endl << me(3,1) << std::endl
    << "Column 2" << std::endl
<< me(1,2) << std::endl << me(2,2) << std::endl << me(3,2) << std::endl
    << "Column 3" << std::endl
<< me(1,3) << std::endl << me(2,3) << std::endl << me(3,3) << std::endl;

```

prints

```

Column 1
(+2.673228708372002e+002,+1.091141066389412e-014)
(+3.071187567026803e+002,+1.535593783513402e+002)
(-1.749365628720766e+002,+3.498731257441531e+002)
Column 2
(+3.071187567026803e+002,-1.535593783513401e+002)
(+4.422594337092766e+002,+1.919736460939478e-015)
(-9.600094996571151e-015,+5.034325040954932e+002)
Column 3
(-1.749365628720765e+002,-3.498731257441531e+002)
(+6.184072406183948e-015,-5.034325040954932e+002)
(+5.744416275398805e+002,+1.540673883337074e-014)

```

Matlab output:

Column 1

```

2.673228708371998e+002 -7.105427357601002e-015i
3.071187567026802e+002 +1.535593783513401e+002i
-1.749365628720764e+002 +3.498731257441527e+002i

```

Column 2

```

3.071187567026802e+002 -1.535593783513401e+002i
4.422594337092769e+002 -5.489286670342458e-016i
3.549798266275454e-015 +5.034325040954932e+002i

```

Column 3

```

-1.749365628720763e+002 -3.498731257441526e+002i
-1.776065298147746e-014 -5.034325040954931e+002i
5.744416275398801e+002 -2.096383162906490e-014i

```

**2.14.56 polynomial**

Functions

```
schmatrix& schmatrix::polynom (const schmatrix& m, const rvector& v)
throw (cvmexception);
```

```
schmatrix schmatrix::polynom (const rvector& v) const
throw (cvmexception);
```

compute hermitian matrix polynomial defined as

$$p(A) = b_0I + b_1A + \dots + b_qA^q$$

using the Horner's rule:

$$p(A) = \sum_{k=0}^r B_k(A^s)^k, \quad s = \text{floor}(\sqrt{q}), \quad r = \text{floor}(q/s)$$

where

$$B_k = \begin{cases} \sum_{i=0}^{s-1} b_{sk+i} A^i, & k = 0, 1, \dots, r-1 \\ \sum_{i=0}^{q-sr} b_{sr+i} A^i, & k = r. \end{cases}$$

See also [2], p. 568. *Real* coefficients  $b_0, b_1, \dots, b_q$  are passed in parameter  $v$ , where  $q$  is equal to  $v.size()-1$ , so functions compute matrix polynomial equal to

$$v[1] * I + v[2] * m + \dots + v[v.size()] * m^{v.size()-1}$$

The first version sets calling hermitian matrix to be equal to the polynomial of hermitian matrix  $m$  and the second one creates object of type `schmatrix` as the polynomial of calling symmetric matrix. Functions use ZPOLY (or CPOLY for float version) FORTRAN subroutine implementing the Horner's algorithm. Functions throw `cvmexception` in case of inappropriate sizes of the operands. See also `schmatrix`. Example:

```
using namespace cvm;
std::cout.setf (std::ios::scientific | std::ios::left |
               std::ios::showpos);
std::cout.precision (10);
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m((std::complex<double>*)a,3);
double re[]={2.2,1.3,1.1,-0.9,0.2,-0.45,45.,-30.,10.,3.,1.13};
const rvector vr(re, 11);
schmatrix mp(3);
```

```

mp.polynom (m, vr);

std::cout << "Column 1" << std::endl
<< mp(1,1) << std::endl << mp(2,1) << std::endl << mp(3,1) << std::endl
    << "Column 2" << std::endl
<< mp(1,2) << std::endl << mp(2,2) << std::endl << mp(3,2) << std::endl
    << "Column 3" << std::endl
<< mp(1,3) << std::endl << mp(2,3) << std::endl << mp(3,3) << std::endl;

```

prints

```

Column 1
(+1.2319548758e+008,+0.0000000000e+000)
(+1.4179323916e+008,+7.0896619580e+007)
(-8.0802738460e+007,+1.6160547692e+008)
Column 2
(+1.4179323916e+008,-7.0896619580e+007)
(+2.0399822604e+008,+0.0000000000e+000)
(+0.0000000000e+000,+2.3250209650e+008)
Column 3
(-8.0802738460e+007,-1.6160547692e+008)
(+0.0000000000e+000,-2.3250209650e+008)
(+2.6498872674e+008,+0.0000000000e+000)

```

Matlab output:

```

Column 1

1.231954875800000e+008
1.417932391600000e+008 +7.089661958000000e+007i
-8.080273845999999e+007 +1.616054769200000e+008i

Column 2

1.417932391600000e+008 -7.089661958000000e+007i
2.039982260400000e+008
0 +2.325020965000000e+008i

Column 3

-8.080273845999999e+007 -1.616054769200000e+008i
0 -2.325020965000000e+008i
2.649887267400000e+008

```

**2.14.57 eig**

Functions

```
rvector schmatrix::eig (schmatrix& mEigVect) const throw (cvmexception);
rvector schmatrix::eig () const throw (cvmexception);
```

solve the **symmetric eigenvalue problem** and return real vector with eigenvalues of calling hermitian matrix. The first version sets output parameter mEigVect to be equal to square matrix containing orthogonal eigenvectors as columns. All functions throw **cvmexception** in case of inappropriate sizes of the operands or in case of convergence error. See also **rvector**, **schmatrix** and **schmatrix**. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::left);
std::cout.precision (2);
double a[] = {1., 0., 2., 1., -1., 2., 2., -1., 2., 0.,
              0., 3., -1., -2., 0., -3., 3., 0.};
schmatrix m((std::complex<double>*)a,3);
schmatrix me(3);
rvector v(3);
```

```
v = m.eig(me);
std::cout << v << std::endl;
cvector vc(v);
```

```
std::cout << m * me(1) - me(1) * vc(1);
std::cout << m * me(2) - me(2) * vc(2);
std::cout << m * me(3) - me(3) * vc(3);
// orthogonality check:
std::cout << me(1) % me(2) << std::endl;
std::cout << me(2) % me(3) << std::endl;
std::cout << me(1) % me(3) << std::endl;
```

prints

```
-8.13e-001 -3.44e-001 7.16e+000
```

```
(1.39e-016,2.22e-016) (5.25e-017,-1.11e-016) (1.94e-016,1.67e-016)
(4.86e-016,4.44e-016) (7.63e-017,0.00e+000) (3.33e-016,2.22e-016)
(-2.22e-016,-8.88e-016) (-5.55e-017,-8.88e-016) (8.88e-016,-5.55e-017)
(-5.17e-017,-9.74e-017)
(-5.81e-017,-5.40e-017)
(2.37e-018,-3.56e-017)
```

**2.14.58 Cholesky**

Function

```
schmatrix schmatrix::cholesky () const throw (cvmexception);
```

forms the Cholesky factorization of hermitian positive-definite matrix A defined as

$$A = U^H U,$$

where U is upper triangular matrix. It utilizes one of ?POTRF routines of the [LAPACK library](#). Function creates object of type schmatrix as factorization of calling matrix. It throws `cvexception` in case of convergence error. See also `schmatrix` and `schmatrix`. Example:

```
using namespace cvm;

try {
    double a[] = {3., 0., 2., 1., -1., 2., 2., -1., 3., 0.,
                 0., 3., -1., -2., 0., -3., 5., 0.};
    const schmatrix m((std::complex<double>*)a,3);

    schmatrix h = m.cholesky();
    std::cout << h << std::endl;
    std::cout << ~h * h - m;
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}

prints

(1.73205,0) (1.1547,-0.57735) (-0.57735,-1.1547)
(0,0) (1.1547,0) (0,-1.1547)
(0,0) (0,0) (1.41421,0)

(-4.44089e-016,0) (0,0) (0,0)
(0,0) (0,0) (0,0)
(0,0) (0,0) (0,0)
```

### 2.14.59 Bunch-Kaufman

Function

```
scmatrix schmatrix::bunch_kaufman () throw (cvmexception);
```

forms the Bunch-Kaufman factorization of calling hermitian matrix (cited from the MKL library documentation):

$$A = PUDU^H P^H,$$

where  $A$  is calling matrix,  $P$  is permutation matrix,  $U$  and  $L$  are upper and lower triangular matrices with unit diagonal, and  $D$  is symmetric block-diagonal matrix with 1-by-1 and 2-by-2 diagonal blocks.  $U$  and  $L$  have 2-by-2 unit diagonal blocks corresponding to the 2-by-2 blocks of  $D$ . It utilizes one of ?HETRF routines of the [LAPACK library](#). Function creates object of type `scmatrix` as factorization of calling matrix. It throws `cvmexception` in case of convergence error. See also `scmatrix` and `schmatrix`. Function is mostly designed to be used for subsequent calls of ?HETRS, ?HECON and ?HETRI routines of the [LAPACK library](#). Currently it's used internally in `scmatrix::det` flow when argument is hermitian but not positive-definite matrix.

### 2.14.60 identity

Function

```
schmatrix& schmatrix::identity();
```

sets calling hermitian matrix to be equal to identity matrix and returns reference to the matrix changed. See also [schmatrix](#). Example:

```
using namespace cvm;
```

```
schmatrix m(3);  
m.randomize_real(0.,3.);  
m.randomize_imag(-1.,2.);
```

```
std::cout << m << std::endl;  
std::cout << m.identity();
```

prints

```
(1.93548,0) (1.84027,1.08353) (0.429579,-0.614093)  
(1.84027,-1.08353) (1.76922,0) (1.71364,1.82788)  
(0.429579,0.614093) (1.71364,-1.82788) (0.824915,0)
```

```
(1,0) (0,0) (0,0)  
(0,0) (1,0) (0,0)  
(0,0) (0,0) (1,0)
```



### 2.14.61 vanish

Function

```
schmatrix& schmatrix::vanish();
```

sets every element of calling hermitian matrix to be equal to zero and returns reference to the matrix changed. See also [schmatrix](#). Example:

```
using namespace cvm;
```

```
schmatrix m(3);  
m.randomize_real(0.,3.);  
m.randomize_imag(-1.,2.);
```

```
std::cout << m << std::endl;  
std::cout << m.vanish();
```

prints

```
(1.95499,0) (1.03925,0.830378) (0.951628,0.563677)  
(1.03925,-0.830378) (0.150426,0) (2.29365,-0.580218)  
(0.951628,-0.563677) (2.29365,0.580218) (0.0841395,0)
```

```
(0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0)  
(0,0) (0,0) (0,0)
```

### 2.14.62 `randomize_real`

Function

```
schmatrix& schmatrix::randomize_real (TR dFrom, TR dTo);
```

fills real part of calling hermitian matrix with pseudo-random numbers distributed between `dFrom` and `dTo` keeping it to be hermitian. Function returns reference to the matrix changed. See also [schmatrix](#). Example:

```
using namespace cvm;
```

```
schmatrix m(3);  
m.randomize_real(0.,3.);  
std::cout << m << std::endl;
```

prints

```
(1.98245,0) (2.72103,0) (0.167272,0)  
(2.72103,0) (0.0285653,0) (1.63765,0)  
(0.167272,0) (1.63765,0) (1.15882,0)
```

### 2.14.63 randomize\_imag

Function

```
schmatrix& schmatrix::randomize_imag (TR dFrom, TR dTo);
```

fills imaginary part of calling hermitian matrix with pseudo-random numbers distributed between dFrom and dTo keeping it to be hermitian. Function returns reference to the matrix changed. See also [schmatrix](#). Example:

```
using namespace cvm;
```

```
schmatrix m(3);  
m.randomize_imag(0.,3.);  
std::cout << m << std::endl;
```

prints

```
(0,0) (0,0.13834) (0,2.39903)  
(0,-0.13834) (0,0) (0,0.609577)  
(0,-2.39903) (0,-0.609577) (0,0)
```

## 2.15 **cvmexception**

The CVM library exceptions class.

```
class cvmexception : public std::exception
{
public:
    explicit cvmexception (int nCause, ...);
    int cause () const;
    virtual const char* what () const;
    static int getNextCause ();
    static bool add (int nNewCause, const char* szNewMessage);
};
```

### 2.15.1 **cause**

Function

```
int cvmexception::cause () const;
```

returns a numeric code of an exception thrown. Possible codes can be found in `cvm.h` file. See also [cvmexception](#). Example:

```
using namespace cvm;
```

```
try {  
    rvector v(10);  
    v[11] = 1.;  
}  
catch (cvmexception& e) {  
    std::cout << "Exception " << e.cause () << ": "  
              << e.what () << std::endl;  
}
```

prints

```
Exception 2: Out of range
```

### 2.15.2 **what**

Function

```
virtual const char* cvmexception::what () const throw();
```

returns a string describing an exception happened. This function overrides `std::exception::what()`. This allows you to catch just one type of exception in your application. See also [cvmexception](#). Example:

```
using namespace cvm;
```

```
try {  
    double a[] = {1., 2., 1., 2.};  
    const srsmatrix m(a, 2);  
    std::cout << m;  
}  
catch (std::exception& e) {  
    std::cout << "Exception: " << e.what () << std::endl;  
}
```

prints

```
Exception: The matrix passed doesn't appear to be symmetric
```

### 2.15.3 Customization

Constructor and functions

```
explicit cvmexception (int nCause, ...);
static bool cvmexception::add (int nNewCause, const char* szNewMessage);
static int cvmexception::getNextCause ();
```

allow to add and use customized exception codes and messages. See also [cvmexception](#).  
Example:

```
using namespace cvm;

const int nNextCause = cvmexception::getNextCause();
cvmexception::add (nNextCause,
                  "My first exception with %d parameter");
cvmexception::add (nNextCause + 1,
                  "My second exception with %s parameter");

try {
    throw cvmexception (nNextCause, 1234);
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}

try {
    throw cvmexception (nNextCause + 1, "Hi!");
}
catch (std::exception& e) {
    std::cout << "Exception: " << e.what () << std::endl;
}

prints
```

```
Exception: My first exception with 1234 parameter
Exception: My second exception with Hi! parameter
```

## 2.16 Utilities

These functions have `cvm` namespace scope and can be used for different purposes.

```
template <typename T>
T* cvmMalloc (size_t nEls) throw (cvmexception);
template <typename T>
T* cvmAddRef (const T* pD);
template <typename T>
int cvmFree (T*& pD);
void cvmExit ();
treal cvmMachMin ();
treal cvmMachSp ();
srmatrix eye_real (int nM);
scmatrix eye_complex (int nM);
operator * (,);
```



### 2.16.1 cvmMalloc

Since version 6.0 if CVM\_USE\_POOL\_MANAGER macro is not defined (this is the default) this function simply calls `::new T[nEls]` operator.

```
template <typename T>
T* cvmMalloc (size_t nEls) throw (cvmexception);
```

allocates nEls units of type T from the CVM library's **memory pool** and returns a pointer to the memory allocated. See also [cvmAddRef](#) and [cvmFree](#). Example:

```
using namespace cvm;
```

```
double* p = cvmMalloc<double> (10);
p[0] = 1.;
p[1] = 2.;
p[2] = 3.;
```

```
rvector v(3);
v.assign(p);
std::cout << v;
```

```
cvmFree (p);
```

```
prints
```

```
1 2 3
```

### 2.16.2 `cvmAddRef`

Since version 6.0 it's available only if `CVM_USE_POOL_MANAGER` is defined.

```
template <typename T>  
void cvmAddRef (const T* pD);
```

increments a reference counter for a memory block pointed to by `pD` if this block was allocated from the CVM library's memory pool (using `cvmMalloc` function). If `pD` points to a foreign memory block then the function does nothing. See also `cvmAddRef`. Example:

```
using namespace cvm;
```

```
double* p = cvmMalloc<double> (10);  
p[0] = 1.;  
p[1] = 2.;  
p[2] = 3.;
```

```
cvmAddRef (p);  
cvmFree (p);    // this call doesn't deallocate a memory  
cvmFree (p);    // this one does
```

### 2.16.3 cvmFree

Since version 6.0 if CVM\_USE\_POOL\_MANAGER macro is not defined (this is the default) then it simply calls `::delete[] pD` operator if `pD!=NULL` and returns zero.

```
template <typename T>
int cvmFree (T*& pD);
```

decrements a reference counter for a memory block pointed to by `pD` if this block was allocated from the CVM library's memory pool (using `cvmMalloc` function) and returns the reference counter it changed. If the function returns zero then it sets the pointer `pD` to be equal to `NULL` and "frees" the memory, i.e. returns the memory block to a list of free ones (see [CVM memory management](#) for details). If `pD` points to a foreign memory block then the function does nothing and returns `-1`. See also `cvmAddRef`. Example:

```
using namespace cvm;

double* pf = new double[10];
double* p = cvmMalloc<double> (10);

cvmAddRef (p);

std::cout << cvmFree (p) << " ";
std::cout << p << std::endl;

std::cout << cvmFree (p) << " ";
std::cout << p << std::endl;

std::cout << cvmFree (pf) << " ";
std::cout << pf << std::endl;

delete[] pf;

prints

1 003C66B0
0 00000000
-1 003C7A40
```

### 2.16.4 cvmExit

Since version 6.0 it's available only if `CVM_USE_POOL_MANAGER` is defined.

```
void cvmExit ();
```

destroys the CVM library's **memory pool** if `CVM_USE_POOL_MANAGER` is defined. Otherwise does nothing. All objects created using this pool are not accessible after calling of this function. Call this function in the last expression only if you have problems with memory deallocation while finishing execution of your program (I was not able to experience such problems, this function is provided just in case). See also `cvmMalloc`. Example:

```
using namespace cvm;
```

```
int main(int argc, char* argv[])
    try {
        rvector v(3);
        v[1] = v[2] = v[3] = 1.;
        std::cout << v.norm2() << std::endl;
    }
    catch (cvmexception& e) {
        std::cout << "Exception " << e.cause ()
            << ": " << e.what () << std::endl;
    }

    cvmExit ();
    return 1;
}
```

prints

```
1.73205
```

### 2.16.5 cvmMachMin

Function

```
treal cvmMachMin ();
```

returns the smallest normalized positive number, i.e. `numeric_limits<treal>::min()` where `treal` is typedef'ed as `double` by default or as `float` for float version of the library. See also [cvmMachSp](#). Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::showpos);  
std::cout.precision (15);  
std::cout << cvmMachMin() << std::endl;
```

on Intel Pentium® III machine prints

```
+2.225073858507201e-308
```

### 2.16.6 `cvmMachSp`

Function

```
treal cvmMachSp ();
```

returns the largest relative spacing or, in other words, the difference between 1 and the least value greater than 1 that is representable, i.e. `numeric_limits<treal>::epsilon()` where `treal` is typedef'ed as `double` by default or as `float` for float version of the library. See also `cvmMachMin`. Example:

```
using namespace cvm;
```

```
std::cout.setf (std::ios::scientific | std::ios::showpos);  
std::cout.precision (15);  
std::cout << cvmMachSp() << std::endl;
```

on Intel Pentium® III machine prints

```
+2.220446049250313e-016
```

### 2.16.7 eye\_real

Function

```
srmatrix eye_real (int nM);
```

creates a  $nM$  by  $nM$  object of type `srmatrix` equal to identity matrix. See also `srmatrix`.

Example:

```
using namespace cvm;
```

```
std::cout << eye_real (4);
```

prints

```
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

### 2.16.8 eye\_complex

Function

```
scmatrix eye_complex (int nM);
```

creates a nM by nM object of type scmatrix equal to identity matrix. See also [scmatrix](#).

Example:

```
using namespace cvm;
```

```
std::cout << eye_complex (4);
```

prints

```
(1,0) (0,0) (0,0) (0,0)
(0,0) (1,0) (0,0) (0,0)
(0,0) (0,0) (1,0) (0,0)
(0,0) (0,0) (0,0) (1,0)
```



**2.16.9 operator \***

Operators

```

inline rvector operator * (TR d, const rvector& v);
inline rmatrix operator * (TR d, const rmatrix& m);
inline srmatrix operator * (TR d, const srmatrix& m);
inline srbmatrix operator * (TR d, const srbmatrix& m);
inline srsrmatrix operator * (TR d, const srsrmatrix& m);
inline cvector operator * (TR d, const cvector& v);
inline cmatrix operator * (TR d, const cmatrix& m);
inline scmatrix operator * (TR d, const scmatrix& m);
inline scbmatrix operator * (TR d, const scbmatrix& m);
inline schmatrix operator * (TR d, const schmatrix& m);
inline cvector operator * (std::complex<TR> c, const cvector& v);
inline cmatrix operator * (std::complex<TR> c, const cmatrix& m);
inline scmatrix operator * (std::complex<TR> c, const scmatrix& m);
inline scbmatrix operator * (std::complex<TR> c, const scbmatrix& m);
inline schmatrix operator * (std::complex<TR> c, const schmatrix& m);
inline rvector operator * (CVM_LONGEST_INT d, const rvector& v);
inline rmatrix operator * (CVM_LONGEST_INT d, const rmatrix& m);
inline srmatrix operator * (CVM_LONGEST_INT d, const srmatrix& m);
inline srbmatrix operator * (CVM_LONGEST_INT d, const srbmatrix& m);
inline srsrmatrix operator * (CVM_LONGEST_INT d, const srsrmatrix& m);
inline cvector operator * (CVM_LONGEST_INT d, const cvector& v);
inline cmatrix operator * (CVM_LONGEST_INT d, const cmatrix& m);
inline scmatrix operator * (CVM_LONGEST_INT d, const scmatrix& m);
inline scbmatrix operator * (CVM_LONGEST_INT d, const scbmatrix& m);
inline schmatrix operator * (CVM_LONGEST_INT d, const schmatrix& m);

```

provide an ability to make left-sided multiplication of numbers by different CVM objects.

Example:

```

using namespace cvm;

const schmatrix scm = eye_complex (4);
std::cout << std::complex<double>(2.,1.) * scm << std::endl;

rvector v(3);
v(1) = 1.;
v(2) = 2.;
v(3) = 3.;

std::cout << 3. * v;

```

prints

```
(2,1) (0,0) (0,0) (0,0)
(0,0) (2,1) (0,0) (0,0)
(0,0) (0,0) (2,1) (0,0)
(0,0) (0,0) (0,0) (2,1)
```

3 6 9

## 2.17 Pool Manager

*The memory management mechanism described below is no longer supported by default. It was a good solution more than 10 years ago when memory allocation operator was relatively expensive, but now standard allocator does this job much faster. However, the algorithm described below may be useful as an error detection helper. You will need to rebuild the library in debug mode with CVM\_USE\_POOL\_MANAGER defined in order to use it.*

Since its third release the CVM library implements nontrivial memory management which should be described in detail. Earlier a memory was being allocated using operator `new` in every constructor, and freed with help of `delete` in every destructor. Let us consider an operation of multiplying of vector `a` by matrix `A` and assignment of result to vector `b`:

```
b = a * A;
```

This harmless code calls two constructors (a constructor allocating memory for output and a copy constructor, returning output to a calling function), as well as two destructors deleting those temporary objects<sup>11</sup>. If sizes of the objects are relatively small, your processor will be longer allocating memory than multiplying<sup>12</sup>. The idea of nontrivial memory management came from Jeff Alger's book [1]. Author suggests some approaches to memory allocation (overloading of operators `new` and `delete` or implementation of a class controlling a pool), and also some ways of a pool compaction (Baker's algorithm and in-place compaction) and references counting (using master or "genius" pointers). I decided to implement a class controlling a pool in the CVM library.

Pool is controlled by `MemoryPool` class. It allocates a memory by blocks (so-called "outer blocks"). The memory block concept is implemented in a class `MemoryBlocks`. Size of an outer block equals to the nearest degree of 2 of a requested number of bytes multiplied by two. This can be illustrated on the following example. Let us suppose that it's required to allocate a memory for storage of a vector consisting of 1000 units:

```
rvector v(1000);
```

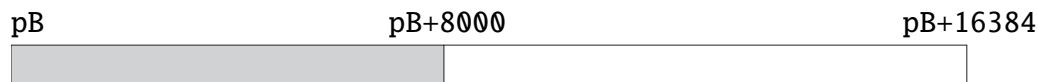
If at the moment of execution of this statement there is no free block of size 8000 or greater in a list of free memory blocks, then the following (simplified) code will be executed:

```
const int nUpBytes = up_value (nBytes);
const int nRest    = nUpBytes - nBytes;
try {
    pB = ::new tbyte[nUpBytes];
}
catch (const std::bad_alloc&) {
    throw (cvmexception (CVM_OUTOFMEMORY));
}
m_lOutBlocks.push_back (pB);
m_blocks.AddPair (pB, nBytes, nRest);
```

<sup>11</sup> In order to avoid those memory allocations you can use `b.mult(a,A)`;

<sup>12</sup> At least under Win32

where variable `nBytes` has value of 8000 ( $1000 * \text{sizeof}(treal)$ ), and function `up_value` returns the nearest degree of two multiplied by two, i.e. 16384. Thus, the allocated outer block can be represented as follows:

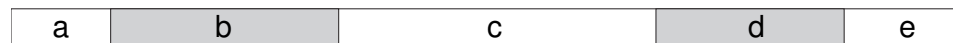


The block used for storage 1000 units of a vector `v` (its start address here is stored in the pointer `pB`) is filled. The remaining block (named as “free block”) consists of `nRest=8384` bytes. Further, if application needs one more block of the same size (it happens in most cases while execution of a copy constructor), the memory will be allocated from this free block without calling of operator `new`. The result will be the following:



Remaining free block of 384 bytes will be utilized for memory allocation of small objects. In case of creation of an object of size greater than 384 bytes one more outer block will be created, etc. While using of this scheme sooner or later memory begins to be like a sieve of free and occupied blocks. To avoid this chaos, I have applied the logic of blocks releasing, which acts like an algorithm of free space compaction. Difference of this logic from the algorithms of compaction described in [1] is that occupied blocks are never being moved.

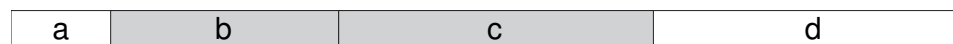
Let's say one of outer blocks looks like



And the block `d` is going to be released. In this case neighbor blocks `c` and `e` will be checked and if one of them (or both, as in this case) will appear as free, then it will be joined with the block being released. The result of release of the block `d` will be the following:



And if the following object to be created will not find a room in the block `a`, but find a room in the block `c`, the result of memory allocation will be the following:



So, there is no any chaos.

## References

- [1] *Jeff Alger*. C++ for Real Programmers, Morgan Kaufmann Publishers, 1998, 388 p., ISBN 0120499428
- [2] *Gene H. Golub, Charles F. Van Loan*. Matrix Computations, The Johns Hopkins University Press, 1996, 694 p., ISBN 0-8018-5413-X
- [3] *Peter Lancaster*. Theory of Matrices, Academic Press, New York, 1969
- [4] *Scott Meyers*. More Effective C++: 35 new ways to improve your programs and designs, Addison-Wesley, 1996, 320 p., ISBN 0-201-63371-X
- [5] *Roger A. Horn, Charles R. Johnson*. Matrix Analysis, Cambridge University Press, 1990, 561 p., ISBN 0-521-38632-2
- [6] Ф.Р. Гантмахер Теория Матриц, Москва, Наука, 1988, 522 стр., ISBN 5-02-013722-7

## Contents

<b>1</b>	<b>Preface</b>	<b>1</b>
1.1	Brief history . . . . .	1
1.2	Libraries Supported . . . . .	1
1.3	Object Model . . . . .	2
1.4	Installation . . . . .	3
1.4.1	Directory Structure . . . . .	3
1.4.2	Usage Notes . . . . .	4
1.4.3	Installation – Win32 . . . . .	4
1.4.4	Installation – Unix and MinGW . . . . .	4
1.5	Storage . . . . .	5
1.6	Indexing . . . . .	6
1.7	Polymorphism . . . . .	7
1.8	Multi-threading . . . . .	7
1.9	Regression test utility . . . . .	7
<b>2</b>	<b>CVM Class Library Reference</b>	<b>8</b>
2.1	basic_array . . . . .	8
2.1.1	basic_array() . . . . .	10
2.1.2	basic_array(int, bool) . . . . .	11
2.1.3	basic_array(T*, int) . . . . .	12
2.1.4	basic_array(const T*, int) . . . . .	13
2.1.5	basic_array(const T*, const T*) . . . . .	14
2.1.6	basic_array(const basic_array&) . . . . .	15
2.1.7	size() . . . . .	16
2.1.8	get(), operator T*() . . . . .	17
2.1.9	Indexing operators . . . . .	18
2.1.10	operator = (const basic_array&) . . . . .	19
2.1.11	assign(const T*) . . . . .	20
2.1.12	set(T) . . . . .	21
2.1.13	resize . . . . .	22
2.1.14	STL-specific type definitions . . . . .	23
2.1.15	STL-specific functions: begin(), end(), rbegin(), rend(), max_size(), capacity(), empty(), front(), back(), assign(), resize(), clear(), swap() . . . . .	25
2.1.16	at() . . . . .	27
2.1.17	push_back(const T&), pop_back() . . . . .	28
2.1.18	insert (iterator, const T&), erase (iterator) . . . . .	29
2.1.19	operator >> << (std::istream& is, basic_array<T>& aIn) . . . . .	30
2.1.20	operator << << (std::ostream& os, const basic_array<T>& aOut) . . . . .	31
2.2	Array . . . . .	32

2.2.1	incr	33
2.2.2	indofmax	34
2.2.3	indofmin	35
2.2.4	norm	36
2.2.5	norminf	37
2.2.6	norm1	38
2.2.7	norm2	39
2.2.8	operator >> <> (std::istream& is, Array<TR,TC>& aIn)	40
2.2.9	operator << <> (std::ostream& os, const Array<TR,TC>& aOut)	41
2.3	rvector	42
2.3.1	rvector ()	45
2.3.2	rvector (int)	46
2.3.3	rvector (int, TR)	47
2.3.4	rvector (TR*,int,int)	48
2.3.5	rvector (const TR*,int,int)	49
2.3.6	rvector (const rvector&)	50
2.3.7	operator = (const rvector&)	51
2.3.8	assign(const TR*, int)	52
2.3.9	assign(int, const TR*, int)	53
2.3.10	assign(int, const TR*, int, int)	54
2.3.11	assign (int, const rvector&)	55
2.3.12	set(TR)	56
2.3.13	resize	57
2.3.14	operator ==	58
2.3.15	operator !=	59
2.3.16	operator <<	60
2.3.17	operator +	61
2.3.18	operator -	62
2.3.19	sum	63
2.3.20	diff	64
2.3.21	operator +=	65
2.3.22	operator -=	66
2.3.23	operator - ()	67
2.3.24	operator * (TR)	68
2.3.25	operator / (TR)	69
2.3.26	operator *=	70
2.3.27	operator /=	71
2.3.28	normalize	72
2.3.29	operator * (const rvector&)	73
2.3.30	operator * (const rmatrix&)	74
2.3.31	mult (const rvector&, const rmatrix&)	75
2.3.32	mult (const rmatrix&, const rvector&)	76
2.3.33	rank1update	77

2.3.34	solve	78
2.3.35	solve_tran	79
2.3.36	operator / (const srmatrix&)	80
2.3.37	operator % (const srmatrix&)	81
2.3.38	solve_lu	82
2.3.39	gels	84
2.3.40	gelsy	85
2.3.41	gelss	86
2.3.42	gelsd	87
2.3.43	svd	88
2.3.44	eig	90
2.3.45	gemv	92
2.3.46	gbmv	93
2.3.47	randomize	94
2.4	cvector	95
2.4.1	cvector ()	98
2.4.2	cvector (int)	99
2.4.3	cvector (int, TC)	100
2.4.4	cvector (TC*,int,int)	101
2.4.5	cvector (const TC*,int,int)	102
2.4.6	cvector (const cvector&)	103
2.4.7	cvector (const TR*,const TR*,int,int,int)	104
2.4.8	cvector (const rvector&, const rvector&)	105
2.4.9	cvector (const TR*,int,bool,int)	106
2.4.10	cvector (const rvector&,bool)	107
2.4.11	real	108
2.4.12	imag	109
2.4.13	operator = (const cvector&)	110
2.4.14	assign(const TC*, int)	111
2.4.15	assign(int, const TC*, int)	112
2.4.16	assign(int, const TR*, int, int)	113
2.4.17	assign (int, const cvector&)	114
2.4.18	set(TC)	115
2.4.19	assign_real	116
2.4.20	assign_imag	117
2.4.21	set_real	118
2.4.22	set_imag	119
2.4.23	resize	120
2.4.24	operator ==	121
2.4.25	operator !=	122
2.4.26	operator <<	123
2.4.27	operator +	124
2.4.28	operator -	125



2.4.29	sum	126
2.4.30	diff	127
2.4.31	operator +=	128
2.4.32	operator -=	129
2.4.33	operator - ()	130
2.4.34	operator * (TR)	131
2.4.35	operator / (TR)	132
2.4.36	operator * (TC)	133
2.4.37	operator / (TC)	134
2.4.38	operator *= (TR)	135
2.4.39	operator /= (TR)	136
2.4.40	operator *= (TC)	137
2.4.41	operator /= (TC)	138
2.4.42	normalize	139
2.4.43	conjugation	140
2.4.44	operator * (const cvector&)	141
2.4.45	operator %	142
2.4.46	operator * (const cmatrix&)	143
2.4.47	mult (const cvector&, const cmatrix&)	144
2.4.48	mult (const cmatrix&, const cvector&)	145
2.4.49	rank1update_u	146
2.4.50	rank1update_c	147
2.4.51	solve	148
2.4.52	solve_tran	149
2.4.53	solve_conj	150
2.4.54	operator / (const scmatrix&)	151
2.4.55	operator % (const scmatrix&)	152
2.4.56	solve_lu	153
2.4.57	gels	155
2.4.58	gelsy	157
2.4.59	gelss	158
2.4.60	gelsd	159
2.4.61	eig	160
2.4.62	gemv	162
2.4.63	gbmv	163
2.4.64	randomize_real	164
2.4.65	randomize_imag	165
2.5	Matrix	166
2.5.1	msize	167
2.5.2	nsize	168
2.5.3	ld	169
2.5.4	rowofmax	170
2.5.5	rowofmin	171

2.5.6	colofmax	172
2.5.7	colofmin	173
2.5.8	norm1	174
2.5.9	operator >> <> (std::istream& is, Matrix<TR,TC>& mIn)	175
2.5.10	operator << <> (std::ostream& os, const Matrix<TR,TC>& mOut)	176
2.6	rmatrix	177
2.6.1	rmatrix ()	181
2.6.2	rmatrix (int,int)	182
2.6.3	rmatrix (TR*,int,int)	183
2.6.4	rmatrix (const TR*,int,int)	184
2.6.5	rmatrix (const rmatrix&)	185
2.6.6	rmatrix (const rvector&,bool)	186
2.6.7	submatrix	187
2.6.8	operator (,)	188
2.6.9	operator ()	189
2.6.10	operator []	190
2.6.11	diag	191
2.6.12	operator = (const rmatrix&)	192
2.6.13	assign (const TR*)	193
2.6.14	assign (int, int, const rmatrix&)	194
2.6.15	set (TR)	195
2.6.16	resize	196
2.6.17	operator ==	197
2.6.18	operator !=	198
2.6.19	operator <<	199
2.6.20	operator +	200
2.6.21	operator -	201
2.6.22	sum	202
2.6.23	diff	203
2.6.24	operator +=	204
2.6.25	operator -=	205
2.6.26	operator - ()	206
2.6.27	operator * (TR)	207
2.6.28	operator / (TR)	208
2.6.29	operator *= (TR)	209
2.6.30	operator /= (TR)	210
2.6.31	normalize	211
2.6.32	transposition	212
2.6.33	operator * (const rvector&)	213
2.6.34	operator * (const rmatrix&)	214
2.6.35	mult	215
2.6.36	rank1update	216
2.6.37	swap_rows	217

2.6.38	swap_cols	218
2.6.39	solve	219
2.6.40	solve_tran	220
2.6.41	solve_lu	221
2.6.42	svd	223
2.6.43	pinv	225
2.6.44	gels	227
2.6.45	gelsy	229
2.6.46	gelss	231
2.6.47	gelsd	233
2.6.48	rank	235
2.6.49	ger	236
2.6.50	gemm	238
2.6.51	symm	239
2.6.52	qr	241
2.6.53	lq	242
2.6.54	rq	243
2.6.55	ql	244
2.6.56	vanish	245
2.6.57	randomize	246
2.7	cmatrix	247
2.7.1	cmatrix ()	251
2.7.2	cmatrix (int,int)	252
2.7.3	cmatrix (TC*,int,int)	253
2.7.4	cmatrix (const TC*,int,int)	254
2.7.5	cmatrix (const cmatrix&)	255
2.7.6	cmatrix (const cvector&,bool)	256
2.7.7	cmatrix (const rmatrix&,bool)	257
2.7.8	cmatrix (const TR*,const TR*,int,int)	258
2.7.9	cmatrix (const rmatrix&, const rmatrix&)	259
2.7.10	submatrix	260
2.7.11	operator (,)	261
2.7.12	operator ()	262
2.7.13	operator []	263
2.7.14	diag	264
2.7.15	real	265
2.7.16	imag	266
2.7.17	operator = (const cmatrix&)	267
2.7.18	assign (const TC*)	268
2.7.19	assign (int, int, const cmatrix&)	269
2.7.20	set (TC)	270
2.7.21	assign_real	271
2.7.22	assign_imag	272

2.7.23	set_real . . . . .	273
2.7.24	set_imag . . . . .	274
2.7.25	resize . . . . .	275
2.7.26	operator == . . . . .	276
2.7.27	operator != . . . . .	277
2.7.28	operator << . . . . .	278
2.7.29	operator + . . . . .	279
2.7.30	operator - . . . . .	280
2.7.31	sum . . . . .	281
2.7.32	diff . . . . .	282
2.7.33	operator += . . . . .	283
2.7.34	operator -= . . . . .	284
2.7.35	operator - () . . . . .	285
2.7.36	operator * (TR) . . . . .	286
2.7.37	operator / (TR) . . . . .	287
2.7.38	operator * (TC) . . . . .	288
2.7.39	operator / (TC) . . . . .	289
2.7.40	operator *= (TR) . . . . .	290
2.7.41	operator /= (TR) . . . . .	291
2.7.42	operator *= (TC) . . . . .	292
2.7.43	operator /= (TC) . . . . .	293
2.7.44	normalize . . . . .	294
2.7.45	conjugation . . . . .	295
2.7.46	transposition . . . . .	296
2.7.47	operator * (const cvector&) . . . . .	297
2.7.48	operator * (const cmatrix&) . . . . .	298
2.7.49	mult . . . . .	299
2.7.50	rank1update_u . . . . .	300
2.7.51	rank1update_c . . . . .	301
2.7.52	swap_rows . . . . .	302
2.7.53	swap_cols . . . . .	303
2.7.54	solve . . . . .	304
2.7.55	solve_tran . . . . .	305
2.7.56	solve_conj . . . . .	306
2.7.57	solve_lu . . . . .	307
2.7.58	svd . . . . .	309
2.7.59	pinv . . . . .	311
2.7.60	gels . . . . .	313
2.7.61	gelsy . . . . .	315
2.7.62	gelss . . . . .	317
2.7.63	gelsd . . . . .	319
2.7.64	rank . . . . .	321
2.7.65	qr . . . . .	322

2.7.66	lq	323
2.7.67	rq	324
2.7.68	ql	325
2.7.69	vanish	326
2.7.70	geru	327
2.7.71	gerc	329
2.7.72	gemm	331
2.7.73	hemm	333
2.7.74	randomize_real	335
2.7.75	randomize_imag	336
2.8	srmatrix	337
2.8.1	srmatrix ()	340
2.8.2	srmatrix (int)	341
2.8.3	srmatrix (TR*,int)	342
2.8.4	srmatrix (const TR*,int)	343
2.8.5	srmatrix (const srmatrix&)	344
2.8.6	srmatrix (const rmatrix&)	345
2.8.7	srmatrix (const rvector&)	346
2.8.8	submatrix	347
2.8.9	operator (,)	348
2.8.10	operator ()	349
2.8.11	operator []	350
2.8.12	operator = (const srmatrix&)	351
2.8.13	assign (const TR*)	352
2.8.14	assign (int, int, const rmatrix&)	353
2.8.15	set (TR)	354
2.8.16	resize	355
2.8.17	operator <<	356
2.8.18	operator +	357
2.8.19	operator -	358
2.8.20	sum	359
2.8.21	diff	360
2.8.22	operator +=	361
2.8.23	operator -=	362
2.8.24	operator - ()	363
2.8.25	operator ++	364
2.8.26	operator --	365
2.8.27	operator * (TR)	366
2.8.28	operator / (TR)	367
2.8.29	operator *= (TR)	368
2.8.30	operator /= (TR)	369
2.8.31	normalize	370
2.8.32	transposition	371

2.8.33	operator * (const rvector&)	372
2.8.34	operator * (const rmatrix&)	373
2.8.35	operator * (const srmatrix&)	374
2.8.36	operator *= (const srmatrix&)	375
2.8.37	swap_rows	376
2.8.38	swap_cols	377
2.8.39	solve	378
2.8.40	solve_tran	380
2.8.41	operator % (const rvector&)	382
2.8.42	operator / (const rvector&)	383
2.8.43	solve_lu	384
2.8.44	det	386
2.8.45	low_up	387
2.8.46	cond	389
2.8.47	inv	390
2.8.48	exp	391
2.8.49	polynomial	393
2.8.50	eig	395
2.8.51	Cholesky	397
2.8.52	Bunch-Kaufman	398
2.8.53	qr	399
2.8.54	lq	400
2.8.55	ql	401
2.8.56	rq	402
2.8.57	identity	403
2.8.58	vanish	404
2.8.59	randomize	405
2.9	scmatrix	406
2.9.1	scmatrix ()	410
2.9.2	scmatrix (int)	411
2.9.3	scmatrix (TC*,int)	412
2.9.4	scmatrix (const TC*,int)	413
2.9.5	scmatrix (const scmatrix&)	414
2.9.6	scmatrix (const cmatrix&)	415
2.9.7	scmatrix (const cvector&)	416
2.9.8	scmatrix (const srmatrix&,bool)	417
2.9.9	scmatrix (const TR*,const TR*,int)	418
2.9.10	scmatrix (const srmatrix&, const srmatrix&)	419
2.9.11	submatrix	420
2.9.12	operator (,)	421
2.9.13	operator ()	422
2.9.14	operator []	423
2.9.15	real	424

2.9.16	imag	425
2.9.17	operator = (const scmatrix&)	426
2.9.18	assign (const TC*)	427
2.9.19	assign (int, int, const cmatrix&)	428
2.9.20	set (TC)	429
2.9.21	assign_real	430
2.9.22	assign_imag	431
2.9.23	set_real	432
2.9.24	set_imag	433
2.9.25	resize	434
2.9.26	operator <<	435
2.9.27	operator +	436
2.9.28	operator -	437
2.9.29	sum	438
2.9.30	diff	439
2.9.31	operator +=	440
2.9.32	operator -=	441
2.9.33	operator - ()	442
2.9.34	operator ++	443
2.9.35	operator --	444
2.9.36	operator * (TR)	445
2.9.37	operator / (TR)	446
2.9.38	operator * (TC)	447
2.9.39	operator / (TC)	448
2.9.40	operator *= (TR)	449
2.9.41	operator /= (TR)	450
2.9.42	operator *= (TC)	451
2.9.43	operator /= (TC)	452
2.9.44	normalize	453
2.9.45	conjugation	454
2.9.46	transposition	455
2.9.47	operator * (const cvector&)	456
2.9.48	operator * (const cmatrix&)	457
2.9.49	operator * (const scmatrix&)	458
2.9.50	operator *= (const scmatrix&)	459
2.9.51	swap_rows	460
2.9.52	swap_cols	461
2.9.53	solve	462
2.9.54	solve_tran	464
2.9.55	solve_conj	466
2.9.56	operator % (const cvector&)	468
2.9.57	operator / (const cvector&)	469
2.9.58	solve_lu	470

2.9.59	det	472
2.9.60	low_up	473
2.9.61	cond	475
2.9.62	inv	476
2.9.63	exp	477
2.9.64	polynomial	479
2.9.65	eig	481
2.9.66	Cholesky	482
2.9.67	Bunch-Kaufman	483
2.9.68	qr	484
2.9.69	lq	485
2.9.70	rq	486
2.9.71	ql	487
2.9.72	identity	488
2.9.73	vanish	489
2.9.74	randomize_real	490
2.9.75	randomize_imag	491
2.10	BandMatrix	492
2.10.1	lsize	493
2.10.2	usize	494
2.11	srbmatrix	495
2.11.1	srbmatrix ()	497
2.11.2	srbmatrix (int)	498
2.11.3	srbmatrix (int,int,int)	499
2.11.4	srbmatrix (TR*,int,int,int)	500
2.11.5	srbmatrix (const TR*,int,int,int)	501
2.11.6	srbmatrix (const srbmatrix&)	502
2.11.7	srbmatrix (const rmatrix&,int,int)	503
2.11.8	srbmatrix (const rvector&)	504
2.11.9	operator (,)	505
2.11.10	operator ()	506
2.11.11	operator []	507
2.11.12	operator = (const srbmatrix&)	508
2.11.13	assign (const TR*)	509
2.11.14	set (TR)	510
2.11.15	resize	511
2.11.16	resize_lu	512
2.11.17	operator ==	513
2.11.18	operator !=	514
2.11.19	operator <<	515
2.11.20	operator +	516
2.11.21	operator -	517
2.11.22	sum	518



2.11.23	diff	519
2.11.24	operator +=	520
2.11.25	operator -=	521
2.11.26	operator - ()	522
2.11.27	operator ++	523
2.11.28	operator --	524
2.11.29	operator * (TR)	525
2.11.30	operator / (TR)	526
2.11.31	operator *= (TR)	527
2.11.32	operator /= (TR)	528
2.11.33	normalize	529
2.11.34	transposition	530
2.11.35	operator * (const rvector&)	531
2.11.36	operator * (const rmatrix&)	532
2.11.37	operator * (const srmatrix&)	533
2.11.38	operator * (const srbmatrix&)	534
2.11.39	low_up	535
2.11.40	operator / (const rvector&)	537
2.11.41	identity	538
2.11.42	vanish	539
2.11.43	randomize	540
2.12	scbmatrix	541
2.12.1	scbmatrix ()	543
2.12.2	scbmatrix (int)	544
2.12.3	scbmatrix (int,int,int)	545
2.12.4	scbmatrix (TC*,int,int,int)	546
2.12.5	scbmatrix (const TC*,int,int,int)	547
2.12.6	scbmatrix (const scbmatrix&)	548
2.12.7	scbmatrix (const cmatrix&,int,int)	549
2.12.8	scbmatrix (const cvector&)	550
2.12.9	scbmatrix (const srbmatrix&,bool)	551
2.12.10	scbmatrix (const srbmatrix&, const srbmatrix&)	552
2.12.11	operator (,)	553
2.12.12	operator ()	554
2.12.13	operator []	555
2.12.14	real	556
2.12.15	imag	557
2.12.16	operator = (const scbmatrix&)	558
2.12.17	assign (const TC*)	559
2.12.18	set (TC)	560
2.12.19	assign_real	561
2.12.20	assign_imag	562
2.12.21	set_real	563

2.12.22	set_imag	564
2.12.23	resize	565
2.12.24	resize_lu	566
2.12.25	operator ==	567
2.12.26	operator !=	568
2.12.27	operator <<	569
2.12.28	operator +	570
2.12.29	operator -	571
2.12.30	sum	572
2.12.31	diff	573
2.12.32	operator +=	574
2.12.33	operator -=	575
2.12.34	operator - ()	576
2.12.35	operator ++	577
2.12.36	operator --	578
2.12.37	operator * (TR)	579
2.12.38	operator / (TR)	580
2.12.39	operator * (TC)	581
2.12.40	operator / (TC)	582
2.12.41	operator *= (TR)	583
2.12.42	operator /= (TR)	584
2.12.43	operator *= (TC)	585
2.12.44	operator /= (TC)	586
2.12.45	normalize	587
2.12.46	conjugation	588
2.12.47	transposition	590
2.12.48	operator * (const cvector&)	592
2.12.49	operator * (const cmatrix&)	593
2.12.50	operator * (const scmatrix&)	594
2.12.51	operator * (const scbmatrix&)	595
2.12.52	low_up	596
2.12.53	operator / (const cvector&)	598
2.12.54	identity	599
2.12.55	vanish	600
2.12.56	randomize_real	601
2.12.57	randomize_imag	602
2.13	srsmatrix	603
2.13.1	srsmatrix ()	605
2.13.2	srsmatrix (int)	606
2.13.3	srsmatrix (TR*,int)	607
2.13.4	srsmatrix (const TR*,int)	608
2.13.5	srsmatrix (const srsmatrix&)	609
2.13.6	srsmatrix (const rmatrix&)	610

2.13.7	srsmatrix (const rvector&)	611
2.13.8	submatrix	612
2.13.9	operator (,)	613
2.13.10	operator ()	614
2.13.11	operator []	615
2.13.12	diag	616
2.13.13	operator = (const srsmatrix&)	617
2.13.14	assign (const TR*)	618
2.13.15	assign (int, const srsmatrix&)	619
2.13.16	set (TR)	620
2.13.17	set (int,int,TR)	621
2.13.18	set_diag (int,rvector)	622
2.13.19	resize	623
2.13.20	operator ==	624
2.13.21	operator !=	625
2.13.22	operator <<	626
2.13.23	operator +	627
2.13.24	operator -	628
2.13.25	sum	629
2.13.26	diff	630
2.13.27	operator +=	631
2.13.28	operator -=	632
2.13.29	operator - ()	633
2.13.30	operator ++	634
2.13.31	operator --	635
2.13.32	operator * (TR)	636
2.13.33	operator / (TR)	637
2.13.34	operator *= (TR)	638
2.13.35	operator /= (TR)	639
2.13.36	normalize	640
2.13.37	transposition	641
2.13.38	operator * (const rvector&)	642
2.13.39	operator * (const rmatrix&)	643
2.13.40	operator * (const srmatrix&)	644
2.13.41	operator / (const rvector&)	645
2.13.42	syrk	646
2.13.43	syr2k	648
2.13.44	inv	650
2.13.45	exp	651
2.13.46	polynomial	653
2.13.47	eig	655
2.13.48	Cholesky	656
2.13.49	Bunch-Kaufman	657

2.13.50	identity . . . . .	658
2.13.51	vanish . . . . .	659
2.13.52	randomize . . . . .	660
2.14	schmatrix . . . . .	661
2.14.1	schmatrix () . . . . .	664
2.14.2	schmatrix (int) . . . . .	665
2.14.3	schmatrix (TC*,int) . . . . .	666
2.14.4	schmatrix (const TC*,int) . . . . .	667
2.14.5	schmatrix (const schmatrix&) . . . . .	668
2.14.6	schmatrix (const cmatrix&) . . . . .	669
2.14.7	schmatrix (const rvector&) . . . . .	670
2.14.8	schmatrix (const srmatrix&) . . . . .	671
2.14.9	schmatrix (const TR*,const TR*,int) . . . . .	672
2.14.10	schmatrix (const srmatrix&, const srmatrix&) . . . . .	673
2.14.11	submatrix . . . . .	674
2.14.12	operator (,) . . . . .	675
2.14.13	operator () . . . . .	676
2.14.14	operator [] . . . . .	677
2.14.15	diag . . . . .	678
2.14.16	real . . . . .	679
2.14.17	imag . . . . .	680
2.14.18	operator = (const schmatrix&) . . . . .	681
2.14.19	assign (const TC*) . . . . .	682
2.14.20	assign (int, int, const schmatrix&) . . . . .	683
2.14.21	set (int,int,TC) . . . . .	684
2.14.22	set_diag . . . . .	685
2.14.23	set_main_diag . . . . .	686
2.14.24	assign_real . . . . .	687
2.14.25	set_real . . . . .	688
2.14.26	resize . . . . .	689
2.14.27	operator == . . . . .	690
2.14.28	operator != . . . . .	691
2.14.29	operator << . . . . .	692
2.14.30	operator + . . . . .	693
2.14.31	operator - . . . . .	694
2.14.32	sum . . . . .	695
2.14.33	diff . . . . .	696
2.14.34	operator += . . . . .	697
2.14.35	operator -= . . . . .	698
2.14.36	operator - () . . . . .	699
2.14.37	operator ++ . . . . .	700
2.14.38	operator -- . . . . .	701
2.14.39	operator * (TR) . . . . .	702

2.14.40	operator / (TR)	703
2.14.41	operator * (TC)	704
2.14.42	operator / (TC)	705
2.14.43	operator *= (TR)	706
2.14.44	operator /= (TR)	707
2.14.45	normalize	708
2.14.46	conjugation	709
2.14.47	transposition	710
2.14.48	operator * (const cvector&)	711
2.14.49	operator * (const cmatrix&)	712
2.14.50	operator * (const scmatrix&)	713
2.14.51	operator / (const cvector&)	714
2.14.52	herk	715
2.14.53	her2k	717
2.14.54	inv	720
2.14.55	exp	721
2.14.56	polynomial	723
2.14.57	eig	725
2.14.58	Cholesky	726
2.14.59	Bunch-Kaufman	727
2.14.60	identity	728
2.14.61	vanish	729
2.14.62	randomize_real	730
2.14.63	randomize_imag	731
2.15	cvmexception	732
2.15.1	cause	733
2.15.2	what	734
2.15.3	Customization	735
2.16	Utilities	736
2.16.1	cvmMalloc	737
2.16.2	cvmAddRef	738
2.16.3	cvmFree	739
2.16.4	cvmExit	740
2.16.5	cvmMachMin	741
2.16.6	cvmMachSp	742
2.16.7	eye_real	743
2.16.8	eye_complex	744
2.16.9	operator *	745
2.17	Pool Manager	747