# Template Matrix/Vector Library for C++ User Manual Version 0.73

Mike Jarvis

March 20, 2016

## Contents

# 1 Overview

First, this library is provided without any warranty of any kind. There is no guarantee that the code will produce accurate results for all inputs. If someone dies because my code gave you a wrong result, do not blame me.

OK, that was mostly for the crazies out there. Really, I think this is a pretty good product, and I've been using it for my own research extensively. So at least for the routines that I use, they are probably reasonably well debugged. I also have a test suite, which tries to be comprehensive, although occasionally I find bugs that I hadn't thought to test for in the test suite, so there may still be a few lurking in there. That means the code should definitely be considered a beta-type release. Hence the "0." version number. I'm also still adding functionality, so there may be interface changes from one version to the next if I decide I want to do something a different way. Just be warned. Any such changes will be mentioned in §2 of subsequent releases. And see §18 for changes introduced in previous versions.

## 1.1 Features of TMV

The Template Matrix/Vector (TMV) Library is a C++ class library designed to make writing code with vectors and matrices both transparent and fast. Transparency means that when you look at your code months later, it is obvious what the code does, making it easier to debug. Fast means the execution time of the code should be as fast as possible - this is mostly algorithm dependent, so we want the underlying library code to use the fastest algorithms possible.

If there were another free C++ Matrix library available that satisfied these requirements and provided all (or even most) of the functionality I wanted, I probably would not have written this. But, at least when I started writing this, the available matrix libraries were not very good. Either they didn't have good operator overloading, or they didn't do complex matrices well, or they didn't include singular value decompositions, or something. Anyway, since I did decide to write my own library, hopefully other people can benefit from my efforts and will find this to be a useful product.

Given the above basic guidelines, the specific design features that I have incorporated into the code include:

1. **Operator overloading**

   Matrix equations look like real math equations in the code. For example, one can write

   ```
   v2 = m * v1;
   v2 += 5 * m.transpose() * v1;
   m *= 3.;
   v2 += m1*v1 + 3*v2 + 8*m2.transpose()*v3;
   ```

   to perform the corresponding mathematical operations.

   I also define division:

   ```
   x = b/A;
   ```

   to mean solve the matrix equation $Ax = b$. If $A$ has more rows than columns, then the solution will be a least-square solution.

2. **Delayed evaluation**

   Equations like the above

   ```
   v2 = m * v1;
   v2 += 5 * m.transpose() * v1;
   ```

   do not create a temporary vector before assigning or adding the result to `v2`. This makes the TMV code just as fast as code which uses a functional form, such as:

```
dgemv('N',m,n,1.,mp,ldm,v1p,s1,0.,v2p,s2);
dgemv('T',m,n,5.,mp,ldm,v1p,s1,1.,v2p,s2);
```

In fact, when installed with a BLAS library, these are the exact BLAS functions that the above TMV statements will call behind the scenes. So speed is not sacrificed for the sake of code legibility.

However, a more complicated equation like

```
v2 += m1*v1 + 3*v2 + 8*m2.transpose()*v3;
```

does not have a specialized routine, so it will require a couple temporary `Vector`s. Generally, if a statement performs just one operation, no temporary will be needed. (This includes all operations with corresponding BLAS functions along with some others that are not included in BLAS.) More complicated equations like this last example will give the right answer, but may not be quite as efficient as if you expand the code to perform one operation per line.

3. **Template classes**

   Of course, all of the matrix and vector classes are templates, so you can have

   ```
   Matrix<float>
   Matrix<double>
   Matrix<complex<double> >
   Matrix<long double>
   Matrix<MyQuadPrecisionType>
   ```

   or whatever.

4. **Mix complex/real**

   One can multiply a real matrix by a complex vector without having to copy the matrix to a complex one for the sake of the calculation and deal with the concomitantly slower calculation. Likewise for the other arithmetic operations.

   However, it does not allow mixing of underlying data types (`float` with `double`, for example), with the exception of simple assignments.

5. **Views**

   Operations like `m.transpose()` or `v.subVector(3,8)` return "views" of the underlying data rather than copying to new storage. This model helps delay calculations, which increases efficiency. And the syntax is fairly obvious. For example:

   ```
   v.subVector(3,8) *= 3.;
   m.row(3) += 4. * m.row(0);
   m *= m.transpose();
   ```

   modifies the underlying `v` and `m` in the obvious ways.

   Note that in the last equation, `m.transpose()` uses the same storage as `m`, which is getting overwritten in the process. The code recognizes this conflict and uses temporary storage to obtain the correct result. See "Alias checking" below for more about this.

6. **C- or Fortran-style indexing**

   Both C- and Fortran-style (i.e. zero-based or one-based) indexing are possible for element access of the matrices and vectors.

   With C-style indexing, all matrix and vector indexing starts with 0. So the upper-left element of a matrix is `m(0,0)`, not `m(1,1)`. Likewise, the lower right element of an $M \times N$ matrix is `m(M-1,N-1)`. For element ranges, such as `v.subVector(0,10)`, the first number is the index of the first element, and the second number is "one-past-the-end" of the range. So, this would return a 10 element vector from `v(0)` to `v(9)` inclusive, not an 11 element one including `v(10)`.

With Fortran-style indexing, all matrix and vector indexing starts with 1. So the upper-left element of a matrix is `m(1,1)`. Likewise, the lower right element of an $M \times N$ matrix is `m(M,N)`. For element ranges, such as `v.subVector(1,10)`, the first number is the index of the first element, and the second number is the last element. So, this would return a 10 element vector from `v(1)` to `v(10)` inclusive, which represents the same actual elements as the C-style example above.

7. **Special matrices**

   Many applications use matrices with known sparsity or a specific structure. The code is able to exploit a number of these structures for increased efficiency in both speed and storage. So far the following special matrices are available: diagonal, upper/lower triangle, symmetric, hermitian, banded, and symmetric or hermitian banded. Special types of banded matrices, such as upper and lower banded, tridiagonal, or Hessenberg, may all be declared as a regular `BandMatrix`. The code checks the number of sub- and super-diagonals and uses the right algorithm when such specialization is advantageous for a particular calculation.

8. **Flexible storage**

   Both row-major and column-major storage are possible as an optional extra template parameter. For banded matrices, there is also diagonal-major storage. This can aid I/O, which may require a particular format to mesh with another program. Also, some algorithms are faster for one storage than than the other, so it can be worth switching storage and checking the speed difference.

9. **Alias checking**

   Expressions such as `m *= m` pose a problem for many matrix libraries, since no matter what order you do the calculation, you will necessarily overwrite elements that you need for later stages of the calculation. The TMV code automatically recognizes the conflict (generally known as an alias) and creates the needed temporary storage.

   The code only checks the addresses of the first elements of the different objects. So expressions such as `m = m.lowerTri() * m.upperTri()` will work correctly, even though there are three types of matrices involved, since the address of the upper-left corner of each matrix is the same. (This particular expression does not even need a temporary. The code orders the steps of this calculation so it can be done in place.)

   However, `v.subVector(5,15) += v.subVector(0,10)` will be calculated incorrectly, since the subvectors start at different locations, so the code doesn't notice the aliasing. Here, elements 5-9 will be overwritten before they are added to the left-side vector.

   Therefore, some care is still needed on the part of the user. But this limited check is sufficient for most applications.

10. **BLAS**

    For the combinations of types for which there are existing BLAS routines, the code can call the optimized BLAS routines instead of its own code. For other combinations (or for user defined types like `MyQuadPrecisionType` or somesuch), TMV has a native implementation that it uses instead, using blocking and recursive techniques to make the code fairly fast on modern CPUs.

    This feature can be turned off at compile time if desired with the option `WITH_BLAS=false`, although this is not generally recommended if BLAS is available on your machine, and speed is important for your application. This is especially true if you have a highly optimized BLAS distribution for your machine like MKL or ACML. However, the native code is usually within a factor of 2 or so of even these BLAS implementations, so if you don't care much about optimizing the speed of your code, it's not so bad to use the native TMV code.

11. **LAPACK**

    TMV can also be configured to call LAPACK routines behind the scenes when possible, which may be faster than the native TMV code. For types that don't have LAPACK routines, or if you do not enable the LAPACK calls, the code uses blocked and/or recursive algorithms, which are similarly fast.

    For almost all algorithms, the TMV code is approximately as fast as LAPACK routines - sometimes faster, since most LAPACK distributions do not use recursive algorithms yet, which are generally slightly faster on modern machines with good compilers. So if you don't want to deal with getting LAPACK up and running, it won't generally be too bad, speedwise, to leave the LAPACK calls turned off.

    Also, it is worth noting that many LAPACK libraries are not thread safe. So if your main program uses multiple threads, and you aren't sure whether your LAPACK library is thread safe, you might want to compile TMV without LAPACK to avoid intermittent mysterious segmentation faults from the LAPACK library. I believe all the native TMV code is thread safe.

## 1.2 Basic usage

All of the basic TMV classes and functions, including the `Vector` and `Matrix` classes, can be accessed with

```
#include "TMV.h"
```

The special matrices described below (other than diagonal and triangular matrices) are not included in this header file. See their sections for the names of the files to include to access those classes.

All of the TMV classes and functions reside in the namespace `tmv`. And of course, they are all templates. So if you want to declare a $10 \times 10$ `Matrix`, one would write:

```
tmv::Matrix<double> m(10,10);
```

If writing `tmv::` all the time is cumbersome, one can use `using` statements near the top of the code:

```
using tmv::Matrix;
using tmv::Vector;
```

Or, while generally considered bad coding style, one can import the whole namespace:

```
using namespace tmv;
```

Or, you could use typedef to avoid having to write the template type as well:

```
typedef tmv::Matrix<double> DMatrix;
typedef tmv::Vector<double> DVector;
```

In this documentation, I will usually write `tmv::` with the class names to help remind the reader that it is necessary, especially near the beginnings of the sections. But for the sake of brevity and readability, I sometimes omit it.

## 1.3 Data types

Throughout most of the documentation, I will write `T` for the underlying type. Wherever you see `T`, you should put `double` or `std::complex<float>` or whatever.

For a user defined type, like `MyQuadPrecisionType` for example, the main requirements are that in addition to the usual arithmetic operators, the functions:

```
std::numeric_limits<T>::epsilon()
sqrt(T x)
exp(T x)
log(T x)
```

need to be defined appropriately, where T is your type name. See §3 for details about compiling the library for types other than double and float.

Some functions in this documentation will return a real value or require a real argument, even if T is complex. In these cases, I will write RT to indicate "the real type associated with T". Similarly, there are places where CT indicates "the complex type associated with T".

It is worth noting that Matrix<int> is possible as well if you compile the library with the SCons option INST_INT=true. However, only simple routines like multiplication and addition will give correct answers. If you try to divide by a Matrix<int>, for example, the required calculations are impossible for int's, so the result will not be correct. But since the possibility of multiplication of integer matrices seemed desirable, we do allow them to be used. *Caveat programor*. If debugging is turned on (with DEBUG=true in the scons installation), then trying to do anything that requires sqrt or epsilon for ints will result in a runtime error. Specifically a FailedAssert exception will be thrown.

The one somewhat complicated algorithm that is available for integer types is the determinant, since the determinant of an integer matrix is always an integer. The algorithm to do this is a so-called integer-preserving algorithm from Bareiss (1968) that only uses division when it is known that the result will not require a fractional part. I've extended his algorithm to complex<int> as well, so those are also accurate.

## 1.4 Getting the version of TMV

At times it can be useful to be able to access what version of TMV is installed on a particular machine, either to log the information as part of the meta-data about a run of a program, or even to modify the code depending on which features of TMV are available. To address this, we provide three ways to access this information.

First, there is a function you can call from within your program:

```
std::string TMV_Version();
```

For this release, this function returns the string "0.73". This is useful for inserting into a log file or anywhere else that you want to record the version somewhere.

Second we also provide three C preprocessor definitions:

```
TMV_MAJOR_VERSION
TMV_MINOR_VERSION
TMV_VERSION_AT_LEAST(major,minor)
```

The first two are defined to be 0 and 73 for this release. The third can be used with an #if directive to change what code you want to compile according the the version of TMV that is installed. For example, in version 0.65 we added the m.addToAll(x) function. So you could write:

```
#if TMV_VERSION_AT_LEAST(0,65)
m.addToAll(1.0);
#else
for(int i=0;i<m.nrows();++i)
    for(int j=0;j<m.ncols();++j)
        m(i,j) += 1.0;
#endif
```

And finally, we also include a bash script called tmv-version that will output the version number, so you can run this directly from the command line. For this release, this script produces the output:

```
$ tmv-version
0.73
$
```

This can be useful as part of a larger script if you want to log the TMV version from that rather than from within a C++ program.

Hopefully these three methods will satisfy any potential manner in which you might want to access the version of TMV that is on your system. If there is something I missed, and none of these work for you, please let me know, and I'll probably be happy to provide another mechanism in future releases.

## 1.5 Notations used in this document

There are three fonts used in this document. First, for the main text, we use times. Second, as you have no doubt already noticed, `typewriter font` is used to indicate bits of code. And finally, when I discuss the math about matrices, I use $italics$ – for example, $v_2 = m * v_1$.

Also, my code syntax in this documentation is not very rigorous, aiming to maximize readability of the code, rather than including all of the type specifications for everything.

I tend to freely mix the syntax of how a function is used with how it is declared in order to try to provide all of the information you will need on one line. For example, the constructor listed as:

```
tmv::Vector<T,A> v(int n, const T* vv)
```

is actually declared in the code as:

```
template <class T, int A>
tmv::Vector<T,A>::Vector(int n, const T* vv);
```

and when it is used in your source code, you would write something like:

```
int n = 5;
const double vv[n] = {1.2, 3.1, 9.1, 9.2, -3.5};
tmv::Vector<double,tmv::CStyle> v(n,vv);
```

So, the notation that I use in the documentation for this constructor is kind of a hybrid between the declaration syntax and the use syntax. The intent is to improve readability, but if you are ever confused about how to use a particular method, you should look at the `.h` header files themselves, since they obviously have the exactly accurate declarations.

## 2 Changes from version 0.72 to 0.73

This release is mostly just some minor bug fixes. Also, since Google Code went under, TMV is now hosted on GitHub at `https://github.com/rmjarvis/tmv`.

Here is a list of the changes from version 0.72 to 0.73. (See §18 for changes from previous versions to 0.72) This release is completely backwards compatible as far as the header files and library are concerned. The library should be link-compatible with previous versions.

The numbers at the ends of some items below indicate which issue is connected with the change:
`https://github.com/rmjarvis/tmv/issues?q=is%3Aissue+is%3Aclosed`

### 2.1 Bug Fixes

- Removed an extraneous 16 bytes of memory usage from `SmallVector`. (#12)
- Gives an error if PREFIX is the source directory (rather than just failing to work properly). (#13)
- Fixed compiler warnings from newer versions of clang++.
- Changed tmv-version to a simple echo line rather than the complicated thing we used to have that stopped working when we switched from svn to git. (#20)

### 2.2 Build updates

- Added support for OpenMP for clang v3.7+.
- Improved automatic library selection for MKL. (#15)
- Fixed broken `USE_UNKNOWN_VARS` feature. (#16)
- No longer includes `$PREFIX/include` in the include path by default. Use scons `IMPORT_PREFIX=true` if you want to include it. (#17)
- Added `LINKFLAGS` as an explicit option. (#21)
- No longer officially support Visual C++ installation. I suspect it still works, but I no longer include it in my list of tested systems.

### 2.3 Documatation updates

- Documented the sort order for eigenvectors returned by `Eigen` and singular values returned `SV_Decompose`. (#14)

# 3 Obtaining and compiling the library

First, there are two email groups that you may be interested in joining to get more information about TMV: `tmv-discuss@googlegroups.com` and `tmv-announce@googlegroups.com`. The first is an interactive list for discussing various issues related to TMV, such as new features, API suggestions, and various other comments that don't really qualify as bug reports. The second is a list for people who just want to get announcements about TMV – namely when a new version is released.

If you find a bug in the code, first check §17.1 for know problems with particular compilers or BLAS/LAPACK distributions and §17.3 for known deficiencies of the TMV library.

If the bug is not already mentioned in either of these sections, then the preferred place to report the bug is `https://github.com/rmjarvis/tmv/issues`. Ideally, please include a short program that reproduces the problem you are seeing, and make sure to mention which compiler (with version) you are using, any LAPACK or BLAS libraries you are linking with, and information about the system you are running the program on.

All of the TMV code is licensed using the Gnu General Public License. See §19 for more details.

## 3.1 Using Fink

Starting with TMV version 0.71, TMV is available on Fink for OSX systems. So if you use a Mac and you have Fink (`http://www.finkproject.org/`) installed on your system already, you just need to type

```
fink install tmv0
```

This should install TMV in the `/sw` directory with a minimum of fuss. (The 0 at the end of the package name is anticipation of an eventual upgrade to version 1.0 whose library file will not be backwards compatible with this one. That will then be fink package tmv1.)

The only caveats here are that it will always install a shared library (`libtmv.dylib`), and it will always use Apple's Frameworks BLAS library. Also, it uses whatever C++ compiler is the default for your version of fink (usually a variety of g++, but they switched to clang++ with MacOS 10.7). If these are not acceptable to you, then you will need to download TMV and install it yourself.

## 3.2 Installing the TMV library from source

### 3.2.1 Obtaining the source code

1. Go to `https://github.com/rmjarvis/tmv` for a link to a tarball with all of the source code (click the Releases tab), and copy it to the directory where you want to put the TMV library.

2. Unpack the tarball:
   ```
   gunzip tmv-0.73.tar.gz
   tar xf tmv-0.73.tar
   ```
   This will make a directory called `tmv-0.73` with the subdirectories: `doc`, `examples`, `include`, `lib`, `src` and `tests` along with the files `README`, `INSTALL` and others in the top directory.

### 3.2.2 Running SCons

Make sure you have SCons installed on your system, available at `http://www.scons.org/`. (It is a very quick installation if you already have Python installed.)

1. Type
   ```
   scons [Optional flags -- see below]
   ```
   This will make the libraries `libtmv.a` and `libtmv_symband.a` and put them into the directory `lib`.

There are a number of command-line options that you might need (but try it with no flags first – it can often find everything automatically). You can get a list of these options from the command line with `scons -h`. The options are listed with their default value. You change them simply by specifying a different value on the command line. For example:
`scons CXX=icpc INST_LONGDOUBLE=true`
If you need to run SCons multiple times (for example, to compile the test suite or install the libraries as described below), you only need to specify the new parameter values the first time you run SCons. The program automatically saves your options and continues to use them until you change a value again.

- `CXX=g++` specifies which C++ compiler to use.
- `FLAGS=''` specifies the basic flags to pass to the compiler. The default behavior is to automatically choose good flags to use according to which kind of compiler you are using. It has defaults for `g++`, `icpc` and `pgCC`. If you are using a different compiler or don't like the default, then you can specify this by hand. Remember to put the flags in quotes, so the whitespace doesn't confuse the parser. e.g. `scons FLAGS='-O3 -g'`
- `EXTRA_FLAGS=''` specifies extra flags to pass to the compiler in addition to what TMV will use by default. This is useful if you just want to add a specific flag but still want to keep the other flags that TMV uses automatically.
- `LINKFLAGS=''` specifies the flags to use at link time.
- `DEBUG=false` specifies whether to keep the debugging assert statements in the compiled library code. `DEBUG=true` turns on all the TMV debugging statements, even those that would normall require `-DTMV_EXTRA_DEBUG`.
- `PREFIX=/usr/local` specifies where to install the library when running `scons install` (see below).
- `FINAL_PREFIX=''` specifies a final installation directory if different from `PREFIX`. This is used for installations (e.g. fink) that install everything in a temporary directory first and then copy everthing to a final location only if the installation was successful.
- `INST_FLOAT=true` specifies whether to instantiate the `<float>` templates.
- `INST_DOUBLE=true` specifies whether to instantiate the `<double>` templates.
- `INST_LONGDOUBLE=false` specifies whether to instantiate the `<long double>` templates.
- `INST_INT=false` specifies whether to instantiate the `<int>` templates.
- `WITH_OPENMP=true` specifies whether to use OpenMP to parallelize some parts of the code.
- `SHARED=true` specifies whether to make shared libraries as opposed to static libraries. Making the libraries shared (e.g. `libtmv.so`, `libtmv.dylib`, etc. depending on your system) allows the compiled programs that use TMV to be quite a bit smaller typically, so we've made that the default.
- `TEST_FLOAT=true` specifies whether to include the `<float>` tests in the test suite.
- `TEST_DOUBLE=true` specifies whether to include the `<double>` tests in the test suite.
- `TEST_LONGDOUBLE=false` specifies whether to include the `<long double>` tests in the test suite.
- `TEST_INT=false` specifies whether to include the `<int>` tests in the test suite.

The next flags set up the paths that SCons will use to try to find your BLAS and LAPACK libraries.

- `EXTRA_INCLUDE_PATH=''` specifies directories in which to search for header files (such as the BLAS or LAPACK header files) in addition to the standard locations such as `/usr/include` and `/usr/local/include`. These directories are specified as `-I` flags to the compiler. If you are giving multiple directories, they should be separated by colons.

- `EXTRA_LIB_PATH=''` specifies directories in which to search for libraries (such as the BLAS or LAPACK libraries) in addition to the standard locations such as `/usr/lib` and `/usr/local/lib`. These directories are specified as `-L` flags to the linker. If you are giving multiple directories, they should be separated by colons.
- `IMPORT_PATHS=false` specifies whether to import extra path directories from the environment variables: `PATH`, `C_INCLUDE_PATH`, `LD_LIBRARY_PATH` and `LIBRARY_PATH`.
- `IMPORT_ENV=true` specifies whether to import the entire environment from the calling shell. The default is for SCons to use the same environment as the shell from which it is called. However, sometimes it can be useful to start with a clean environment and manually add paths (see below) for various things, in which case you would want to set this to `false`.
- `EXTRA_PATH=''` specifies directories in which to search for executables (notably the compiler, although you can also just give the full path in the `CXX` parameter) in addition to the standard locations such as `/usr/bin` and `/usr/local/bin`. If you are giving multiple directories, they should be separated by colons.
- `IMPORT_PREFIX=true` specifies whether to use `PREFIX/include` and `PREFIX/lib` in your search paths. This is the default (and is equivalent to the old behavior), but sometimes it is necessary or preferable to not include these directories.

The next options can be used to specify what BLAS and/or LAPACK libraries to use (if any), overriding the default of using whatever libraries SCons chooses from searching through your path and trying to link the libraries that it finds. The `FORCE` options can be useful if SCons finds a library before trying the one that you want, or if SCons fails in the linking step even though the library should link successfully, or if you want to compile for a library that requires different linking instructions than the ones that SCons tries[1]. The `FORCE` options will try to test linking with the library requested, but if it fails, then it will just give a warning message.

- `WITH_BLAS=true` specifies whether to look for and try to use a BLAS library.
- `WITH_LAPACK=false` specifies whether to look for and try to use a LAPACK library. Note: the default here used to be `true`. But my experience has been that the TMV code is more stable than typical LAPACK distributions, especially with regard to overflow and underflow. And it is generally equally fast, and sometimes faster. The only exception is finding Eigenvectors for extremely large hermitian matrices[2]. And even then, TMV may still be faster if you use a machine with multiple cores, since TMV seems to have better parallelization of this algorithm (if `WITH_OPENMP=true`) than many LAPACK libraries. Also, it is worth noting that many LAPACK libraries are not thread safe. So if your main program uses multiple threads, and you aren't sure whether your LAPACK library is thread safe, you might want to compile TMV without LAPACK to avoid intermittent mysterious segmentation faults from the LAPACK library. All the native TMV code is thread safe.
- `FORCE_MKL=false` forces the use of the Intel Math Kernel library. It requires the header file `"mkl.h"` to be found in your path.
- `FORCE_ACML=false` forces the use of the AMD Core Math library. It requires the header file `"acml.h"` to be found in your path.
- `FORCE_GOTO=false` forces the use of the GotoBlas library.

[1] If you have a case of needing different linking instructions, and your BLAS or LAPACK is a standard installation on your machine (not some goofy personal installation that no one else will duplicate), then let me know and I'll add it to the SConstruct file for the next release.

[2] LAPACK implements an algorithm called Relatively Robust Representation (RRR) to find the eigenvectors, rather than the divide and conquer algorithm used by TMV. RRR is faster, but only when the matrix size is sufficiently large. Including this algorithm in TMV is #5 on my To-do list (§17).

- `FORCE_ATLAS=false` forces the use of the ATLAS library (for BLAS). It requires the header file `"cblas.h"` to be found in your path.
- `FORCE_CBLAS=false` forces the use of a CBLAS library.
- `FORCE_FBLAS=false` forces the use of a Fortran BLAS library.
- `FORCE_CLAPACK=false` forces the use of the CLAPACk library. It requires the CLAPACK version of the header file `"clapack.h"` to be found in your path.
- `FORCE_ATLAS_LAPACK=false` forces the use of the LAPACK portion of the ATLAS Library. It requires the ATLAS version of the header file `"clapack.h"` to be found in your path.
- `FORCE_FLAPACK=false` forces the use of a Fortran LAPACK library.
- `LIBS=''` directly specifies the library flags to use for linking if the automatic methods aren't working for you. Because of the way SCons works, these should omit the `-l` part of the flag, since SCons will add this to what is provided. For example, to specify an alternate name for the CLAPACK library, use `scons LIBS=lapack_LINUX`. Multiple libraries here should be separated by whitespace and enclosed in quotes.

Finally, some miscellaneous options that you are less likely to need:

- `STATIC=false` specifies whether to use static linkage for the test program. Some systems have trouble with dynamic linkage of libraries. This usually indicates that something is installed incorrectly, but it can be easier to just use static linkage when you compile as a workaround. This flag does this for the test suite executables.[3]
- `WITH_SSE=true` specifies whether to use the `-msse2` flag with `icpc` compilations. It seems like most machines that have `icpc` are able to use SSE commands. However, I couldn't figure out a compiler flag that would turn on SSE *if and only if* the machine supports it. So the default is to use the flag `-msse2`, but you can disable it by setting `WITH_SSE=false` if your machine doesn't have SSE support. In fact, I don't even know if this option is ever necessary, since `icpc` might be smart enough to ignore this flag if your machine doesn't support SSE instructions.
- `XTEST=0` specifies whether to include extra tests in the test suite. `XTEST` is treated as a bit set, with each non-zero bit turning on particular tests. Type "`scons -h`" for more information.
- `MEM_TEST=false` specifies whether to include extra memory tests in the library and test suite.
- `NAN_TEST=false` specifies whether to stress test the code's ability to deal with `nans` in the allocated memory by initializing all memory allocations with all `nans`.
- `USE_STEGR=false` specifies whether to use the LAPACK algorithm called `dstegr` (or `sstegr` for `<float>`) for symmetric eigenvector calculation. If it is false, the divide-and-conquer algorithm, named `dstedc` (or `sstedc`) will be used instead.

  I've had trouble with the LAPACK `?stegr` algorithms on a number of systems. Mostly, it doesn't always deal with `nan`'s and `inf`'s very well. The documentation for it says:

  > ?stegr works only on machines which follow IEEE-754 floating-point standard in their handling of infinities and NaNs. Normal execution of ?stegr may create NaNs and infinities and hence may abort due to a floating point exception in environments which do not conform to the IEEE-754 standard.

  I haven't had too many aborts (just one system I think). More often, I get things like `nans` in the output and such, so I guess these systems must not conform to IEEE-754. If you think your system is conforming, and you want to enable `stegr`, go ahead and set `USE_STEGR` to `true`.

[3]Note: combining `STATIC=true` with `SHARED=true` does not work.

- USE_GEQP3=false specifies whether to use the LAPACK algorithm called dgeqp3 (or its variants) for the strict QRP decomposition. I think the native TMV code is superior to the LAPACK code, so it is not enabled by default. But if you want to use for the so-called Strict QRP decomposition (c.f. §3), you can set this to true.
- SMALL_TESTS=false specifies whether to make the smaller test suite programs: tmvtest1a, tmvtest1b, etc.
- WARN=false specifies whether to have the compiler report warning (with -Wall or something similar).
- CACHE_LIB=true specifies whether to cache the results of the library checks. Scons is pretty good at only recompiling what needs to be recompiled based on what options or files you have changed. However, it is not good at detecting when changed options might change the results of a BLAS or LAPACK library check. So if you add new paths for it to search, or even moreso, if you change the names or locations of libraries on your system, then you should set CACHE_LIB=false to force SCons to redo the BLAS and LAPACK checks each time.
- WITH_UPS=false specifies whether to install TMV information for ups into the PREFIX/ups directory.
- N_BUILD_THREADS=0 specifies how many threads SCons should use when building the code. This is equivalent to using scons -jN, but this option gets saved, so you don't need to type it each time when doing multiple builds. The default value of 0 indicates that SCons should try to automatically determine the number of cpus and use that.

When SCons starts up, it will look through the standard paths, along with any extra paths you have specified with the above options, to find BLAS and LAPACK libraries. This can sometimes require a few iterations to get working correctly. You should look at the initial output from SCons to make sure it finds the correct BLAS and LAPACK libraries that you think it should find. Here is a sample output:[4]

```
$ scons
scons: Reading SConscript files ...
Using the default scons options
Using compiler: /usr/bin/g++
Detected clang++ masquerading as g++
compiler version: 5.1
Determined that there are 4 cpus, so use this many jobs.
You can override this behavior with scons -jN
Debugging turned off
Checking for MKL... no
Checking for ACML... no
Checking for GotoBLAS... no
Checking for CBLAS... no
Checking for ATLAS... no
Checking for Fortran BLAS... yes
Using Fortran BLAS
No OpenMP support for clang++
TMV Version  0.72

scons: done reading SConscript files.
scons: Building targets ...
```

[4] This is the exact output that I get with the default options on my MacBook.

(*Starts the actual compiling*)

If a "`Checking for...`" line ends with `no`, even though you think that library is installed on your computer, then it probably means that you need to tell SCons which directories to search, in addition to the standard locations. The most straightforward way to do this is with the parameters `EXTRA_INCLUDE_PATH` and `EXTRA_LIB_PATH`. These are described in detail above. See also `IMPORT_ENV` and `IMPORT_PATHS`.

2. Type
   `scons install`
   (or possibly `sudo scons install` if you are installing into `/usr/local` or somewhere similar).

   This will install the necessary header files into the directory `/usr/local/include` and the libraries into `/usr/local/lib`. As mentioned above, you can also specify a different prefix with the command line option `PREFIX=`*install-dir*. A common choice for users without `sudo` privileges is `PREFIX=~` which will install the library in `~/include` and `~/lib`.

   At the end of the installation process, you should see a message similar to:

```
The TMV library was successfully installed.
To link your code against the TMV library, you should use the
link flags:

-ltmv -lblas -lpthread -fopenmp

Or if you are using Band, Sym or SymBand matrices, use:

-ltmv_symband -ltmv -lblas -lpthread -fopenmp

These flags (except for the optional -ltmv_symband) have been
saved in the file:

/usr/local/share/tmv/tmv-link

so you can automatically use the correct flags in a makefile
(for example) by using lines such as:

TMVLINK := $(shell cat /usr/local/share/tmv/tmv-link)
LIBS = $(TMVLINK) [... other libs ...]

In a SConstruct file, you can do something like the following:

env.MergeFlags(open('/usr/local/share/tmv/tmv-link').read())


scons: done building targets.
```

   (Again, this is the actual output on my laptop when I type `sudo scons install`.) These instructions tell you what BLAS and LAPACK libraries (if any) were found on your system and are needed for proper linkage of your code. The linkage flags are stored in a file on your computer so that you can automatically get the linkage correct according to what options you decide to use when installing TMV. This is especially useful on systems where a system administrator installs the library, which is then used by many users.

You can have your `makefiles` or `SConstruct` files read this file as described above. The `examples` directory has a `makefile` that uses the above lines (using the local `share` directory rather than the installed location) that you can look at as a example.

3. (Optional) Type
   `scons tests`

   This will make three executables called `tmvtest1`, `tmvtest2` and `tmvtest3` in the `tests` directory.

   Then you should run the three test suites. They should output a bunch of lines reading `[`*`Something`*`] passed all tests`. If one of them ends in a line that starts with `Error`, then please post a bug report at `https://github.com/rmjarvis/tmv/issues` about the problem including what compiler you are using, some details about your system, and what (if any) BLAS and LAPACK libraries you are linking to.

   If you specify `SMALL_TESTS=true`, then the smaller test executables `tmvtest1a-d`, `tmvtest2a-c`, and `tmvtest3a-e` (where `a-d` means four files with each of `a`, `b`, `c` and `d`) will be made instead. These perform the same tests as the larger test executables, but can be easier for some linkers.

## 3.3 Compiling your own program

Each `.cpp` file that uses TMV will need to have

```
#include "TMV.h"
```

at the top.

Furthermore, if you are using band, symmetric, or hermitian matrices, you will need to include their particular header files as well:

```
#include "TMV_Band.h"
#include "TMV_Sym.h"
#include "TMV_SymBand.h"
```

When compiling your object files, there are a couple of compiler flags that you might want to use:

- `-DNDEBUG`
  `-DTMV_NDEBUG`

  Either of these will turn off the normal debugging statements in the TMV header files. All of the normal debugging statements in TMV take $O(1)$ time, so they are not too intrusive to leave turned on, and they can be very useful in catching errors in your code. For example, they check that the sizes of two matrices being multiplied or added match up correctly, and that the indices given for submatrices and subvectors fit into the main matrix or vector.

  The normal behavior that we think will be appropriate for most users is to leave these statements active during development, and then when you turn off your own `assert` statements with `-DNDEBUG`, that will turn off the TMV asserts as well. However, if you want to turn off the TMV asserts without otherwise affecting your own debug lines, then `-DTMV_NDEBUG` will do that.

- `-DTMV_EXTRA_DEBUG`

  This will turn on some extra debugging statements that can sometimes be useful. First, whenever you create a matrix that is nominally uninitialized (e.g. `tmv::Matrix<T> m(4,3);`), this will instead initialize it with all 888's. This can make it easier to find bugs, since uninitialized memory can often be 0's, which might hide that fact that you forgot to initialize some of the values to 0. So your code might work for a while and then give wrong answers when something else about your code changes.

Perhaps more important, when any matrix is destroyed, this will first set all the values to 999. That makes it easier to notice if you accidentally use a view to that matrix after that point. Again, without this step, your code might work correctly for a while, and then suddenly fail, so this can help you find such errors.

Finally, there are a few $O(N)$ asserts that get turned on with this flag, such as checking that hermitian views of a complex `Matrix` actually has real elements on the diagonal (as required for the view to be hermitian). This takes $O(N)$ time, since it needs to check every element on the diagonal.

Also, if you did not install TMV into a standard place like `/usr/local` (specified by `PREFIX` in the installation process), then you will also need to tell your compiler where you installed the header files with

```
-IPREFIX/include
```

Then for the linking step, you need to link your program with the TMV library with

```
-ltmv
```

or if TMV is not installed in your path then

```
-LPREFIX/lib -ltmv
```

If you are using band or symmetric/hermitian matrices, then you will need to link with the flags

```
-ltmv_symband -ltmv
```

And if you are using BLAS and/or LAPACK calls from the TMV code, then you will also need to link with whatever libraries you specified for TMV to use. For example, for my version of Intel's Math Kernel LIbrary, I use `-lmkl_lapack -lmkl_ia32 -lguide -lpthread`. For ATLAS, I use `-llapack -lcblas -latlas`. For your specific installation, you may need the same thing, or something slightly different, including possibly `-L` flags to indicate where the BLAS or LAPACK libraries are located.

As mentioned above, you can automatically use whatever BLAS libraries TMV found and compiled against by reading the file `/usr/local/share/tmv/tmv-link` (or `PREFIX/share/tmv/tmv-link`) and using that for your linking flags. You can use this file in a `makefile` by writing something like the following:

```
TMVLINK := $(shell cat /usr/local/share/tmv/tmv-link)
LIBS = $(TMVLINK) [... other libs ...]
```

Or in a `SConstruct` file, you can do something like

```
env.MergeFlags(open('/usr/local/share/tmv/tmv-link').read())
```

Remember though that if you are using band and/or symmetric/hermitian matrices, you also need to add the flag `-ltmv_symband` to `LIBS` as well.

# 4 Vectors

The `Vector` class in TMV represents a mathematical vector[5].

The `Vector` class is templated with two template arguments. The first one is the type of the data, which we call `T` below. The second template argument is an integer that defines some attributes about the `Vector`. The only valid attributes for a `Vector` are `tmv::CStyle` and `tmv::FortranStyle`, which specify how the element access is performed. (We'll have more attribute options later for matrices.) We call the attribute parameter `A` below. If it is omitted, `CStyle` is assumed.

With C-style indexing, the first element of a `Vector` of length `N` is `v(0)` and the last is `v(N-1)`. Also, methods that take range arguments use the common C convention of "one-past-the-end" for the last element; so `v.subVector(0,3)` returns a 3-element vector, not 4.

With Fortran-style indexing, the first element of the vector is `v(1)` and the last is `v(N)`. Ranges are specified by the first and last elements, so the same subvector as above would now be accessed using `v.subVector(1,3)` to return the first three elements.

All views of a `Vector` keep the same indexing style as the original unless you explicitly change it with a cast. You can cast a `VectorView<T,CStyle>` as a `VectorView<T,FortranStyle>` and vice versa. Likewise for `ConstVectorView`.

## 4.1 Constructors

Remember, the `A` attributes parameter listed here is optional. If it is omitted, `CStyle` is assumed.

- `tmv::Vector<T,A> v()`

  Makes a `Vector` of zero size. You would normally use the `resize` function later to change the size to some useful value.

- `tmv::Vector<T,A> v(int n)`

  Makes a `Vector` of size `n` with *uninitialized* values. If extra debugging is turned on (with the compiler flag `-DTMV_EXTRA_DEBUG`), then the values are in fact initialized to 888. This should help you notice when you have neglected to initialize the `Vector` correctly.

- `tmv::Vector<T,A> v(int n, T x)`

  Makes a `Vector` of size `n` with all values equal to `x`

- `tmv::Vector<T,A> v = tmv::BasisVector<T,A>(int n, int i)`

  Makes a `Vector` whose elements are all `0`, except `v(i) = 1`. Note the `BasisVector` also has the `A` template argument to indicate which element is meant by `v(i)`. Again, if it is omitted, `CStyle` is assumed.

- `tmv::Vector<T,A> v1(const tmv::Vector<T2,A2>& v2)`
  `v1 = v2`

  Copy the `Vector v2`, which may be of any type `T2` so long as values of type `T2` are convertible into type `T`. The assignment operator has the same flexibility.

[5] Don't confuse it with the standard template library's `vector` class. Our `Vector` class name is capitalized, while the STL `vector` is not. If this is not enough of a difference for you, and you are using both extensively in your code, we recommend keeping the full `tmv::Vector` designation for ours and `std::vector` for theirs to distinguish them. Or you might want to `typedef tmv::Vector` to some other name that you prefer.

-
  ```
  tmv::VectorView<T,A> v =
        tmv::VectorViewOf(T* vv, int n, int step=1)
  tmv::ConstVectorView<T,A> v =
        tmv::VectorViewOf(const T* vv, int n, int step=1)
  ```

  Makes a `VectorView` (see §4.5 below) which refers to the exact elements of `vv`, not copying them to new storage. The parameter `n` is the number of values to include in the view. The optional `step` parameter allows a non-unit spacing between successive vector elements in memory.

## 4.2 Access

-
  ```
  int v.size() const
  ```

  Returns the size (length) of `v`.

-
  ```
  T v[int i] const
  T v(int i) const
  Vector<T>::reference v[int i]
  Vector<T>::reference v(int i)
  T v.cref(int i) const
  Vector<T>::reference v.ref(int i)
  ```

  The `[]` and `()` forms are equivalent. Each returns the `i`-th element of `v`. With `A = CStyle`, the first element is `v(0)`, and the last element is `v(n-1)`. With `A = FortranStyle`, they are `v(1)` and `v(n)`.

  If `v` is a `const Vector` or a `ConstVectorView`, then the return type is just the value, not a reference.

  If `v` is a non-`const Vector`, then the return type is a normal reference, `T&`.

  If `v` is a `VectorView`, then the return type is an object, which is an lvalue (i.e. it is assignable), but which may not be `T&`. Specifically, it has the type `VectorView<T>::reference`. For a real-typed `VectorView`, it is just `T&`. But for a complex-typed `VectorView`, the return type is an object that keeps track of the possibility of a conjugation.

  The main difference between the operator forms and `cref` or `ref` is that the latter versions do not check for the validity of the parameter `i`, even when compiling with debugging turned on. Also, `cref` and `ref` always use `CStyle` indexing.

-
  ```
  Vector<T>::iterator v.begin()
  Vector<T>::iterator v.end()
  Vector<T>::const_iterator v.begin() const
  Vector<T>::const_iterator v.end() const
  Vector<T>::reverse_iterator v.rbegin()
  Vector<T>::reverse_iterator v.rend()
  Vector<T>::const_reverse_iterator v.rbegin() const
  Vector<T>::const_reverse_iterator v.rend() const
  ```

  These provide iterator-style access into a `Vector`. The iterators all conform to the standard library's random-access iterator requirements, so you can use these in any standard library routine that takes iterators. For example, to copy from a `std::vector v1` to a `tmv::Vector v2`, you could write:

  ```
  std::copy(v1.begin(),v1.end(),v2.begin());
  ```

  Note: If `v` is a `VectorView`, the iterator types are slightly different from the `Vector` iterators, so you should declare them as `VectorView<T>::iterator`, etc. instead.

- `T* v.ptr()`
  `const T* v.cptr() const`

  These methods return the pointer to the memory location of the first element of the vector. These aren't usually necessary, but sometimes they can be useful for writing optimized code or for meshing with other libraries that need the direct memory access.

- `int v.step() const`

  This returns the step size in memory between elements in the vector. It is usually only required if you are accessing the memory directly with `v.ptr()`.

- `bool v.isconj() const`

  This returns whether or not the elements of the vector are actually the conjugate of the values in memory. Again, this is usually only required if you are accessing the memory directly.

## 4.3 Initialization

There are a number of ways to initialize the elements of a `Vector`. The most straightforward way is to simply set each element individually:

```
for(int i=0; i<N; ++i) {
    v[i] = 5+i; // or whatever
}
```

But this is often inconvenient, especially if you want to assign a particular list of values that does not lend itself to being put in a `for` loop. Of course you can assign each element one line at a time, but that can be a bit unwieldy. So another way is to use a C array initializer, and then copy the values into the vector. For example:

```
double var[5] = { 1.2, 3.2, -9.2, -1, 3.3 };
tmv::Vector<double> v(5);
std::copy(var,var+5,v.begin());
```

This works, but it seems a bit inefficient to use a temporary variable. The better way to do this with TMV is to initialize the `Vector` with the << operator, analogous to the << syntax for `std::ostream`. The idea is that we are sending information into the `Vector`. After the initial <<, the elements are separated by either more <<'s or commas.[6]

```
tmv::Vector<double> v(5);
v << 1.2, 3.2, -9.2, -1, 3.3;
v << 1.2 << 3.2 << -9.2 << -1 << 3.3;
```

There must be precisely as many values as there are elements in the `Vector`, or a `tmv::ReadError` will be thrown, although unlike most `ReadErrors` this is only checked when TMV is in debugging mode (i.e. not compiled with `-DNDEBUG` or `-DTMV_NDEBUG`).

One advantage of this method for initialization is that the values do not need to be simple numbers. Normally the elements in a list initializer need to be numerical literals, or at least to be computable at compile time. With the above syntax, the elements can be variables, or even values returned from functions. Anything that is convertible into the element type of the `Vector` will work.

Also, `v` can be a `VectorView`, not just a `Vector`, so you can use this method to initialize just part of a `Vector`, or a single row or column of a `Matrix`, etc.

[6] I tend to prefer the commas, but using the << operator between each element would match the analogy with `ostream` better.

## 4.4 Resizing

It is possible to change the size of a vector after it has been created using the method `resize`:

```
v.resize(int newsize);
```

This reallocates memory for the new size with *uninitialized* values. There are no correlates to the other constructors, so after a resize, the values should be set directly in some fashion.

Note that there is no automatic resizing if assigning a result that is a different size. For example, the code:

```
tmv::Vector<double> v(10);
tmv::Vector<double> w(20);
// w = ...
v = 5. * w;  // Error: sizes don't match.
```

will throw a `tmv::FailedAssert` exception (c.f. §12.1) (Or if compiled with `-DNDEBUG` or `-DTMV_NDEBUG` to skip TMV's asserts, then it will probably produce a segmentation fault.), rather than resizing `v` to a size of 20.

The reason for this is that such code is more often the result of a bug, rather than intent. So if this code is what you intend, you should write the following:

```
v.resize(20);
v = 5. * w;  // OK: sizes match now.
```

Also, only actual `Vectors` can be resized, not the views that we will discuss in the next section.

## 4.5 Views

A `VectorView<T>` object refers to the elements of some other object, such as a regular `Vector<T>` or `Matrix<T>`, so that altering the elements in the view alters the corresponding elements in the original object. A `VectorView` can have non-unit steps between elements (for example, a view of a column of a row-major matrix). It can also be a conjugation of the original elements, so that

```
tmv::VectorView<double> cv = v.conjugate();
cv(3) = z;
```

would actually set the original element, `v(3)` to `conj(z)`.

Also, we have to keep track of whether we are allowed to alter the original values or just look at them. Since we want to be able to pass these views around, it turns out that the usual `const`-ing doesn't work the way you would want. Thus, there are two objects that are views of a `Vector`: `ConstVectorView` and `VectorView`. The first is only allowed to view, not modify, the original elements. The second is allowed to modify them. This distinction is akin to the `const_iterator` and `iterator` types in the standard template library.

The following methods return views to portions of a `Vector`. If `v` is either a (non-`const`) `Vector` or a `VectorView`, then a `VectorView` is returned. If `v` is a `const Vector` or a `ConstVectorView`, then a `ConstVectorView` is returned.

- `v.subVector(int i1, int i2, int istep=1)`

  This returns a view to a subset of the original vector. `i1` is the first element in the subvector. `i2` is either "one past the end" (C-style) or the last element (Fortran-style) of the subvector according to the attribute parameter of `v`. `istep` is an optional step size. Thus, if you have a `Vector v` of length 10, and you want to multiply the first 3 elements by 2, with C-style indexing, you could write:

  ```
  v.subVector(0,3) *= 2.;
  ```

  To set all the even elements to 0, you could write:

  ```
  v.subVector(0,10,2).setZero();
  ```

And then to output the last 4 elements of `v`, you could write:

```
std::cout << v.subVector(6,10);
```

For Fortran-style indexing, the same steps would be accomplished by:

```
v.subVector(1,3) *= 2.;
v.subVector(1,9,2).setZero();
std::cout << v.subVector(7,10);
```

- `v.reverse()`

  This returns a view whose elements are the same as `v`, but in the reverse order

- `v.conjugate()`

  This returns the conjugate of a `Vector` as a view, so it still points to the same physical elements, but modifying this will set the actual elements in memory to the conjugate of what you set. Likewise, accessing an element will return the conjugate of the value in memory.

- `v.view()`

  Returns a view of a `Vector`. This seems at first like a silly function to have, but if you write a function that takes a mutable `Vector` argument, and you want to be able to pass it views in addition to regular `Vector`s, it is easier to write the function once with a `VectorView` parameter. Then you only need a second function with a `Vector` parameter, which calls the first function using `v.view()` as the argument:

  ```
  double foo(tmv::VectorView<double> v)
  { ... [modifies v] ... }
  double foo(tmv::Vector<double>& v)
  { return foo(v.view()); }
  ```

- `v.cView()`
  `v.fView()`

  Like `view()` but forces the result to have C- or Fortran-style indexing respectively.

- `v.realPart()`
  `v.imagPart()`

  These return views to the real and imaginary parts of a complex `Vector`. Note the return type is a real view in each case:

  ```
  tmv::Vector<std::complex<double> > v(10,std::complex<double>(1,4));
  tmv::VectorView<double> vr = v.realPart();
  tmv::VectorView<double> vi = v.imagPart();
  ```

- `tmv::VectorView<RT> v.flatten()`

  This returns a real view to the real and imaginary elements of a complex `Vector`. The vector `v` is required to have unit step. The returned view has twice the length of `v` and also has unit step.

## 4.6 Functions of a vector

Functions that do not modify the `Vector` can take as their argument either a regular `Vector` or either kind of `View`. You may even pass it an arithmetic expression that is assignable to a `Vector`, and TMV will automatically create the necessary temporary storage for the result. (e.g. `Norm(v1-5*v2)`.)

Functions that modify the `Vector` are only defined for `Vector` and `VectorView`.

Some functions are invalid if T is `int` or `complex<int>` because they require a `sqrt`. If called, these functions typically return `0` if there is a return value. And if the library is compiled with debugging on (`DEBUG=true` in the scons installation), then they will throw a `FailedAssert` exception. The descriptions for such functions will mention if they are invalid for integer types.

### 4.6.1 Non-modifying functions

Each of the following functions can be written in two ways, either as a method or a function. For example, the expressions:

```
double normv = v.norm();
double normv = Norm(v);
```

are equivalent. Also, several of the functions below have multiple equivalent names. For example, `norm1` and `sumAbsElements` are equivalent, so you can use whichever one is clearer to you in your situation. And just to remind you, `RT` refers to the real type associated with `T`. So if `T` is either `double` or `complex<double>`, `RT` would be `double`.

-
  ```
  RT v.norm1() const
  RT Norm1(v)
  RT v.sumAbsElements() const
  RT SumAbsElements(v)
  ```

  The 1-norm of `v`: $||v||_1 = \sum_i |v(i)|$.

  Invalid for `T = complex<int>`.

-
  ```
  RT v.norm2() const
  RT Norm2(v)
  RT v.norm() const
  RT Norm(v)
  ```

  The 2-norm of `v`: $||v||_2 = (\sum_i |v(i)|^2)^{1/2}$. This is the most common meaning for the norm of a vector, so we define the `norm` function to be the same as `norm2`.

  Invalid for `T = int` or `complex<int>`.

-
  ```
  RT v.normSq(const RT scale=1) const
  RT NormSq(v)
  ```

  The square of the 2-norm of `v`: $(||v||_2)^2 = \sum_i |v(i)|^2$. In the method version of this function, you may provide an optional scale factor, in which case the return value is equal to NormSq(scale*v) instead, which can help avoid underflow or overflow problems.

-
  ```
  RT v.normInf() const
  RT NormInf(v)
  RT v.maxAbsElement() const
  RT MaxAbsElement(v)
  ```

The infinity-norm of `v`: $||v||_\infty = \max_i |v(i)|$.

Invalid for `T = complex<int>`.

- `T v.sumElements() const`
  `T SumElements(v)`

  The sum of the elements of `v` $= \sum_i v(i)$.

- `RT v.sumAbs2Elements() const`
  `RT SumAbs2Elements(v)`

  The sum of the "`Abs2`" values of the elements of `v`: $\sum_i |real(v(i))| + |imag(v(i))|$. This is faster than the version using real absolute values for complex vectors and is often just as useful, depending on your purpose. Also this version is valid for `T = complex<int>`, while the normal version is not.

- `T v.maxElement(int* i=0) const`
  `T MaxElement(v)`
  `T v.minElement(int* i=0) const`
  `T MinElement(v)`

  The maximum/minimum element. For complex values, there is no way to define a max or min element, so just the real component of each element is used. The `i` argument is available in the method versions of these function, and it is optional. If it is present (and not 0), then `*i` is set to the index of the max/min element returned.

- `RT v.maxAbsElement(int* i=0) const`
  `RT MaxAbsElement(v)`
  `RT v.minAbsElement(int* i=0) const`
  `RT MinAbsElement(v)`

  The maximum/minimum element by absolute value. $max_i|v(i)|$ or $min_i|v(i)|$. The `i` argument is available in the method versions of these function, and it is optional. If it is present (and not 0), then `*i` is set to the index of the max/min element returned.

  Invalid for `T = complex<int>`.

- `RT v.maxAbs2Element(int* i=0) const`
  `RT MaxAbs2Element(v)`
  `RT v.minAbs2Element(int* i=0) const`
  `RT MinAbs2Element(v)`

  The same as the above functions, but using $|real(v(i))| + |imag(v(i))|$ instead of the normal absolute value if `T` is complex. This is faster than doing the normal absolute value, and for many purposes (such as finding a suitably large value with which to scale a vector), it is just as useful. Also these functions are valid for `T = complex<int>`, while the normal versions are not.

### 4.6.2 Modifying functions

The following functions are methods of both `Vector` and `VectorView`, and they work the same way in the two cases, although there may be speed differences between them. All of these are usually written on a line by themselves. However, they do return the (modified) `Vector`, so you can string them together if you want. For example:

```
v.clip(1.e-10).conjugateSelf().reverseSelf();
```

would first clip the elements at `1.e-10`, then conjugate each element, then finally reverse the order of the elements. (This would probably not be considered very good programming style, however.) Likewise, the expression:

```
foo(v.clip(1.e-10));
```

which would first clip the elements at `1.e-10`, then pass the resulting `Vector` to the function `foo`.

- `v.setZero();`

  Clear the `Vector v`. i.e. Set each element to 0.

- `v.setAllTo(T x);`

  Set each element to the value `x`.

- `v.addToAll(T x)`

  Add the value `x` to each element.

- `v.clip(RT thresh)`

  Set each element whose absolute value is less than `thresh` equal to 0. Note that `thresh` should be a real value even for complex valued `Vector`s.

  Invalid for `T = complex<int>`.

- `v.conjugateSelf()`

  Change each element into its complex conjugate. Note the difference between this and `v.conjugate()`, which returns a view to a conjugated version of `v` without actually changing the underlying data. This function, `v.conjugateSelf()`, does change the underlying data.

- `v.reverseSelf()`

  Reverse the order of the elements. Note the difference between this and `v.reverse()` which returns a view to the elements in reversed order.

- `v.makeBasis(int i)`

  Set all elements to 0, except for `v(i)` = 1.

- `v.swap(int i1, int i2)`

  Swap elements `v(i1)` and `v(i2)`.

-
  ```
  v.sort(Permutation& p, tmv::ADType ad=tmv::Ascend,
        tmv::CompType comp=tmv::RealComp)
  v.sort(tmv::ADType ad=tmv::Ascend,
        tmv::CompType comp=tmv::RealComp)
  ```

  Sorts the vector `v`, optionally returning the corresponding permutation in `p`. If you do not care about the permutation, the second form is slightly more efficient.

  The returned permutation is such that if you have stored the initial vector as `v_i`, then you can reproduce the sort operation with: `v = p * v_i`. More commonly, the permutation is used to effect the same permutation on a different vector or matrix.

The next parameter, `ad`, determines whether the sorted `Vector` will have its elements in ascending or descending order. The possible values are `Ascend` and `Descend`. The default if omitted is to sort in ascending order.

The final parameter, `comp`, determines what component of the elements to use for the sorting. This is especially relevant if T is complex, since complex values are not intrinsically sortable. The possible values are `RealComp`, `AbsComp`, `ImagComp`, and `ArgComp`. Only the first two make sense for non-complex vectors. The default if omitted is to sort the real values of the elements.

`AbsComp` and `ArgComp` are invalid for `T = complex<int>`.

- `Swap(v1,v2)`

  Swap the corresponding elements of `v1` and `v2`. Note that if v1 and/or v2 are views, this does physically swap the data elements, not just some pointers to the data. This is the intuitive meaning of a statement like

  ```
  Swap(m.row(4),m.row(5));
  ```

  Clearly what is desired by that is to swap the actual values, and this is what we actually do.

  However, if `v1` and `v2` are both `tmv::Vector` objects, rather than views, then the swap efficiently swaps the pointers to the data, and so takes $O(1)$ time, rather than $O(N)$.

## 4.7 Arithmetic

### 4.7.1 Operators

All the usual operators work the way you would expect for `Vector`s. For shorthand in the following list, I use `x` for a scalar of type `T` or `RT`, and `v` for a `Vector`. When there are two `Vector`s listed, they may either be both of the same type `T`, or one may be of type `T` and the other of `complex<T>`. Whenever `v` is an lvalue, if may be either a `Vector` or a `VectorView`. Otherwise it can be a `Vector`, either kind of view, or even a more complicated expression that evaluates to a `Vector`.

Also, I use the notation `[+-]` to mean either + or −, since the syntax is generally the same for these operators. Likewise, I use `[*/]` when their syntax is equivalent.

- `v [*/]= x;`

  Scale the `Vector v` by the scalar value `x`.

-
  ```
  v2 = -v1;
  v2 = x * v1;
  v2 = v1 [*/] x;
  ```

  Assign a multiple of one `Vector` to another `Vector`. It is permissible for both vectors to be the same thing (or aliases to the same memory), in which case the `Vector` will be modified in place.

-
  ```
  v2 [+-]= v1;
  v2 [+-]= -v1;
  v2 [+-]= x * v1;
  v2 [+-]= v1 [*/] x;
  ```

  Add one `Vector` (or some multiple of it) to another `Vector`.

-
  ```
  v3 = v1 [+-] v2;
  v3 = x1 * v1 [+-] x2 * v2;
  ```

Assign a sum or difference of two `Vectors` (or scalings thereof) to a third `Vector`.

- `x = v1 * v2;`

  Calculate the inner product of two vectors, which is a scalar. That is, the product is a row vector times a column vector. This is the only case (so far) where the specific row or column orientation of a vector matters. For the others listed above, the left side and the right side are implied to be of the same orientation, but that orientation is otherwise arbitrary. Later, when we get to a matrix times a vector, the orientation of the vector will be inferred from context.

### 4.7.2 Element-by-element product

In addition to the inner product, there are two other kinds of vector products. The outer product will be discussed in §5.7.2 in the section on matrices.

There is also a third kind of product that TMV implements, namely an element-by-element product. This isn't a product that has a normal arithmetic operator associated with it, but it is occasionally useful when you want to multiply each element in a vector by the corresponding element in another vector: $v(i) = v(i) \cdot w(i)$. In TMV, this calculation would be written:

```
v = ElemProd(v,w);
```

The `ElemProd` function returns a composite object that can be used as part of more complicated arithmetic statements. e.g.

```
v += 5*ElemProd(w,v) - 6*w;
```

This is getting slightly ahead of ourselves, but it is worth noting at this point that the element-by-element product for vectors is equivalent to calculations using diagonal matrices. A diagonal matrix times a vector does an element-by-element multiplication. So the above expressions could also be written:

```
v *= DiagMatrixViewOf(w);
v += 5*DiagMatrixViewOf(w)*v - 6*w;
```

respectively. Depending on you preference and the meanings of your vectors, these statements may or may not be clearer as to what you are doing.

### 4.7.3 Delayed evaluation

Each of the above arithmetic operations use delayed evaluation so that the sum or product is not calculated until the storage is known. The equations can even be a bit more complicated without requiring a temporary. Here are some equations that do not require a temporary `Vector` for the calculation:

```
v2 = -(x1*v1 + x2*v2);
v2 += x1*(x2*(-v1));
v2 -= x1*(v1/=x2);
```

TMV does this by creating composite objects that merely keep track of the arithmetic operation that is to be done, and then calls a function to do the calculations when that object is assigned to a `Vector` or `VectorView`. If you try to use one of these composite objects in a way that TMV doesn't know how to deal with directly (e.g. in a more compilicated arithmetic expression), it will instantiate a temporary `Vector` to store the result, and then use that in the next calculation. Usually, this is the appropriate behavior, but in some cases, it might be a bit inefficient, so it might be better to break up the calculation into smaller pieces. For example,

```
v1 += x*(v1+v2+v3) + (x*v3-v4);
```

will produce the correct answer; however, it will use several temporaries. If you want to avoid the temporaries, you would want to break up the calculation into

```
v1 *= x+1;
v1 += x*v2;
v1 += 2*x*v3;
v1 -= v4;
```

The limit to how complicated the right hand side can be without using a temporary is set by the functions that the code eventually calls to perform the calculation, but generally speaking, you can assume that scalings will always be delayed, and other than that you can have a single operator acting on two `Vector`s.

## 4.8 I/O

The simplest output syntax is the usual:

```
os << v;
```

where `os` is any `std::ostream`. The output format is:

```
n ( v(0)  v(1)  v(2)  ...  v(3) )
```

where `n` is the length of the `Vector`.

The same format can be read back using:

```
tmv::Vector<T> v;
is >> v;
```

The `Vector v` is automatically resized to the correct size if necessary based in the size given in the input stream. This is one place where the default constructor (which creates a zero-sized vector) can be useful.

Often, it is convenient to output only those values that aren't very small. This can be done using an equivalent of a stream manipulator:

```
os << tmv::ThreshIO(thresh) << v;
```

which writes as 0 any value smaller (in absolute value) than `thresh`. For real `v` it is equivalent to

```
os << tmv::Vector<T>(v).clip(thresh);
```

but without requiring the temporary `Vector`. For complex `v` it is slightly different, since it separately tests each component of the complex number. So if `thresh` is `1.e-8`, the complex value `(8,7.5624e-12)` would be writtend `(8,0)`.

There is also a compact I/O format which is mediated by the manipulator `tmv::CompactIO()`. For `Vector` it doesn't make much difference, basically just skipping the parentheses and used a little less space between elements:

```
os << tmv::CompactIO() << v;
```

would produce the output:

```
V n v(0) v(1) v(2) ... v(3)
```

The compact I/O is more useful for some of the special matrix varieties where it can skip some zeros that would otherwise be printed and output extra information that might be required to resize the matrix correctly on input.

See §15 for more information about specifying custom I/O styles, including features like using brackets instead of parentheses, or putting commas between elements, or specifying an output precision.

## 4.9 Vector Iterators

We mentioned that the iterators through a `Vector` are:

```
Vector<T>::iterator
Vector<T>::const_iterator
Vector<T>::reverse_iterator
Vector<T>::const_reverse_iterator
```

just like for standard library containers. Sometimes it can be useful to use the underlying iterator types that TMV uses rather than merely the above typedefs. The specific types to which these typedefs refer are:

```
tmv::VIt<T,1,tmv::NonConj>
tmv::CVIt<T,1,tmv::NonConj>
tmv::VIt<T,-1,tmv::NonConj>
tmv::CVIt<T,-1,tmv::NonConj>
```

respectively. They all satisfy the requirements of a STL random-access iterator. `VIt` is a mutable iterator, and `CVIt` is a const iterator.

The second template parameter is the step size between successive elements, if known at compile time. If the step size is not known at compile time, you can use the special value `tmv::Unknown`, and TMV will get the correct value at run time.

The third template parameter must be either `tmv::NonConj` or `tmv::Conj` and indicates whether the values being iterated over are the actual values in memory or their conjugates, respectively.

These types can be worth using explicitly if you want to optimize code that uses iterators of `VectorViews`. This is because their iterators always use `tmv::Unknown` as the step parameter. For complex `VectorViews`, it is even worse, since the possible conjugation is also a run-time variable. These use a different type that allows for the variable conjugation possibility:

```
tmv::VarConjIter<T>
tmv::CVarConjIter<T>
```

If you know that you are dealing with a complex view that is not conjugated, you can convert your iterator into one of the above `VIt` or `CVIt` types, which will be faster, since they won't check the conjugation bit each time. Likewise, if you know that it *is* conjugated, then you can use `tmv::Conj` for the third template parameter above.

Also, if you know the step size between elements at compile time, converting to an iterator with that step size will iterate faster, especially if that step size is known to be 1.

All of these conversions can be done seamlessly through assignments. E.g.

```
if (v.step() == 1)
    for(VIt<float,1,NonConj> it = v.begin(); it != v.end(); ++it)
        (*it) = sqrt(*it);
else
    for(VIt<float,Unknown,NonConj> it = v.begin(); it != v.end(); ++it)
        (*it) = sqrt(*it);
```

## 4.10 Small vectors

For small vectors, it is often the case that you know the size of the vector at compile time. Thus, we provide a class `SmallVector` that takes an additional template argument, `N`, the size of the vector.

All the `SmallVector` routines are included by:

```
#include "TMV_Small.h"
```

`SmallVector` does not inherit from the regular `Vector` class, but it has essentially all the same methods, functions, and arithmetic operators.

### 4.10.1 Constructors

The template argument `N` below is an integer and represent the size of the vector. The final template argument `A` specifies the known attributes. It may be either `tmv::CStyle` or `tmv::FortranStyle` and has the same meanings as for a regular `Vector`. The default is `CStyle` if it is omitted.

- `tmv::SmallVector<T,N,A> v()`

  Makes a `SmallVector` of size `N` with *uninitialized* values.

- `tmv::SmallVector<T,N,A> v(T x)`

  Makes a `SmallVector` with all values equal to `x`.

- `tmv::SmallVector<T,N,A> v1(const Vector<T>& v2)`

  Makes a `SmallVector` from a regular `Vector`.

-
  ```
  tmv::SmallVector<T,N,A> v1(const SmallVector<T2,N,A2>& v2)
  v1 = v2
  ```

  Copy the `SmallVector v2`, which may be of any type `T2` so long as values of type `T2` are convertible into type `T`. The assignment operator has the same flexibility.

-
  ```
  v << v0 , v1 , v2 , v3 ...
  v << v0 << v1 << v2 << v3 ...
  ```

  Initialize the `SmallVector v` with a list of values.

### 4.10.2 Access

The basic access methods are the same as for a regular `Vector`. (See 4.2.) However, since the size is known to the compiler, the inline calculation is able to be a lot faster, often reducing to a trivial memory access.

The various view methods, like `reverse` or `subVector`, do not return a `SmallVector`, so operations with the returned views will not necessarily be done inline. However, you can copy the view back to a "`Small`" object, which will be done inline, so that should be fast.

Also, if the code can determine the size from other objects in the calculation, then it will be done inline. e.g. if `v` is a `SmallVector`, then `v2 = v + v.reverse()` will be done inline, since the first `v` does have the compile-time knowledge about the size of the calculation.

### 4.10.3 Functions

`SmallVector` has exactly the same function methods as the regular `Vector`. (See 4.6.) Likewise, the syntax of the arithmetic is identical. There are only a few methods that are not done inline. First, reading `SmallVector` from a file uses the regular `Vector` I/O methods. Also, the `sort` command for a `SmallVector` just uses the regular `Vector` version.

# 5 Dense rectangular matrices

The `Matrix` class is our dense matrix class. In addition to the data type template parameter (indicated here by `T` as before), there is also a second template parameter that specifies attributes of the `Matrix`. The two attributes that are allowed are:

- `CStyle` or `FortranStyle`
- `ColMajor` or `RowMajor`

With C-style indexing, the upper-left element of an $M \times N$ `Matrix` is `m(0,0)`, the lower-left is `m(M-1,0)`, the upper-right is `m(0,N-1)`, and the lower-right is `m(M-1,N-1)`. Also, methods that take a pair of indices to define a range use the common C convention of "one-past-the-end" for the meaning of the second index. So `m.subMatrix(0,3,3,6)` returns a $3 \times 3$ submatrix.

With Fortran-style indexing, the upper-left element of an $M \times N$ `Matrix` is `m(1,1)`, the lower-left is `m(M,1)`, the upper-right is `m(1,N)`, and the lower-right is `m(M,N)`. Also, methods that take range arguments take the pair of indices to be the actual first and last elements in the range. So `m.subMatrix(1,3,4,6)` returns the same $3 \times 3$ submatrix as given above.

The other attribute specifies which storage order you want to use for the matrix: column-major or row-major. `ColMajor` stores the elements of the first column first, then the second column, and so on. `RowMajor` stores the elements by rows instead.

If you leave off the Attributes parameter, it will use the default values: `CStyle` and `ColMajor`. If you want to specify either `FortranStyle` or `RowMajor`, you simply write (e.g. for double):

```
tmv::Matrix<double,tmv::FortranStyle> m1;
tmv::Matrix<double,tmv::RowMajor> m2;
```

If you want to specify both, then the two values are combined with the bit-wise or operator (`|`):

```
tmv::Matrix<double,tmv::FortranStyle|tmv::RowMajor> m3;
```

Other special matrix types will have the possibility of still more attributes, which act the same way: You only need to specify ones that aren't the default, and multiple values are combined with the `|` operator.

All views of a `Matrix` keep the same indexing style as the original unless you explicitly change it with a cast. You can cast a `MatrixView<T,CStyle>` as a `MatrixView<T,FortranStyle>` and vice versa. (Likewise for `ConstMatrixView`.) This is the only attribute available for a `MatrixView`, since it automatically inherits the storage pattern from the matrix being viewed.

## 5.1 Constructors

Here, `T` is used to represent the data type of the elements of the `Matrix` (e.g. `double`, `complex<double>`, `int`, etc.) and `A` is the Attributes parameter. In all of the constructors the Attributes parameter may be omitted, in which case `CStyle|ColMajor` is assumed.

- `tmv::Matrix<T,A> m()`

  Makes a `Matrix` with zero size. You would normally use the `resize` function later to change the size to something useful.

- `tmv::Matrix<T,A> m(int nrows, int ncols)`

  Makes a `Matrix` with `nrows` rows and `ncols` columns with *uninitialized* values. If extra debugging is turned on (with the compiler flag `-DTMV_EXTRA_DEBUG`), then the values are in fact initialized to 888. This should help you notice when you have neglected to initialize the `Matrix` correctly.

- `tmv::Matrix<T,A> m(int nrows, int ncols, T x)`

  Makes a `Matrix` with `nrows` rows and `ncols` columns with all values equal to `x`

-
  ```
  tmv::Matrix<T,A> m1(const Matrix<T2,A2>& m2)
  m1 = m2
  ```

  Copy the `Matrix m2`, which may be of any type `T2` so long as values of type `T2` are convertible into type `T`. The assignment operator has the same flexibility.

-
  ```
  tmv::MatrixView<T,A> m =
        tmv::MatrixViewOf(T* mm, int nrows, int ncols,
        StorageType stor)
  tmv::ConstMatrixView<T,A> m =
        tmv::MatrixViewOf(const T* mm, int nrows, int ncols,
        StorageType stor)
  tmv::MatrixView<T,A> m =
        tmv::MatrixViewOf(T* mm, int nrows, int ncols,
        int stepi, int stepj)
  tmv::ConstMatrixView<T,A> m =
        tmv::MatrixViewOf(const T* mm, int nrows, int ncols,
        int stepi, int stepj)
  ```

  Makes a `MatrixView` (see §5.5 below) that refers to the exact elements of `mm`, not copying them to new storage.

  The first two versions indicate the ordering of the elements using the normal `StorageType` designation, which must be either `RowMajor` or `ColMajor`.

  The next two versions allow you to provide an arbitrary step through the data in the $i$ and $j$ directions. This means that (for C-style indexing):

  ```
  m(i,j) == *(mm + i*stepi + j*stepj)
  ```

-
  ```
  tmv::MatrixView<T,A> m = RowVectorViewOf(VectorView<T> v)
  tmv::ConstMatrixView<T,A> m = RowVectorViewOf(ConstVectorView<T> v)
  ```

  Makes a view of the `Vector`, which treats it as a $1 \times n$ `Matrix` (i.e. a single row).

-
  ```
  tmv::MatrixView<T,A> m = ColVectorViewOf(VectorView<T> v)
  tmv::ConstMatrixView<T,A> m = ColVectorViewOf(ConstVectorView<T> v)
  ```

  Makes a view of the `Vector`, which treats it as an $n \times 1$ `Matrix` (i.e. a single column).

## 5.2 Initialization

A `Matrix` can be initialized with a comma-delimited list using the << operator, just like with a `Vector`. For example:

```
tmv::Matrix<double> m(3,5);
m << 1.2,  3.2, -9.2, -1.0,  3.3,
     2.0, -2.7,  2.2, -6.5, -7.6,
    -8.2,  3.2, -8.7,  5.1,  1.6;
```

There must be precisely as many values as there are elements in the `Matrix`, or a `tmv::ReadError` will be thrown.

This initialization list is always in row-major order, regardless of the storage order of the matrix in memory. That way, the elements in the list appear in the same order as the real matrix. If the `Matrix` uses `ColMajor` storage, then the assignment is somewhat less efficient, but since the initialization is almost never a time-critical part of the code, it probably isn't worth worrying about.

The list initialization works for a `MatrixView` as well, even if the actual memory elements being assigned to are not contiguous in memory. e.g.:

```
tmv::Matrix<double> m(4,4,0.);
m.subMatrix(0,2,0,2) << // Upper left corner
    1,  3,
    5,  2;
m.subMatrix(2,4,2,4) << // Lower right corner
    2,  3,
    5,  1;
```

Note: As with `Vector` initialization, it is also permissible to use more << operators instead of commas between the elements. I prefer the comma notation, but using << between the elements matches the analogy with `ostream` better, so that is also allowed.

## 5.3 Access

-
  ```
  int m.nrows() const
  int m.ncols() const
  int m.colsize() const
  int m.rowsize() const
  ```

  Returns the size of each dimension of `m`. `nrows()` and `colsize()` are equivalent. Likewise `ncols()` and `rowsize()` are equivalent.

-
  ```
  T m(int i, int j) const
  T m[i][j] const
  T m.cref(int i, int j) const
  Matrix<T>::reference m(int i, int j)
  Matrix<T>::reference m[i][j]
  Matrix<T>::reference m.ref(int i,int j)
  ```

  Returns the `i,j` element of `m`. i.e. the `i`th element in the `j`th column. Or equivalently, the `j`th element in the `i`th row.

  With C-style indexing, the upper-left element of a `Matrix` is `m(0,0)`, the lower-left is `m(nrows-1,0)`, The upper-right is `m(0,ncols-1)`, and the lower-right is `m(nrows-1,ncols-1)`.

  With Fortran-style indexing, these four elements would instead be `m(1,1)`, `m(nrows,1)`, `m(1,ncols)`, and `m(nrows,ncols)`, respectively.

  If `m` is a `const Matrix` or a `ConstMatrixView` then the return type is just the value, not a reference.

  If `m` is a non-`const Matrix`, then the return type is a normal reference `T&`.

  If `m` is a `MatrixView`, then the return type is an lvalue (i.e. it is assignable), but it may or may not be `T&`. It has the type `MatrixView<T>::reference` (which is the same as `VectorView<T>::reference`). It is equal to `T&` for real `MatrixViews`, but is more complicated for complex `MatrixViews` since it needs to keep track of the possibility of conjugation.

The main difference between the operator forms and `cref` or `ref` is that the latter versions do not check for the validity of the parameters `i` and `j`, even when compiling with debugging turned on. Also, `cref` and `ref` always use `CStyle` indexing.

-
  ```
  ConstVectorView<T> m.row(int i) const
  ConstVectorView<T> m.col(int j) const
  VectorView<T> m.row(int i)
  VectorView<T> m.col(int j)
  ```

  Return a view of the `i`th row or `j`th column respectively. If `m` is mutable (either a non-`const Matrix` or a `MatrixView`), then a `VectorView` is returned. Otherwise, a `ConstVectorView` is returned.

-
  ```
  ConstVectorView<T> m.row(int i, int j1, int j2) const
  ConstVectorView<T> m.col(int j, int i1, int i2) const
  VectorView<T> m.row(int i, int j1, int j2)
  VectorView<T> m.col(int j, int i1, int i2)
  ```

  Variations on the above, where only a portion of the row or column is returned.

  For example, with C-style indexing, `m.col(3,2,6)` returns a 4-element vector view containing the elements [`m(2,3), m(3,3), m(4,3), m(5,3)`].

  With Fortran-style indexing, the same elements are returned by `m.col(4,3,6)`. (And these elements would be called: [`m(3,4), m(4,4), m(5,4), m(6,4)`].)

-
  ```
  ConstVectorView<T> m.diag() const
  ConstVectorView<T> m.diag(int i) const
  ConstVectorView<T> m.diag(int i, int k1, int k2) const
  VectorView<T> m.diag()
  VectorView<T> m.diag(int i)
  VectorView<T> m.diag(int i, int k1, int k2)
  ```

  Return the diagonal or one of the sub- or super-diagonals. This first one returns the main diagonal. For the second and third, `i=0` refers to the main diagonal; `i>0` are the super-diagonals; and `i<0` are the sub-diagonals. The last version is equivalent to the expression `m.diag(i).subVector(k1,k2)`.

-
  ```
  ConstVectorView<T> m.subVector(int i, int j, int istep, int jstep,
        int size) const
  VectorView<T> m.subVector(int i, int j, int istep, int jstep, int size)
  ```

  If the above methods aren't sufficient to obtain the `VectorView` you need, this function is available, which returns a view through the `Matrix` starting at `m(i,j)`, stepping by `(istep,jstep)` between elements, for a total of `size` elements. For example, the diagonal from the lower-left to the upper-right of an $n \times n$ `Matrix` would be obtained by: `m.subVector(n-1,0,-1,1,n)` for C-style or `m.subVector(n,1,-1,1,n)` for Fortran-style.

-
  ```
  ConstVectorView<T> m.constLinearView() const
  VectorView<T> m.linearView()
  bool m.canLinearize()
  ```

  These return a view to the elements of a `Matrix` as a single vector. They are always allowed for an actual `Matrix`. For a `MatrixView` (or `ConstMatrixView`), they are only allowed if all of the elements in the view are in one contiguous block of memory. The helper function `m.canLinearize()` returns whether or not the first two methods will work.

- `Matrix<T>::rowmajor_iterator m.rowmajor_begin()`
  `Matrix<T>::rowmajor_iterator m.rowmajor_end()`
  `Matrix<T>::const_rowmajor_iterator m.rowmajor_begin() const`
  `Matrix<T>::const_rowmajor_iterator m.rowmajor_end() const`
  `Matrix<T>::colmajor_iterator m.colmajor_begin()`
  `Matrix<T>::colmajor_iterator m.colmajor_end()`
  `Matrix<T>::const_colmajor_iterator m.colmajor_begin() const`
  `Matrix<T>::const_colmajor_iterator m.colmajor_end() const`

  These iterators traverse the elements of the `Matrix` in either row-major or column-major order. The iterators are only guaranteed to be consistent with the STL requirements for forward iterator. However, for a regular `Matrix` (i.e. not a `MatrixView`) the iterators in the same direction as the matrices storage order will be a random-access iterator. In fact, it will be a `VIt<T,1,NonConj>` (c.f. §4.9).

- `T* m.ptr()`
  `const T* m.cptr() const`

  These methods return the pointer to the memory location of the upper-left element of the matrix ($m(0,0)$). These aren't usually necessary, but sometimes they can be useful for writing optimized code or for meshing with other libraries that need the direct memory access.

- `int m.stepi() const`
  `int m.stepj() const`

  This returns the step size in memory between the elements of a column and of a row, respectively. In other words, the step in the "down" direction along a column is `stepi()`, and the step to the "right" along a row is `stepj()`. They are usually only required if you are accessing the memory directly with `m.ptr()`.

- `bool m.isconj() const`

  This returns whether or not the elements of the matrix are actually the conjugate of the values in memory. Again, this is usually only required if you are accessing the memory directly.

  `bool m.isrm() const`
  `bool m.iscm() const`

  These are really just convenience functions that return whether a matrix is `RowMajor` or `ColMajor`, and are equivalent to `(m.stepj()==1)` and `(m.stepi()==1)` respectively.

## 5.4 Resizing

It is possible to change the size of a matrix after it has been created using the method `resize`:

```
m.resize(int new_nrows, int new_ncols);
```

This reallocates memory for the new size with *uninitialized* values. There are no correlates to the other constructors, so after a resize, the values should be set directly in some fashion.

Note that there is no automatic resizing if assigning a result that is a different size. For example, the code:

```
tmv::Matrix<double> m(10,10);
tmv::Matrix<double> q(20,15);
// q = ...
m = q.transpose();
```

will throw a `tmv::FailedAssert` exception (c.f. §12.1) (Or if compiled with `-DNDEBUG` or `-DTMV_NDEBUG` to skip TMV's asserts, then it will probably produce a segmentation fault.), rather than resizing `m` to a size of 15 × 20.

The reason for this is that such code is more often the result of a bug, rather than intent. So if this code is what you intend, you should write the following:

```
m.resize(15,20);
m = q.transpose()
```

Also, only actual `Matrix`es can be resized, not the views that we will discuss in the next section.

## 5.5 Views

A `MatrixView` object refers to some or all of the elements of a regular `Matrix`, so that altering the elements in the view alters the corresponding elements in the original object. A `MatrixView` can be either row-major, column-major, or neither. That is, the view can span a `Matrix` with non-unit steps in both directions. It can also be a conjugation of the original elements.

There are two view classes for a `Matrix`: `ConstMatrixView` and `MatrixView`. The first is only allowed to view, not modify, the original elements. The second is allowed to modify them.

The following methods return views to portions of a `Matrix`. If `m` is either a (non-`const`) `Matrix` or a `MatrixView`, then a `MatrixView` is returned. If `m` is a `const Matrix` or a `ConstMatrixView`, then a `ConstMatrixView` will be returned.

- `m.subMatrix(int i1, int i2, int j1, int j2)`
  `m.subMatrix(int i1, int i2, int j1, int j2, int istep, int jstep)`

  This returns a view to a submatrix contained within the original matrix.

  If `m` uses C-style indexing, the upper-left corner of the returned view is `m(i1,j1)`, the lower-left corner is `m(i2-1,j1)`, the upper-right corner is `m(i1,j2-1)`, and the lower-right corner is `m(i2-1,j2-1)`.

  If `m` uses Fortran-style indexing, the upper-left corner of the view is `m(i1,j1)`, the lower-left corner is `m(i2,j1)`, the upper-right corner is `m(i1,j2)`, and the lower-right corner is `m(i2,j2)`.

  The second version allows for non-unit steps in the two directions. To set a `Matrix` to be a checkerboard of 1's, you could write (for C-style indexing):

  ```
  tmv::Matrix<int> board(8,8,0)
  board.subMatrix(0,8,0,8,2,2).setAllTo(1);
  board.subMatrix(1,9,1,9,2,2).setAllTo(1);
  ```

  For Fortran-style indexing, the same thing would be accomplished by:

  ```
  tmv::Matrix<int,tmv::FortranStyle> board(8,8,0)
  board.subMatrix(1,7,1,7,2,2).setAllTo(1);
  board.subMatrix(2,8,2,8,2,2).setAllTo(1);
  ```

- `m.rowRange(int i1, int i2)`
  `m.colRange(int j1, int j2)`

  Since pulling out a bunch of contiguous rows or columns is a common submatrix use, we provide these functions. They are shorthand for

  ```
  m.subMatrix(i1,i2,0,ncols)
  m.subMatrix(0,nrows,j1,j2)
  ```

respectively. (For Fortran-style indexing, replace the 0 with a 1.)

- `m.rowPair(i1,i2)`
  `m.colPair(i1,i2)`

  Another common submatrix is to select a pair of rows or columns, not necessarily adjacent to each other. These return a $2 \times N$ and an $M \times 2$ submarix respectively with just the rows or columns indicated.

  While it may not be overly enlightening, we also note that they can be considered short hand for:

  ```
  m.subMatrix(i1,i2+(i2-i1),0,ncols,i2-i1,1)
  m.subMatrix(0,nrows,j1,j2+(j2-j1),1,j2-j1)
  ```

  in the C-style notation or

  ```
  m.subMatrix(i1,i2,1,ncols,i2-i1,1)
  m.subMatrix(1,nrows,j1,j2,1,j2-j1).
  ```

  in the Fortran-style notation.

- `m.transpose()`
  `m.conjugate()`
  `m.adjoint()`

  These return the transpose, conjugate, and adjoint (aka conjugate-transpose) of a `Matrix`. They point to the same physical elements as the original matrix, so modifying these will correspondingly modify the original matrix.

  Note that some people define the adjoint of a matrix as the determinant times the inverse. This combination is also called the adjugate or the cofactor matrix. It is not the same as our `m.adjoint()`. What we call the adjoint is usually written as $m^\dagger$, or variously as $m^H$ or $m^*$, and is sometimes referred to as the hermitian conjugate or (rarely) tranjugate. This definition of the adjoint seems to be the more modern usage. Older texts tend to use the other definition. However, if this is confusing for you, it may be clearer to explicitly write out `m.conjugate().transpose()`, which will not produce any efficiency reduction in your code compared with using `m.adjoint()` (assuming your compiler inlines these methods properly).

- `m.view()`

  Returns a view of a `Matrix`. As with the `view()` function for a `Vector`, it is mostly useful for passing a `Matrix` to a function that takes a `MatrixView` argument. This lets you convert the first into the second.

- `m.cView()`
  `m.fView()`

  Like `view()` but forces the result to have C- or Fortran-style indexing respectively.

- `m.realPart()`
  `m.imagPart()`

  These return views to the real and imaginary parts of a complex `Matrix`. Note the return type is a real view in each case:

  ```
  tmv::Matrix<std::complex<double> > m(10,std::complex<double>(1,4));
  tmv::MatrixView<double> mr = m.realPart();
  tmv::MatrixView<double> mi = m.imagPart();
  ```

- `m.upperTri(DiagType dt = NonUnitDiag)`
  `m.lowerTri(DiagType dt = NonUnitDiag)`
  `m.unitUpperTri()`
  `m.unitLowerTri()`

  These return an `UpperTriMatrixView` or a `LowerTriMatrixView` which views either the upper triangle or the lower triangle of a square `Matrix`. If `m` has more rows than columns, then only `UpperTri` is valid, since the portion below the diagonal is not triangular. Likewise, if `m` has more columns than rows, then only `LowerTri` is valid.

  In first two cases, you may provide an optional parameter `dt`, which declares whether the diagonal elements are treated as all 1s (`dt = UnitDiag`) or as their actual values (`dt = NonUnitDiag`). See §7 for more details about this parameter and triangular matrices in general.

  The latter two versions are equivalent to `m.upperTri(UnitDiag)` and `m.lowerTri(UnitDiag)` respectively. However, future versions of TMV (specifically version 0.90, which is in development) will be able to exploit the compile-time knowledge of `dt` to make this more efficient.

- `tmv::VectorView<T> m.linearView()`
  `tmv::ConstVectorView<T> m.constLinearView()`
  `bool m.canLinearize()`

  These return a view to the elements of a `Matrix` as a single vector. It is only valid if all the elements are stored contiguously in memory. It is always allowed for an actual `Matrix`. For a `MatrixView` (or `ConstMatrixView`), it is only allowed if all of the elements in the view are in one contiguous block of memory. The helper function `m.canLinearize()` returns whether or not the first two methods will work.

## 5.6 Functions of a matrix

Functions that do not modify the `Matrix` can take as their argument either a regular `Matrix` or either kind of `View`. You may even pass it an arithmetic expression that is assignable to a `Matrix`, and TMV will automatically create the necessary temporary storage for teh result. (e.g. `Norm(m1-m2*m3)`.) Functions that modify the `Matrix` are only defined for `Matrix` and `MatrixView`.

Some functions are invalid if T is `int` or `complex<int>` because they require a `sqrt`, `log` or division. If called, these functions typically return `0` if there is a return value. And if the library is compiled with debugging on (using the SCons flag `DEBUG=true`), then they will throw a `FailedAssert` exception. The descriptions for such functions will mention if they are invalid for integer types.

### 5.6.1 Non-modifying functions

Each of the following functions can be written in two ways, either as a method or a function. For example, the expressions `m.norm()` and `Norm(m)` are equivalent. As a reminder, `RT` refers to the real type associated with `T`. In other words, `T` is either the same as `RT` or it is `std::complex<RT>`.

- `RT m.norm1() const`
  `RT Norm1(m)`

  The 1-norm of `m`: $||m||_1 = \max_j(\sum_i |m(i,j)|)$.

  Invalid for `T = complex<int>`.

- `RT m.norm2() const`
  `RT Norm2(m)`

The 2-norm of `m`: $||m||_2$ = the largest singular value of $m$, which is also the square root of the largest eigenvalue of $(m^\dagger m)$.

If you are using `SV` for division (using `DivideUsing(tmv::SV)`), and the decomposition has already been performed, then this function will be very fast, since it can use the existing decomposition.

If you are not using `SV` for division, then this function is fairly expensive, scaling as $O(N^2 log(N))$, rather than $O(N^2)$ like most other norms. However, it won't bother accumulating the singular vectors, so it will be significantly faster than doing the full decomposition that would be used for division, which takes $O(N^3)$ time.

Invalid for `T = int` or `complex<int>`.

-
  ```
  RT m.normInf() const
  RT NormInf(m)
  ```

  The infinity-norm of `m`: $||m||_\infty = \max_i(\sum_j |m(i,j)|)$.

  Invalid for `T = complex<int>`.

-
  ```
  RT m.normF() const
  RT NormF(m)
  RT m.norm() const
  RT Norm(m)
  ```

  The Frobenius norm of `m`: $||m||_F = (\sum_{i,j} |m(i,j)|^2)^{1/2}$.

  This is the most common meaning for the norm of a matrix, so we define the `norm` function to be the same as `normF`.

  Invalid for `T = int` or `complex<int>`.

-
  ```
  RT m.normSq(RT scale=1) const
  RT NormSq(m)
  ```

  The square of the Frobenius norm of `m`: $(||m||_F)^2 = \sum_{i,j} |m(i,j)|^2$.

  In the method version of this function, you may provide an optional scale factor, in which case the return value is equal to `NormSq(scale*v)`, which can help avoid underflow or overflow problems.

-
  ```
  RT m.maxAbsElement() const
  RT MaxAbsElement(m)
  ```

  The element of `m` with the maximum absolute value: $||m||_\Delta = \max_{i,j} |m(i,j)|$.

  Invalid for `T = complex<int>`.

-
  ```
  RT m.maxAbs2Element() const
  RT MaxAbs2Element(m)
  ```

  The same as `maxAbsElement`, but using $|real(m(i,j))| + |imag(m(i,j))|$ instead of the normal absolute value for complex matrices.

-
  ```
  T m.trace() const
  T Trace(m)
  ```

  The trace of `m`: $\text{Tr}(m) = \sum_i m(i,i)$.

- `T m.sumElements() const`
  `T SumElements(m)`

  The sum of all elements of `m`: $\sum_{i,j} m(i,j)$.

- `RT m.sumAbsElements() const`
  `RT SumAbsElements(m)`

  The sum of the absolute values of all elements of `m`: $\sum_{i,j} |m(i,j)|$.

  Invalid for `T = complex<int>`.

- `RT m.sumAbs2Elements() const`
  `RT SumAbs2Elements(m)`

  The sum of the "`Abs2`" values of all elements of `m`: $\sum_{i,j} |real(m(i,j))| + |imag(m(i,j))|$.

- `T m.det() const`
  `T Det(m)`

  The determinant of `m`, $\det(m)$. For speed issues regarding this function, see §5.8 below on division.

- `RT m.logDet(T* sign=0) const`
  `RT LogDet(m)`

  The log of the absolute value of the determinant of `m`. If the optional argument `sign` is provided (possible in the method version only), then on output `*sign` records the sign of the determinant. See 5.8.6 for more details.

  Invalid for `T = int` or `complex<int>`.

- `bool m.isSingular() const`

  Return whether `m` is singular, i.e. $\det(m) = 0$. (Singular matrices are discussed in more detail in §5.8.8.)

- `RT m.condition() const`
  `RT m.doCondition() const`

  The condition (technically the 2-condition) is the ratio of the largest singular value of $m$ to the smallest.

  Like `norm2`, this function requires a singular value decomposition to be performed, so it can be fairly expensive if you have not already performed an SV decomposition of `m`. So the first version outputs a warning to `stdout` if there is no SV decomposition already set for the matrix.

  And as with `norm2`, you can bypass the warning by calling `doCondition()` instead, which will do the necessary SV decomposition without any warnings. Also, see §12.5 for how to change the warning behavior of TMV.

  Invalid for `T = int` or `complex<int>`.

- `tmv::Matrix<T> minv = m.inverse()`
  `tmv::Matrix<T> minv = Inverse(m)`
  `void m.makeInverse(Matrix<T>& minv)`

Set `minv` to the inverse of `m`.

If `m` is not square, then `minv` is set to the pseudo-inverse, or an approximate pseudo-inverse. If `m` is singular, then an error may result, or the pseudo-inverse may be returned, depending on the division method specified for the matrix. See §5.8.4 and §5.8.8 on pseudo-inverses and singular matrices for more details.

Note that the first two forms do not actually require a temporary (despite appearances), so they are just as efficient as the third version. This is because `m.inverse()` actually returns a composite object that delays the calculation of the inverse until there is a place to store the result.

Invalid for `T = int` or `complex<int>`.

- `void m.makeInverseATA(Matrix<T>& cov) const`

  Set `cov` to be $(m^\dagger m)^{-1}$ if `m` has at least as many rows as columns.

  If `m` has more rows than columns, then using it to solve a system of equations really amounts to finding the least-square solution, since there is (typically) no exact solution. When you do this, `m` is known as the "design matrix" of the system, and is commonly called $A$. Solving $Ax = b$ gives $x$ as the least-square solution. And the covariance matrix for the solution vector $x$ is $\Sigma = (A^\dagger A)^{-1}$. It turns out that computing this matrix is generally easy to do once you have performed the decomposition needed to solve for $x$ (either a QR or SV decomposition - see §5.8.3). Thus this function is provided, which sets the argument `cov` to the equivalent of `Inverse(m.adjoint()*m)`, but generally does so much more efficiently than doing this explicitly, and also probably more accurately.

  If `m` has more columns than rows, then the return value will instead be $(mm^\dagger)^{-1}$.

  Invalid for `T = int` or `complex<int>`.

### 5.6.2 Modifying functions

The following functions are methods of both `Matrix` and `MatrixView`, and they work the same way for each. As with the `Vector` modifying functions, these all return a reference to the newly modified `Matrix`, so you can string them together if you want.

- `m.setZero()`

  Set all elements to 0.

- `m.setAllTo(T x)`

  Set all elements to the value `x`.

- `m.addToAll(T x)`

  Adds the value `x` to all elements of `m`.

- `m.clip(RT thresh)`

  Set each element whose absolute value is less than `thresh` equal to 0. Note that `thresh` should be a real value even for complex valued `Matrix`es.

  Invalid for `T = complex<int>`.

- `m.setToIdentity(T x = 1)`

  Set `m` to `x` times the identity matrix. If the argument `x` is omitted, it is taken to be `1`, so `m` is set to the identity matrix. This is equivalent to `m.setZero().diag().setAllTo(x)`.

- `m.conjugateSelf()`

  Conjugate each element. Note the difference between this and `m.conjugate()`, which returns a <u>view</u> to the conjugate of `m` without actually changing the underlying data. Contrariwise, `m.conjugateSelf()` does change the underlying data.

- `m.transposeSelf()`

  Transpose the `Matrix`. Note the difference between this and `m.transpose()`, which returns a <u>view</u> to the transpose without actually changing the underlying data.

-
  ```
  m.swapRows(int i1, int i2)
  m.swapCols(int j1, int j2)
  ```

  Swap the corresponding elements of two rows or two columns.

- `Swap(m1,m2)`

  Swap the corresponding elements of `m1` and `m2`. If `m1` and `m2` are both regular `Matrix` objects, then this only takes $O(1)$ time, since it just swaps pointers to the data. However if either of them are a `MatrixView`, then it has to directly swap the corresponding data elements, so it takes $O(N^2)$ time.

## 5.7 Arithmetic

### 5.7.1 Basic operators

We'll start with the simple operators that require little explanation aside from the notation: `x` is a scalar, `v` is a `Vector`, and `m` is a `Matrix`. As a reminder, the notation `[+-]` is used to indicate either + or −. Likewise for `[*/]`.

- `m [*/]= x`

  Scale the `Matrix m` by the scalar value `x`.

-
  ```
  m2 = -m1
  m2 = x * m1
  m2 = m1 [*/] x
  ```

  Assign a multiple of one `Matrix` to another `Matrix`. It is permissible for both vectors to be the same thing (or aliases to the same memory), in which case the `Vector` will be modified in place.

-
  ```
  m2 [+-]= m1
  m2 [+-]= -m1
  m2 [+-]= x * m1
  m2 [+-]= m1 [*/] x
  ```

  Add one `Matrix` (or some multiple of it) to another `Matrix`.

-
  ```
  m3 = m1 [+-] m2
  m3 = x1 * m1 [+-] x2 * m2
  ```

  Assign a sum or difference of two `Matrix`es (or scalings thereof) to a third `Matrix`.

-
  ```
  v2 = m * v1
  v2 = x * m * v1
  v2 [+-]= m * v1
  v2 [+-]= x * m * v1
  ```

  Multiply a `Matrix` by a `Vector`, possibly with a scaling. The `Vectors` are taken to be oriented as column vectors.

-
  ```
  v2 = v1 * m
  v2 = x * v1 * m
  v2 [+-]= v1 * m
  v2 [+-]= x * v1 * m
  v *= m
  ```

  Multiply a `Vector` by a `Matrix`, possibly with a scaling. The `Vectors` are taken to be oriented as row vectors. For the final line, TMV will automatically create the necessary temporary `Vector`.

-
  ```
  m3 = m1 * m2
  m3 = x * m1 * m2
  m3 [+-]= m1 * m2
  m3 [+-]= x * m1 * m2
  m2 *= m1
  ```

  Multiply two `Matrix`es, possibly with a scaling. For the final line, TMV will automatically create the necessary temporary `Matrix`.

-
  ```
  m2 = m1 [+-] x
  m2 = x [+-] m1
  m [+-]= x
  m = x
  ```

  It is sometimes convenient to be able to treat scalars as the scalar times an identity matrix. So these operations are allowed and use that convention taking `x` to be `x` times an identity matrix the same size as the other `Matrix`es in the equation.

  For example, you could check if a matrix is numerically close to the identity matrix with:

  ```
  if (Norm(m-1.) < 1.e-8) { ... }
  ```

### 5.7.2 Outer products

For the product of two vectors, there are two orientation choices that make sense mathematically. It could mean a row vector times a column vector, which is called the inner product. Or it could mean a column vector times a row vector, which is called the outer product.[7] We chose to let `v1*v2` indicate the inner product, since this is far more common. For the outer product, we use a different symbol:

```
m = v1^v2
```

One problem with this choice is that the order of operations in C++ for `^` is not the same as for `*`. So, one needs to be careful when combining it with other calculations. For example

```
m2 = m1 + v1^v2   // ERROR!
```

[7] We also discussed a third possibility in §4.7.2 of multiplying the corresponding elements.

will give a compile time error indicating that you can't add a `Matrix` and a `Vector`, because the operator + has higher precedence in C++ than ^. So you need to write:

```
m2 = m1 + (v1^v2)
```

### 5.7.3 Element-by-element product

Occasionally, it is useful to multiply the corresponding elements of two matrices rather than use the normal meaning of matrix multiplication. This can be done in TMV using the function `ElemProd`:

```
m = ElemProd(m1,m2);
```

This does the calculation `m(i,j) = m1(i,j) * m2(i,j)` for each element in the matrix. Of course, one of the matrices on the right can be the same as the matrix on the left. So you can do `m(i,j) *= m2(i,j)` by writing

```
m = ElemProd(m,m2);
```

The return value of the `ElemProd` function is a composite object, so it can be used in more complicated arithmetic expressions just like any other arithmetic operator.

### 5.7.4 Delayed evaluation

As with `Vectors`, the above arithmetic operations delay the evaluation of any arithmetic expression until we have a place to store the result. So `m*v` returns an object that can be assigned to a `Vector` or `VectorView`, but hasn't performed the calculation yet. Likewise `v^v` and `m*m` return objects that can be assigned to a `Matrix` or `MatrixView`.

Generally, if you limit the calculation to a single operator between two `Matrix` or `Vector` objects, possibly with some scalings of these objects, then no temporary will be required. Here are some examples that do not require temporaries:

```
v1 += x*m*v2;
m -= (x1*v1) ^ (v2*=x2)
m1 = m2 * (x*m3)
```

If you make the equation more complicated, TMV may create one or more temporary objects to store intermediate values. Depending on your needs, this might be inefficient, so you might want to split up complicated expressions into smaller pieces that don't require temporaries. However, no matter how complicated you make the expression, TMV should always calculate the correct result.

## 5.8 Matrix division

One of the main things people often want to do with a matrix is use it to solve a set of linear equations. The set of equations can be written as a single matrix equation:

$$Ax = b$$

where $A$ is a matrix and $x$ and $b$ are vectors. $A$ and $b$ are known, and one wants to solve for $x$. Sometimes there are multiple systems to be solved using the same coefficients, in which case $x$ and $b$ become matrices as well.

Note – essentially all of the functions and operations in this section are invalid for integer-typed matrices. If $A$ and $b$ have integer values, it is not necessarily true that $x$ will too. So if you want to do this kind of operation on `int` matrices, you should first copy them to either `float` or `double` before doing any division operation.

The exception to this is determinants. The determinant of a matrix with integer values is necessarily an integer too. So TMV uses a special algorithm that does not involve division to calculate the determinant of `int` and `complex<int>` matrices[8].

### 5.8.1 Operators

Using the TMV classes, one would solve this equations by writing simply:

```
x = b / A
```

Note that this really means $x = A^{-1}b$, which is different from $x = bA^{-1}$. Writing the matrix equation as we did ($Ax = b$) is much more common than writing $xA = b$, so "left-division" is correspondingly much more common than "right-division". Therefore, it makes sense to use left-division for our definition of the `/` operator.

However, we do allow for the possibility of wanting to right-multiply a vector by $A^{-1}$ (in which case the vector is inferred to be a row-vector). We designate this operation by:

```
x = b % A
```

which means $x = bA^{-1}$.

Given this explanation, the rest of the division operations should be self-explanatory, where we use the notation `[/%]` to indicate that either `/` or `%` may be used with the above difference in meaning:

```
v2 = v1 [/%] m
m3 = m1 [/%] m2
v [/%]= m
m2 [/%]= m1
m2 = x [/%] m1
```

If you feel uncomfortable using the `/` and `%` symbols, you can also explicitly write things like

```
v2 = m.inverse() * v1
v3 = v1 * m.inverse()
```

which delay the calculation in exactly the same way that the above forms do. These forms do not ever explicitly calculate the matrix inverse, since this is not (numerically) a good way to perform these calculations. Instead, the appropriate decomposition (see §5.8.3) is used to calculate `v2` and `v3`.

### 5.8.2 Least-square solutions

If $A$ is not square, then the equation $Ax = b$ does not have a unique solution. If $A$ has more rows than columns, then there is in general no solution. And if $A$ has more columns than rows, then there are (usually) an infinite number of solutions.

The former case is more common and represents an overdetermined system of equations. In this case, one is not looking for an exact solution for $x$, but rather the value of $x$ that minimizes $||b - Ax||_2$. This is the meaning of the least-square solution, and is the value returned by `x = b/A` for the TMV classes.

The matrix $A$ is called the "design" matrix. For example, assume you are doing a simple quadratic fit to some data $(x_i, y_i)$, and the model your are fitting for can be written as $y = c + dx + ex^2$. Furthermore, assume that each $y_i$ value has a measurement error of $s_i$. Then the rows of $A$ should be set to: $\begin{pmatrix} 1/s_i & x_i/s_i & x_i^2/s_i \end{pmatrix}$. The corresponding element of the vector $b$ should be $\begin{pmatrix} y_i/s_i \end{pmatrix}$. It is easily verified that $||b - Ax||_2^2$ is the normal $\chi^2$ expression. The solution returned by `x = b/A` would then be the least-square fit solution: $\begin{pmatrix} c & d & e \end{pmatrix}$, which would be the solution which minimizes $\chi^2$.

[8] Well, actually there are division operations, but only when they are guaranteed to result in exact integer results without rounding. So it is safe to perform on integer values.

The underdetermined case is not so common, but it can still be defined reasonably. As mentioned above, there are usually infinitely many solutions to such an equation, so the value returned by `x = b/A` in this case is the value of `x` that satisfies the equation and has minimum 2-norm, $||x||_2$.

When you have calculated a least-square solution for `x`, it is common to want to know the covariance matrix of the returned values. It turns out that this matrix is $(A^\dagger A)^{-1}$. Furthermore, once you have calculated the decomposition needed for the division, it is quite easy to do this calculation as well. So we provide the routine

```
A.makeInverseATA(Matrix<T>& cov)
```

to perform the calculation efficiently. (Make sure you save the decomposition with `A.saveDiv()` – see §5.8.5 for more about this.)

### 5.8.3 Decompositions

There are quite a few ways to go about solving the equations written above. The more efficient ways involve decomposing $A$ into a product of matrices with special structures or properties. You can select which decomposition to use with the method:

```
m.divideUsing(tmv::DivType dt)
```

where `dt` can be any of `{tmv::LU, tmv::QR, tmv::QRP, tmv::SV}`. If you do not specify which decomposition to use, `LU` is the default for square matrices, and `QR` is the default for non-square matrices.

1. **LU decomposition**: (`dt = tmv::LU`) $A = PLU$, where $L$ is a lower-triangle matrix with all 1's along the diagonal, $U$ is an upper-triangle matrix, and $P$ is a permutation matrix. This decomposition is only available for square matrices.

2. **QR decomposition**: (`dt = tmv::QR`) $A = QR$ where $Q$ is a unitary matrix and $R$ is an upper-triangle matrix. (Note: real unitary matrices are also known as orthogonal matrices.) A unitary matrix is such that $Q^\dagger Q = I$.

   If $A$ has dimensions $M \times N$ with $M > N$, then $R$ has dimensions $N \times N$, and $Q$ has dimensions $M \times N$. In this case, $Q$ will only be column-unitary. That is $QQ^\dagger \neq I$.

   If $M < N$, then $A^T$ is actually decomposed into $QR$.

3. **QRP decomposition**: (`dt = tmv::QRP`) $A = QRP$ where $Q$ is unitary, $R$ is upper-triangle, and $P$ is a permutation. This decomposition is somewhat slower than a simple QR decomposition, but it is numerically more stable if $A$ is singular, or nearly so. (Singular matrices are discussed in more detail in §5.8.8.)

   There are two slightly different algorithms for doing a QRP decomposition that I call Strict QRP and Loose QRP. The Strict QRP algorithm will make the diagonal elements of $R$ be strictly decreasing (in absolute value) from upper-left to lower-right. This is the usual formulation that you might see described in books on linear algebra algorithms.

   The Loose QRP algorithm is a modification to the usual algorithm that only guarantees that there will be no diagonal element of $R$ below and to the right of one which is much smaller in absolute value. Here "much" means the ratio will be at most $\epsilon^{1/4}$, where $\epsilon$ is the machine precision for the type in question. This restriction is almost always sufficient to make the decomposition useful for singular or nearly singular matrices, and it is much faster than the Strict QRP algorithm.

   The default algorithm is the Loose QRP algorithm. If you want to switch to the Strict QRP algorithm, use the function

   ```
   tmv::UseStrictQRP();
   ```

   To return back to the Loose QRP algorithm, use

```
tmv::UseStrictQRP(false);
```

And finally, you can query whether TMV is currently set to use Strict QRP with

```
bool tmv::QRP_IsStrict();
```

4. **Singular value decomposition**: (`dt = tmv::SV`) $A = USV^\dagger$ where $U$ is unitary (or column-unitary if $M > N$), $S$ is diagonal and real, and $V$ is unitary (or column-unitary if $M < N$). The values of $S$ will be such that all the values will be non-negative and will decrease along the diagonal. The singular value decomposition is most useful for matrices that are singular or nearly so. We will discuss this decomposition in more detail in §5.8.8 on singular matrices below.

### 5.8.4 Pseudo-inverse

If `m` is not square, then `m.inverse()` should return what is called the pseudo-inverse. If `m` has more rows than columns, then `m.inverse() * m` is the identity matrix, but `m * m.inverse()` is not an identity. If `m` has more columns than rows, then the opposite holds.

Here are some features of the pseudo-inverse (we use $X$ to represent the pseudo-inverse of $M$):

$$\begin{aligned} MXM &= M \\ XMX &= X \\ (MX)^T &= MX \\ (XM)^T &= XM \end{aligned}$$

For singular square matrices, one can also define a pseudo-inverse with the same properties.

In the first sentence of this section, I used the word "should". This is because the different decompositions calculate the pseudo-inverse differently and result in slightly different answers. For QR or QRP, the matrix returned by `m.inverse()` for non-square matrices isn't quite correct. When $M$ is not square, but is also not singular, then $X$ will satisfy the first three of the above equations, but not the last one. With the SV decomposition, however, $X$ is the true pseudo-inverse and all four equations are satisfied.

For singular matrices, QR will fail to give a good pseudo-inverse (and may throw the `tmv::Singular` exception - c.f. §12), QRP will be close to correct (again failing only the last equation) and will not throw an exception. SV will again be correct.

### 5.8.5 Efficiency issues

Let's say you compute a matrix decomposition for a particular division calculation, and then later want to use it again for a different right hand side:

```
x = b / m;
[...]
y = c / m;
```

Ideally, the code would just use the same decomposition that had already been calculated for the `x` assignment when it gets to the later assignment of `y`, so this second division would be very fast. However, what if somewhere in the `[...]`, the matrix `m` is modified? Then using the same decomposition would be incorrect - a new decomposition would be needed for the new matrix values.

One solution might be to try to keep track of when a `Matrix` gets changed. We could set an internal flag whenever it is changed to indicate that any existing decomposition is invalid. While not impossible, this type of check is made quite difficult by the way we have allowed different view objects to point to the same data. It would be difficult, and probably very slow, to make sure that any change in one view invalidates the decompositions of all other views to the same data.

Our solution is instead to err on the side of correctness over efficiency and to always recalculate the decomposition by default. Of course, this can be quite inefficient, so we allow the programmer to override this behavior for a specific `Matrix` object with the method:

```
m.saveDiv()
```

After this call, whenever a decomposition is set, it is saved for any future uses. You are telling the program to assume that the values of `m` will not change after that point (technically after the next decomposition is calculated).

If you do modify `m` after a call to `saveDiv()`, you can manually reset the decomposition with

```
m.resetDiv()
```

which deletes any current saved decomposition, and recalculates it. Similarly,

```
m.unsetDiv()
```

will delete any current saved decomposition, but not calculate a new one. This can be used to free up the memory that the decomposition had been using.

Sometimes you may want to set a decomposition before you actually need it. For example, the division may be in a speed critical part of the code, but you have access to the `Matrix` well before then. You can tell the object to calculate the decomposition with

```
m.setDiv()
```

This may also be useful if you just want to perform and access the decomposition separate from any actual division statement (e.g. SVD for principal component analysis). You can also determine whether the decomposition has be set yet with:

```
bool m.divIsSet()
```

Also, if you change what kind of decomposition the `Matrix` should use by calling `divideUsing(...)`, then this will also delete any existing decomposition that might be saved (unless you "change" it to the same thing).

Finally, there is another efficiency issue, which can sometimes be important. The default behavior is to use extra memory for calculating the decomposition, so the original matrix is left unchanged. However, it is often the case that once you have calculated the decomposition, you don't need the original matrix anymore. In that case, it is ok to overwrite the original matrix. For very large matrices, the savings in memory may be significant. (The $O(N^2)$ steps in copying the `Matrix` is generally negligible compared to the $O(N^3)$ steps in performing the decomposition. So memory issues are probably the only reason to do this.)

Therefore, we provide another routine that lets the decomposition be calculated in place, overwriting the original `Matrix`[9]:

```
m.divideInPlace()
```

This method implicitly also calls `m.saveDiv()`, since once the matrix decomposition has been performed, you can't redo it. So most of the time, you will want to save the decomposition after you made it. But if you update the matrix with some new values, and you want to do some division operation with the new values, then you will need to explicitly call either `m.unsetDiv()` or `m.resetDiv()` to redo the decomposition for the new values.

### 5.8.6 Determinants

Aside from very small matrices, the calculation of the determinant typically requires calculations similar to those performed in the above decompositions. Since a determinant only really makes sense for square matrices, one would typically perform an LU decomposition to calculate the determinant. Then the determinant of $A$ is just the determinant of $U$ (possibly times $-1$ depending on the details of $P$), which in turn is simply the product of the values along the diagonal.

[9] Note: for a regular `Matrix`, this is always possible. However, for some of the special matrix varieties, there are decompositions which cannot be done in place. Whenever that is the case, this directive will be ignored.

Therefore, calling `m.det()` involves calculating the LU decomposition, and then finding the determinant of $U$. If you are also performing a division calculation, you should probably use `m.saveDiv()` to avoid calculating the decomposition twice.

If you have set `m` to use some other decomposition using `m.divideUsing(...)`, then the determinant will be determined from that decomposition instead (which is always similarly easy).

For large matrices, the value of the determinant can be extremely large, which can easily lead to overflow problems, even for only moderately large matrices. Therefore, we also provide the method `m.logDet()` which calculates the natural logarithm of the absolute value of the determinant. This method can be given an argument, `sign`, which returns the sign of the determinant. The actual determinant can then be reconstructed from

```
T sign;
T logdet = m.logDet(&sign);
T det = sign * exp(logdet);
```

If `m` is a complex matrix, then the "sign" is really a complex number whose absolute vale is 1. It is defined to be the value of `(det/abs(det))`.

This alternate calculation is especially useful for non-linear maximum likelihood calculations. Since log is a monotonic function, the maximum likelihood is coincident with the maximum of its logarithm. And, since likelihoods in matrix formulation often involve a determinant, the `logDet()` function is exactly what is needed to calculate the log likelihood.

Finally, if `m`'s type is either `int` or `complex<int>` (or some other integer type), then the determinant returned by `m.det()` is calculated a bit differently. In general, division is not accurate for integer types, so we can't use the normal decomposition algorithms for the determinant, since they all use division (and sometimes `sqrt`) as part of the calculation.

But the determinant of an integer matrix is always an integer, so we provide the ability to calculate accurate results in this case using a so-called integer-preserving algorithm by Bareiss (1968). He claims that this algorithm keeps the intermediate values as small as possible to help avoid overflow. In addition to using this algorithm, we also store the intermediate values as `long double`, which has the same number of digits in its mantissa as a 64 bit integer. And if the intermediate values need to exceed this, the values will be rounded somewhat, but in most cases the final answer will still be accurate to the nearest integer, which is what is returned.

Obviously, if your system uses 32 bit integers, then any answer that is not representable in 32 bits will be inaccurate. In this case, the returned answer is either `numeric_limits<int>::max()` (if the correct answer is positive) or `numeric_limits<int>::min()` (if the correct answer is negative). Depending on your application, you may want to include checks for these return values in your code.

### 5.8.7 Accessing the decompositions

Sometimes, you may want to access the components of the decomposition directly, rather than just use them for performing the division or calculating the determinant.

- For the LU decomposition, we have:

  ```
  tmv::ConstLowerTriMatrixView<T> m.lud().getL()
  tmv::ConstUpperTriMatrixView<T> m.lud().getU()
  const Permutation& m.lud().getP()
  bool m.lud().isTrans()
  ```

  `getL()` and `getU()` return $L$ and $U$. `getP()` returns the permutation, $P$. Finally, `isTrans()` returns whether the product $PLU$ is equal to `m` or `m.transpose()`.

  The following should result in a `Matrix m2`, which is numerically very close to the original `Matrix m`:

  ```
  tmv::Matrix<T> m2 = m.lud().getP() * m.lud().getL() * m.lud().getU();
  if (m.lud().isTrans()) m2.transposeSelf();
  ```

- For the QR decomposition, we have:

  ```
  tmv::PackedQ<T> m.qrd().getQ();
  tmv::ConstUpperTriMatrix<T> m.qrd().getR();
  bool m.qrd().isTrans();
  ```

  `getQ()` and `getR()` return $Q$ and $R$. `isTrans()` returns whether the decomposition is equal to `m` or `m.transpose()`.

  Note: the `PackedQ` class is convertible into a regular `Matrix`, but if you are doing arithmetic with it, then these can generally be done without any conversion, so it is more efficient.

  The following should result in a `Matrix m2`, which is numerically very close to the original `Matrix m`:

  ```
  tmv::Matrix<T> m2(m.nrows(),m.ncols());
  tmv::MatrixView<T> m2v =
        m.qrd().isTrans() ? m2.transpose() : m2.view();
  m2v = m.qrd().getQ() * m.qrd().getR();
  ```

- For the QRP decomposition, we have:

  ```
  tmv::PackedQ<T> m.qrpd().getQ()
  tmv::ConstUpperTriMatrixView<T> m.qrpd().getR()
  const Permutation& m.qrpd().getP()
  bool m.qrpd().isTrans()
  ```

  `getQ()` and `getR()` return $Q$ and $R$. `getP()` returns the permutation, $P$. `isTrans()` returns whether the decomposition is equal to `m` or `m.transpose()`.

  The following should result in a `Matrix m2`, which is numerically very close to the original `Matrix m`:

  ```
  tmv::Matrix<T> m2(m.nrows(),m.ncols());
  tmv::MatrixView<T> m2v =
        m.qrpd().isTrans() ? m2.transpose() : m2.view();
  m2v = m.qrpd().getQ() * m.qrpd().getR() * m.qrpd().getP();
  ```

- For the SV decomposition, we have:

  ```
  tmv::ConstMatrixView<T> m.svd().getU()
  tmv::ConstDiagMatrixView<RT> m.svd().getS()
  tmv::ConstMatrixView<T> m.svd().getVt()
  ```

  `getU()`, `getS()`, and `getVt()` return $U$, $S$, and $V^\dagger$. Note that if you would prefer to get the $V$ matrix, you need to write `V = m.svd().getVt().adjoint()`. I find it simpler to deal with the $V^\dagger$ matrix directly, rather than $V$. The adjoint just makes the algorithms more confusing in my opinion, so the TMV algorithms all work in terms of `Vt` $= V^\dagger$. That's why this is the natural matrix to return to the user with a `getVt` method.

  The following should result in a `Matrix m2` that is numerically very close to the original `Matrix m`:

  ```
  tmv::Matrix<T> m2 = m.svd().getU() * m.svd().getS() * m.svd().getVt();
  ```

Each of the above access methods may also be used to perform the decomposition. You do not need to have used the matrix to perform a division before calling `m.lud()`, `m.qrd()`, `m.qrpd()`, or `m.svd()`. Also, calling `m.lud()` implicitly calls `m.divideUsing(tmv::LU)` (likewise for the other decompositions). Thus, if you use one of these decomposition methods, any subsequent division statement will use that decomposition, unless you explicitly call `divideUsing` or call a different decomposition method.

Note that a matrix can only store one decomposition at a time. So if you call `m.lud()` then `m.qrd()`, the LU decomposition is deleted and the QR decomposition is calculated. If you call `m.lud()` again, then it will have to be recalculated.

### 5.8.8 Singular matrices

If a matrix is singular (i.e. its determinant is 0), then LU and QR decompositions will fail when you attempt to divide by the matrix, since the calculation involves division by 0. When this happens, the TMV code throws an exception, `tmv::Singular`. Furthermore, with numerical rounding errors, a matrix that is close to singular may also end up with 0 in a location that gets used for division and thus throw `tmv::Singular`.

Or, more commonly, a singular or nearly singular matrix will just have numerically very small values rather than actual 0's. In this case, there won't be an error, but the results will be numerically very unstable, since the calculation will involve dividing by numbers that are comparable to the machine precision, $\epsilon$.

You can check whether a Matrix is (exactly) singular with the method

```
bool m.isSingular()
```

which basically just returns whether `m.det() == 0` (although it generally doesn't actually have to calculate the determinant to determine whether it is 0). But this will not tell you if a matrix is merely close to singular, so it does not guard against unreliable results.

Singular value decompositions provides a way to deal with singular and nearly singular matrices. There are a number of methods for the object returned by `m.svd()`, which can help diagnose and fix potential problems with a singular matrix.

First, the so-called "singular values", which are the elements of the diagonal $S$ matrix, tell you how close the matrix is to being singular. Specifically, if the ratio of the smallest and the largest singular values, $S(N-1)/S(0)$, is close to the machine precision, $\epsilon$ [10], for the underlying type (`double`, `float`, etc.), then the matrix is singular, or effectively so.

The inverse of this ratio, $S(0)/S(N-1)$, is known as the "condition" of the matrix (specifically the 2-condition, or $\kappa_2$), which can be obtained by:

```
m.svd().condition()
```

The larger the condition, the closer the matrix is to singular, and the less reliable any calculation would be.

So, how does SVD help in this situation? (So far we have diagnosed the possible problem, but not fixed it.) Well, we need to figure out what solution we want from a singular matrix. If the matrix is singular, then there are not necessarily any solutions to $Ax = b$. Furthermore, if we are looking for a least squares solution (rather than an exact solution) then there are an infinite number of choices for $x$ that give the same minimum value of $||Ax - b||_2$.

Another way of looking at is is that there will be particular values of $y$ for which $Ay = 0$. Then given a solution $x$, the vector $x' = x + \alpha y$ for any $\alpha$ will produce the same solution: $Ax' = Ax = b$.

The usual desired solution is the $x$ with minimum 2-norm, $||x||_2$. With SVD, we can get this solution by setting to 0 all of the values in $S^{-1}$ that would otherwise be infinity (or at least large compared to $1/\epsilon$). It is somewhat ironic that the best way to deal with an infinite value is to set it to 0, but that is actually the solution we want.

There are two methods that can be used to control which singular values are set to 0[11]:

```
m.svd().thresh(RT thresh)
m.svd().top(int nsing)
```

`thresh` sets to 0 any singular values with $S(i)/S(0) <$ `thresh`. `top` uses only the largest `nsing` singular values, and sets the rest to 0.

The default behavior is equivalent to:

```
m.svd().thresh(std::numeric_limits<T>::epsilon());
```

[10]Note: the value of $\epsilon$ is accessible with: `std::numeric_limits<T>::epsilon()`.

[11]Technically, the actual values are preserved, but an internal value, `kmax`, keeps track of how many singular values to use.

since at least these values are unreliable. For different applications, you may want to use a larger threshold value.

You can check how many singular values are currently considered non-zero with

```
int m.svd().getKMax()
```

A QRP decomposition can deal with singular matrices similarly, but it doesn't have the flexibility in checking for not-quite-singular but somewhat ill-conditioned matrices like SVD does. QRP will put all of the small elements of R's diagonal in the lower right corner. Then it ignores any that are less than $\epsilon$ when doing the division. For actually singular matrices, this should produce essentially the same result as the SVD solution.

We should also mention again that the 2-norm of a matrix is the largest singular value, which is just $S(0)$ in our decomposition. So this norm requires an SVD calculation, which is relatively expensive compared to the other norms if you have not already calculated the singular value decomposition (and saved it with `m.saveDiv()`). On the other hand, if you have already computed the SVD for division, then `norm2` is trivial and is the fastest norm to compute.

If you just want to calculate the singular values, but don't need to do the actual division, then you don't need to accumulate the $U$ and $V$ matrices. This saves a lot of the calculation time. Or you might want $U$ or $V$, but not both for some purpose. See section §14 for how to do this.

## 5.9 I/O

The simplest output syntax is the usual:

```
os << m;
```

where `os` is any `std::ostream`. The output format is:

```
nrows ncols
( m(0,0)  m(0,1)  m(0,2)  ...  m(0,ncols-1) )
( m(1,0)  m(1,1)  m(1,2)  ...  m(1,ncols-1) )
...
( m(nrows-1,0) ...  ...  m(nrows-1,ncols-1) )
```

The same format can be read back using:

```
tmv::Matrix<T> m;
is >> m;
```

The `Matrix m` is automatically resized to the correct size if necessary based in the size given in the input stream. This is one place where the default constructor (which creates a zero-sized matrix) can be useful.

Often, it is convenient to output only those values that aren't very small. This can be done using

```
os << tmv::ThreshIO(thresh) << m;
```

which writes as 0 any value smaller (in absolute value) than `thresh`. For real `m` it is equivalent to

```
os << tmv::Matrix<T>(m).clip(thresh);
```

but without requiring the temporary `Matrix`. For complex `m` it is slightly different, since it separately tests each component of the complex number. So if `thresh` is `1.e-8`, the complex value `(8,7.5624e-12)` would be writtend `(8,0)`.

There is also a compact I/O format which is mediated by the manipulator `tmv::CompactIO()`, which puts everything on one line and skips the parentheses.

```
os << tmv::CompactIO() << m;
```

would produce the output:

```
M nrows ncols m(0,0) m(0,1) m(0,2) ... m(nrows-1,ncols-1)
```

Note the `M` at the start of the line. Each kind of special matrix will use a different letter (or sometimes 2 letters) to indicate what kind of information follows on the rest of the line, since most special matrices will not need to write out every single value to fully define the matrix. So the initial character (or string) specifies the compact I/O format that follows.

If you want to use a threshold with the compact I/O format, you can do so by writing

```
os << tmv::CompactIO().setThresh(thresh) << m;
```

See §15 for more information about specifying custom I/O styles, including features like using brackets instead of parentheses, putting commas between elements, or specifying an output precision.

## 5.10 Small matrices

The algorithms for regular `Matrix` operations are optimized to be fast for large matrices. Usually, this makes sense, since any code with both large and small matrices will probably have its performance dominated by the speed of the large matrix algorithms.

However, it may be the case that a particular program spends all of its time using $2 \times 2$ or $3 \times 3$ matrices. In this case, many of the features of the TMV code are undesirable. For example, the alias checking in the operation `v2 = m * v1` becomes a significant fraction of the operating time. Even the simple act of performing a function call, rather than doing the calculation inline may be a big performance hit.

So we include the alternate matrix class called `SmallMatrix`, along with the corresponding vector class called `SmallVector` (See §4.10), for which most of the operations are done inline. Furthermore, the sizes are template arguments, rather than normal parameters. This allows the compiler to easily optimize simple calculations that might only be 2 or 3 arithmetic operations, which may significantly speed up your code.

`SmallMatrix` does not inherit from the regular `Matrix` class, because the virtual functions can even be a performance problem for a `SmallMatrix`. However, it does have essentially all the same methods, functions, and arithmetic operators.

### 5.10.1 Constructors

The template arguments `M` and `N` below are both integers and represent the size of the matrix. The template argument `A` can be used to specify the same attributes as for a regular `Matrix`, namely either `tmv::RowMajor` or `tmv::ColMajor` and either `tmv::CStyle` or `tmv::FortranStyle`. These both have the same meanings as they do for a regular `Matrix`. The defaults are `ColMajor` and `CStyle` if not otherwise specified.

- `tmv::SmallMatrix<T,M,N,A> m()`

  Makes an `M` $\times$ `N` `SmallMatrix` with *uninitialized* values.

- `tmv::SmallMatrix<T,M,N,A> m(T x)`

  Makes an `M` $\times$ `N` `SmallMatrix` with all values equal to `x`.

- `tmv::SmallMatrix<T,M,N,A> m1(const Matrix<T>& m2)`

  Makes an `M` $\times$ `N` `SmallMatrix` copy of a regular `Matrix`.

- `tmv::SmallMatrix<T,M,N,A> m1(const SmallMatrix<T2,M,N,A2>& m2)`
  `m1 = m2`

  Copy the `SmallMatrix m2`, which may be of any type `T2` so long as values of type `T2` are convertible into type `T`. The assignment operator has the same flexibility.

-
  ```
  tmv::SmallMatrix<T,M,N,A> m;
  m << m00 , m01 , m02 ...
       m10 , m11 , m12 ...
       ...
  ```

  Initialize the `SmallMatrix m` with a list of values. The elements should be listed in row-major order, but they will be assigned to the correct places in the matrix, even if the matrix does not have `RowMajor` storage.

### 5.10.2 Access

The basic access methods are the same as for a regular `Matrix`. (See 5.3.) However, since the size is known to the compiler, the inline calculation is able to be a lot faster, often reducing to a trivial memory access.

The various view methods, like `row`, `col`, `transpose`, `conjugate`, etc. do not return a `SmallVector` or `SmallMatrix`, so operations with the returned views will not be done inline. However, you can copy the view back to a "`Small`" object, which will be done inline, so that should be fast.

Also, views may be combined with `Small` objects in arithmetic statements. In fact, simple views – those where all the memory elements are contiguous in memory – will usually have the arithmetic calculation done inline. In this case, a good optimizing compiler will probably produce code that eliminates the temporary non-`Small` view. For example:

```
SmallMatrix<T,M,N> m3 = m1.transpose() * m2;
SmallVector<T,N> v2 = m1 * m3.col(2);
```

will both be done inline, even though `m1.transpose()` is not a `SmallMatrix` and `m3.col(2)` is not a `SmallVector`.

### 5.10.3 Functions

`SmallMatrix` has exactly the same function methods as the regular `Matrix`. (See 5.6.) Likewise, the syntax of the arithmetic is identical. The only things that don't take into account the compile-time knowledge of the size are the I/O methods: reading from or writing to a file.

### 5.10.4 Limitations

There are a few limitations on `SmallMatrix` and `SmallVector` objects that we impose in order to give the compiler the ability to optimize them as much as possible.

1. **No alias checking**

   Statements such as

   ```
   v = m * v;
   ```

   will not produce correct code if `v` and `m` are a `SmallVector` and `SmallMatrix`.

   For a regular `Vector` and `Matrix`, the TMV library checks whether any of the objects in an arithmetic statement use the same memory. Then it uses the correct algorithm to deal with it correctly. This alias checking is relatively expensive for small matrices, so we don't do it if `m` and `v` are a `SmallMatrix` and a `SmallVector`. If you need to perform this operation you can write:

   ```
   v = m * tmv::SmallVector<N>(v);
   ```

   to create a temporary copy of the vector on the right hand side.

2. **Views are not “`Small`”**

   All the operations that return some kind of view, either a `VectorView` or a `MatrixView`, are not “`Small`”. As described above, this means that they don’t have all the inlining advantages of `SmallVector` and `SmallMatrix`.

   However, you can copy them back to a `Small` object or combine with `Small` objects in an arithmetic statement, which will be done inline in some cases.

3. **Cannot change division algorithm**

   A `SmallMatrix` does not have the various division control methods like `divideUsing`, `saveDiv`, etc. So a square `SmallMatrix` will always use LU decomposition, and a non-square one will always use QR decomposition. And if you are doing multiple division statements with the same matrix, the library will not save the decomposition between statements.

   There are also some specializations for particular sizes like $2 \times 2$ and $3 \times 3$ matrices.

# 6 Diagonal matrices

The `DiagMatrix` class is our diagonal matrix class. A diagonal matrix is only non-zero along the main diagonal of the matrix.

In addition to the data type template parameter (indicated here by `T` as usual), there is also a second template parameter that specifies attributes of the `DiagMatrix`. The only attributes that are allowed are:

- `CStyle` or `FortranStyle`

which work basically the same way as for a `Vector`. The default value is `CStyle`.

Most functions and methods for `DiagMatrix` work the same as they do for `Matrix`. In these cases, we will just list the functions that are allowed with the effect understood to be the same as for a regular `Matrix`. Of course, there are almost always algorithmic speed-ups, which the code will use to take advantage of the diagonal structure. Whenever there is a difference in how a function works, we will explain the difference.

## 6.1 Constructors

As usual, the optional `A` template argument specifies attributes about the `DiagMatrix`. The default value if not specified is `CStyle`.

- `tmv::DiagMatrix<T,A> d()`

  Makes a `DiagMatrix` with zero size. You would normally use the `resize` function later to change the size to some useful value.

- `tmv::DiagMatrix<T,A> d(int n)`

  Makes an `n` $\times$ `n` `DiagMatrix` with *uninitialized* values. If extra debugging is turned on (with the compiler flag `-DTMV_EXTRA_DEBUG`), then the values along the diagonal are in fact initialized to 888.

- `tmv::DiagMatrix<T,A> d(int n, T x)`

  Makes an `n` $\times$ `n` `DiagMatrix` with all values along the diagonal equal to `x`.

- `tmv::DiagMatrix<T,A> d(const Vector<T,A2>& v)`

  Makes a `DiagMatrix` with `v` as the diagonal.

- `tmv::DiagMatrix<T,A> d(const Matrix<T,A2>& m)`

  Makes a `DiagMatrix` with the diagonal of `m` as the diagonal.

-
  ```
  tmv::DiagMatrix<T,A> d1(const DiagMatrix<T2,A2>& d2)
  d1 = d2
  ```

  Copy the `DiagMatrix d2`, which may be of any type `T2` so long as values of type `T2` are convertible into type `T`. The assignment operator has the same flexibility.

-
  ```
  tmv::DiagMatrix<T,A> d;
  d << d00,
           d11,
               d22,
                   d33,
                       ...
  ```

Initialize the `DiagMatrix d`, with a list of values. Note that only the values on the diagonal should be given. (They do not need to be on different lines as shown – that is merely to suggest the structure of the values in the matrix.)

-
  ```
  tmv::DiagMatrixView<T,A> d = DiagMatrixViewOf(Vector<T>& v)
  tmv::ConstDiagMatrixView<T,A> d = DiagMatrixViewOf(const Vector<T>& v)
  ```

  Makes a `DiagMatrixView` whose diagonal is `v`. Note: `v` may also be a view, rather than an actual `Vector`.

-
  ```
  tmv::DiagMatrixView<T,A> d = tmv::DiagMatrixViewOf(T* vv, int n)
  tmv::ConstDiagMatrixView<T,A> d =
        tmv::DiagMatrixViewOf(const T* vv, int n)
  tmv::DiagMatrixView<T,A> d =
        tmv::DiagMatrixViewOf(T* vv, int n, int step)
  tmv::ConstDiagMatrixView<T,A> d =
        tmv::DiagMatrixViewOf(const T* vv, int n, int step)
  ```

  Make a `DiagMatrixView` whose diagonal consists of the actual memory elements `vv`. The last two allow for a non-unit step between the elements.

## 6.2 Access

-
  ```
  d.nrows() = d.ncols() = d.colsize() = d.rowsize() = d.size()
  d.resize(int newsize)
  d(i,j)
  d(i) = d(i,i)
  d.cref(i)
  d.ref(i)
  ```

  For the mutable `d(i,j)` version, `i` must equal `j`. If `d` is not mutable, then `d(i,j)` with `i`≠`j` returns the value 0.

-
  ```
  d.diag()
  ```

  This returns the main diagonal as usual.

-
  ```
  DiagMatrix<T>::iterator d.begin()
  DiagMatrix<T>::iterator d.end()
  DiagMatrix<T>::const_iterator d.begin() const
  DiagMatrix<T>::const_iterator d.end() const
  ```

  These iterate along the diagonal of the `DiagMatrix`.

-
  ```
  T* d.ptr()
  const T* d.cptr() const
  int d.step() const
  bool d.isconj() const
  ```

  These methods allow for direct memory access of a `DiagMatrix`.

-
  ```
  d.subDiagMatrix(int i1, int i2, int istep = 1)
  ```

  This is equivalent to `DiagMatrixViewOf(d.diag().subVector(i1,i2,istep))`.

-
  ```
  d.transpose() = d.view()
  d.conjugate() = d.adjoint()
  d.cview()
  d.fview()
  d.realPart()
  d.imagPart()
  ```

  These return `DiagMatrixViews`.

## 6.3 Functions

```
RT d.norm1() = Norm1(d)
RT d.norm2() = Norm2(d)
RT d.normInf() = NormInf(d)
RT d.maxAbsElement() = MaxAbsElement(d)
```

(Actually for a diagonal matrix, all of the above norms are equal.)

```
RT d.maxAbs2Element() = MaxAbs2Element(d)
RT d.normF() = NormF(d) = d.norm() = Norm(d)
RT d.normSq() = NormSq(d)
RT d.normSq(RT scale)
T d.trace() = Trace(d)
T d.sumElements() = SumElements(d)
RT d.sumAbsElements() = SumAbsElements(d)
RT d.sumAbs2Elements() = SumAbs2Elements(d)
T d.det() = Det(d)
RT d.logDet(T* sign=0) = LogDet(d)
bool d.isSingular()
RT d.condition()
RT d.doCondition()
dinv = d.inverse() = Inverse(d)
d.makeInverse(Matrix<T>& minv)
d.makeInverse(DiagMatrix<T>& dinv)
d.makeInverseATA(Matrix<T>& cov)
d.makeInverseATA(DiagMatrix<T>& cov)
```

Since the inverse of a `DiagMatrix` is a `DiagMatrix`, we also provide a version of the `makeInverse` syntax, which allows dinv to be a `DiagMatrix`. (Likewise for `makeInverseATA`.) The same option is available with the operator version: `dinv = d.inverse()`.

```
d.setZero()
d.setAllTo(T x)
d.addToAll(T x)
d.clip(RT thresh)
d.setToIdentity(T x = 1)
d.conjugateSelf()
d.transposeSelf() // null operation
d.invertSelf()
Swap(d1,d2)
```

There is one new method here for `DiagMatrix`: The method `invertSelf` calculates $d^{-1}$ in place. It is equivalent to `d = d.inverse()` and, like the other division operations, is invalid for `T = int` or `complex<int>`.

## 6.4 Arithmetic

In addition to `x`, `v`, and `m` from before, we now add `d` for a `DiagMatrix`.

```
d2 = -d1
d2 = x * d1
d2 = d1 [*/] x
d3 = d1 [+-] d2
m2 = m1 [+-] d
m2 = d [+-] m1
d [*/]= x
d2 [+-]= d1
m [+-]= d
v2 = d * v1
v2 = v1 * d
v *= d
d3 = d1 * d2
d3 = ElemProd(d1,d2)
m2 = d * m1
m2 = m1 * d
d2 *= d1
m *= d
d2 = d1 [+-] x
d2 = x [+-] d1
d [+-]= x
d = x
```

## 6.5 Division

The division operations are:

```
v2 = v1 [/%] d
m2 = m1 [/%] d
m2 = d [/%] m1
d3 = d1 [/%] d2
d2 = x [/%] d1
v [/%]= d
d2 [/%]= d1
m [/%]= d
```

Division by a diagonal matrix does not require any decomposition, so there are none of the usual helper functions for division that a regular `Matrix` has such as `divideUsing`, `saveDiv`, etc. If a `DiagMatrix` is singular, you can find out with `d.isSingular()`, but there is no direct way to use SVD for the division and avoid any divisions by 0. If you want to do this, you should use `BandMatrixViewOf(d)` to treat `d` as a `BandMatrix`, which can use SVD.

## 6.6 I/O

The simplest I/O syntax is the usual:

```
os << d;
is >> d;
```

The output format is the same as for a `Matrix`, including all the 0's. (See §5.9.) On input, if any off diagonal elements are not 0, a `tmv::ReadError` is thrown.

There is also a compact I/O style:

```
os << tmv::CompactIO() << d;
is >> tmv::CompactIO() >> d;
```

which uses the format:

```
D n d(0,0) d(1,1) d(2,2) ... d(n-1,n-1)
```

One can also write small values as 0 with

```
os << tmv::ThreshIO(thresh) << d;
os << tmv::CompactIO().setThresh(thresh) << d;
```

See §15 for more information about specifying custom I/O styles, including features like using brackets instead of parentheses, or putting commas between elements, or specifying an output precision.

# 7 Upper/lower triangle matrices

The `UpperTriMatrix` class is our upper triangle matrix class, which is non-zero only on the main diagonal and above. `LowerTriMatrix` is our class for lower triangle matrices, which are non-zero only on the main diagonal and below.

In addition to the data type template parameter (indicated here by `T` as usual), there is also a second template parameter that specifies attributes of the `UpperTriMatrix` or `LowerTriMatrix`. The attributes that are allowed are:

- `CStyle` or `FortranStyle`
- `ColMajor` or `RowMajor`
- `NonUnitDiag` or `UnitDiag`

The first two options work the same as for a regular `Matrix`. The final option refers to whether the elements on the diagonal should be taken to be all 1's (`UnitDiag`) rather than what is actually stored in memory (`NonUnitDiag`). The default attributes are `CStyle`, `ColMajor` and `NonUnitDiag`.

The storage of both an `UpperTriMatrix` and a `LowerTriMatrix` takes $N \times N$ elements of memory, even though approximately half of them are never used. Someday, I'll write the packed storage versions, which will allow for more efficient storage of the matrices.

Most functions and methods for `UpperTriMatrix` and `LowerTriMatrix` work the same as they do for `Matrix`. In these cases, we will just list the functions that are allowed with the effect understood to be the same as for a regular `Matrix`. Of course, there are almost always algorithmic speed-ups, which the code will use to take advantage of the triangle structure. Whenever there is a difference in how a function works, we will explain the difference. Also, all of the routines are analogous for `UpperTriMatrix` and `LowerTriMatrix`, so we only list each routine once (the `UpperTriMatrix` version for definiteness).

## 7.1 Constructors

As usual, the optional `A` template argument specifies attributes about the `UpperTriMatrix`. The default attributes if not specified are `CStyle|ColMajor|NonUnitDiag`. The constructors for `LowerTriMatrix` are completely analogous and are omitted for brevity.

- `tmv::UpperTriMatrix<T,A> U()`

  Makes an `UpperTriMatrix` with zero size. You would normally use the `resize` function later to change the size to some useful value.

- `tmv::UpperTriMatrix<T,A> U(int n)`

  Makes an `n` $\times$ `n` `UpperTriMatrix` with *uninitialized* values. If extra debugging is turned on (with the compiler flag `-DTMV_EXTRA_DEBUG`), then the values are in fact initialized to 888.

- `tmv::UpperTriMatrix<T,A> U(int n, T x)`

  Makes an `n` $\times$ `n` `UpperTriMatrix` with all values equal to `x`.

- `tmv::UpperTriMatrix<T,A> U(const Matrix<T,A2>& m)`
  `tmv::UpperTriMatrix<T,A> U(const UperTriMatrix<T,A2>& U2)`

  Make an `UpperTriMatrix` which copies the corresponding values of `m` or `U2`. Note that the attributes are allowed to be different. Specifically, `U2` can be `NonUnitDiag` and `U` can be `UnitDiag`, in which case, it will only copy the off-diagonal values, and take the diagonal values to be all 1's. Going the other way from `UnitDiag` to `NonUnitDiag` will put actual 1's into memory along the diagonal.

-
  ```
  tmv::UpperTriMatrix<T,A> U1(const UpperTriMatrix<T2,A2>& U2)
  U1 = U2
  ```

  Copy the `UpperTriMatrix U2`, which may be of any type `T2` so long as values of type `T2` are convertible into type `T`. The assignment operator has the same flexibility.

-
  ```
  tmv::UpperTriMatrix<T,A> U;
  U << U00, U01, U02, ...
            U11, U12, ...
                 U22, ...
                      ...
  tmv::LowerTriMatrix<T,A> L;
  L << L00,
       L10, L11,
       L20, L21, L22,
       ...
  ```

  Initialize the `UpperTriMatrix U` and `LowerTriMatrix L` with lists of values. As usual, the list elements should be listed in RowMajor order, and only the values that are in the upper or lower triangle should be given. If `U` or `L` is `UnitDiag`, then the diagonal elements should be listed as 1's.

-
  ```
  tmv::UpperTriMatrixView<T,A> U =
        tmv::UpperTriMatrixViewOf(T* vv, int n, StorageType stor,
        DiagType dt=NonUnitDiag)
  tmv::ConstUpperTriMatrixView<T,A> U =
        tmv::UpperTriMatrixViewOf(const T* vv, int n, StorageType stor,
        DiagType dt=NonUnitDiag)
  tmv::UpperTriMatrixView<T,A> U =
        tmv::UnitUpperTriMatrixViewOf(T* vv, int n, StorageType stor)
  tmv::UpperTriMatrixView<T,A> U =
        tmv::UnitUpperTriMatrixViewOf(T* vv, int n, StorageType stor)
  ```

  Make a `UpperTriMatrixView` of the actual memory elements, `vv`. One wrinkle here is that if `dt` is `UnitDiag`, then `vv` is still the location of the upper left corner, even though that value is never used (since the value is just taken to be 1). Also, `vv` must be of length $\texttt{n} \times \texttt{n}$, so all of the lower triangle elements must be in memory, even though they are never used.

  The last two functions are currently equivalent to the first two with `dt=UnitDiag`. However, in future versions of TMV this will allow for the `dt` value to be known at compile time rather than run time which could improve efficiency in some cases.

  Also, `A` here should only be `CStyle` or `FortranStyle`. For now, this is the only attribute we allow for views (of any matrix variety). Later versions will allow views to keep track of more of their attributes at compile time, but for now things like whether it is `UnitDiag` or `NonUnitDiag` are runtime variables.

-
  ```
  tmv::UpperTriMatrixView<T,A> U =
        tmv::UpperTriMatrixViewOf(T* vv, int n, int stepi, int stepj,
        DiagType dt=NonUnitDiag)
  tmv::ConstUpperTriMatrixView<T,A> U =
        tmv::UpperTriMatrixViewOf(const T* vv, int n, int stepi,
        int stepj, DiagType dt=NonUnitDiag)
  tmv::UpperTriMatrixView<T,A> U =
        tmv::UnitUpperTriMatrixViewOf(T* vv, int n, int stepi, int stepj)
  ```

```
tmv::ConstUpperTriMatrixView<T,A> U =
      tmv::UnitUpperTriMatrixViewOf(const T* vv, int n, int stepi,
      int stepj)
```

Make a `UpperTriMatrixView` of the actual memory elements, `vv`. These versions allow you to provide an arbitrary step through the data in the $i$ and $j$ directions.

## 7.2 Access

-
  ```
  U.nrows() = U.ncols() = U.colsize() = U.rowsize() = U.size()
  U.resize(int newsize)
  U(i,j)
  U.cref(i,j)
  U.ref(i,j)
  ```

  For the mutable `U(i,j)` version, the position must fall within the upper triangle. If `U` is `const`, then `U(i,j)` returns 0 for positions in the lower triangle. `U(i,i)` does work if `U` is `UnitDiag`, but assigning anything other than `1` is an error.

-
  ```
  U.row(int i, int j1, int j2)
  U.col(int j, int i1, int i2)
  U.diag()
  U.diag(int i)
  U.diag(int i, int k1, int k2)
  ```

  Note that the versions of `row` and `col` with only one argument are missing, since the full row or column isn't accessible as a `VectorView`. You must specify a valid range within the row or column that you want, given the upper triangle shape of `U`. Likewise for the `LowerTriMatrix` versions of these. If `U` is `UnitDiag`, then the range may not include the diagonal element. Similarly, `U.diag()` is valid only if `U` is `NonUnitDiag`.

-
  ```
  UpperTriMatrix<T>::rowmajor_iterator U.rowmajor_begin()
  UpperTriMatrix<T>::rowmajor_iterator U.rowmajor_end()
  UpperTriMatrix<T>::const_rowmajor_iterator U.rowmajor_begin() const
  UpperTriMatrix<T>::const_rowmajor_iterator U.rowmajor_end() const
  UpperTriMatrix<T>::colmajor_iterator U.colmajor_begin()
  UpperTriMatrix<T>::colmajor_iterator U.colmajor_end()
  UpperTriMatrix<T>::const_colmajor_iterator U.colmajor_begin() const
  UpperTriMatrix<T>::const_colmajor_iterator U.colmajor_end() const
  ```

  These iterators only traverse the elements that are actually stored in memory. So for an `UpperTriMatrix`, the iteration skips all the elements the the strict lower triangle.

-
  ```
  T* U.ptr()
  const T* U.cptr() const
  int U.stepi() const
  int U.stepj() const
  bool U.isconj() const
  bool U.isrm() const
  bool U.iscm() const
  bool U.isunit() const
  ```

These methods allow for direct memory access of an `UpperTriMatrix` or `LowerTriMatrix` or views thereof. The last item is new and returns whether `U` is `UnitDiag` or not.

- `U.subVector(int i, int j, int istep, int jstep, int size)`
  `U.subMatrix(int i1, int i2, int j1, int j2)`
  `U.subMatrix(int i1, int i2, int j1, int j2, int istep, int jstep)`

  This works the same as for `Matrix` (See 5.3), except that all of the elements in the subvector or submatrix must be completely within the upper or lower triangle, as appropriate. If `U` is `UnitDiag`, then no elements may be on the main diagonal.

- `U.subTriMatrix(int i1, int i2, int istep = 1)`

  This returns the upper or lower triangle matrix whose upper-left corner is `U(i1,i1)`, and whose lower-right corner is `U(i2-istep,i2-istep)` for C-style indexing or `U(i2,i2)` for Fortran-style indexing. If `istep` $\neq 1$, then it is the step in both the `i` and `j` directions.

- `U.offDiag()`

  This returns a view to the portion of the triangle matrix that does not include the diagonal elements. It will always be `NonUnitDiag`. Internally, it provides an easy way to deal with the `UnitDiag` triangle matrices for many routines. But it may be useful for some users as well.

- `U.viewAsUnitDiag()`

  This returns a view to a `NonUnitDiag` triangle matrix that treats it instead as a `UnitDiag` triangle matrix.

- `U.transpose()`
  `U.conjugate()`
  `U.adjoint()`
  `U.view()`
  `U.cView()`
  `U.fView()`
  `U.realPart()`
  `U.imagPart()`

  Note that the transpose and adjoint of an `UpperTriMatrix` is an `LowerTriMatrixView` and vice versa. Otherwise, these all return `UpperTriMatrixViews`.

## 7.3 Functions

```
RT U.norm1() = Norm1(U)
RT U.norm2() = Norm2(U)
RT U.normInf() = NormInf(U)
RT U.normF() = NormF(U) = U.norm() = Norm(U)
RT U.normSq() = NormSq(U)
RT U.normSq(RT scale)
RT U.maxAbsElement() = MaxAbsElement(U)
RT U.maxAbs2Element() = MaxAbs2Element(U)
T U.trace() = Trace(U)
T U.sumElements() = SumElements(U)
RT U.sumAbsElements() = SumAbsElements(U)
RT U.sumAbs2Elements() = SumAbs2Elements(U)
T U.det() = Det(U)
RT U.logDet(T* sign=0) = LogDet(U)
bool U.isSingular()
RT U.condition()
RT U.doCondition()
Uinv = U.inverse() = Inverse(U)
U.makeInverse(Matrix<T>& minv)
U.makeInverse(UpperTriMatrix<T>& Uinv)
U.makeInverseATA(Matrix<T>& cov)
```

Since the inverse of an `UpperTriMatrix` is also upper triangular, the object returned by `U.inverse()` is assignable to an `UpperTriMatrix`. Of course you can also assign it to a regular `Matrix` if you prefer. Similarly, there are versions of `U.makeInverse(minv)` for both argument types.

```
U.setZero()
U.setAllTo(T x)
U.addToAll(T x)
U.clip(RT thresh)
U.setToIdentity(T x = 1)
U.conjugateSelf()
U.invertSelf()
Swap(U1,U2)
```

Like for `DiagMatrix`, `invertSelf` calculates $U^{-1}$ in place. It is equivalent to `U = U.inverse()` and, like the other division operations, is invalid for `T = int` or `complex<int>`.

## 7.4 Arithmetic

In addition to `x`, `v`, and `m` from before, we now add `U` and `L` for a `UpperTriMatrix` and `LowerTriMatrix` respectively. Where the syntax is identical for the two cases, only the `U` form is listed.

```
U2 = -U1
U2 = x * U1
U2 = U1 [*/] x
U3 = U1 [+-] U2
m2 = m1 [+-] U
m2 = U [+-] m1
m = L [+-] U
m = U [+-] L
U [*/]= x
U2 [+-]= U1
m [+-]= U
v2 = U * v1
v2 = v1 * U
v *= U
U3 = U1 * U2
U3 = ElemProd(U1,U2)
m2 = U * m1
m2 = m1 * U
m = U * L
m = L * U
U2 *= U1
```

```
m *= U
U2 = U1 [+-] x
U2 = x [+-] U1
U [+-]= x
```

## 7.5 Division

The division operations are: (again omitting the L forms when redundant)

```
v2 = v1 [/%] U
m2 = m1 [/%] U
m2 = U [/%] m1
U3 = U1 [/%] U2
U2 = x [/%] U1
m = U [/%] L
m = L [/%] U
v [/%]= U
U2 [/%]= U1
m [/%]= U
```

Division by a triangular matrix does not require any decomposition, so there are none of the usual helper functions for division that a regular `Matrix` has such as `divideUsing`, `saveDiv`, etc. If an `UpperTriMatrix` or `LowerTriMatrix` is singular, you can find out with `m.isSingular()`, but there is no direct way to use SVD for the division and avoid any divisions by 0. If you want to do this, you should use `BandMatrixViewOf(U)` to treat `U` as a `BandMatrix`, which can use SVD.

## 7.6 I/O

The simplest I/O syntax is the usual:

```
os << U << L;
is >> U >> L;
```

The output format is the same as for a `Matrix`, including all the 0's. (See 5.9.) On input, if any of the elements in the wrong triangle are not 0, a `tmv::ReadError` is thrown. Likewise if the input matrix is `UnitDiag`, but a diagonal element read in is not equal to 1.

There is also a compact I/O style that puts all the elements that aren't trivially 0 all on a single line and skips the parentheses.

```
os << tmv::CompactIO() << U;
os << tmv::CompactIO() << L;
is >> tmv::CompactIO() >> U;
is >> tmv::CompactIO() >> L;
```

One can also write small values as 0 with

```
os << tmv::ThreshIO(thresh) << U;
os << tmv::ThreshIO(thresh) << L;
os << tmv::CompactIO().setThresh(thresh) << U;
os << tmv::CompactIO().setThresh(thresh) << L;
```

See §15 for more information about specifying custom I/O styles, including features like using brackets instead of parentheses, or putting commas between elements, or specifying an output precision.

## 8 Band-diagonal matrices

The BandMatrix class is our band-diagonal matrix, which is only non-zero on the main diagonal and a few sub- and super-diagonals. While band-diagonal matrices are usually square, we allow for non-square banded matrices as well. You may even have rows or columns that are completely outside of the band structure, and hence are all 0. For example a $10 \times 5$ band matrix with 2 sub-diagonals is valid even though the bottom 3 rows are all 0.

Throughout, we use nlo to refer to the number of sub-diagonals (below the main diagonal) stored in the BandMatrix, and nhi to refer to the number of super-diagonals (above the main diagonal).

All the BandMatrix routines are included by:

```
#include "TMV_Band.h"
```

In addition to the data type template parameter (indicated here by T as usual), there is also a second template parameter that specifies attributes of the BandMatrix. The attributes that are allowed are:

- CStyle or FortranStyle
- ColMajor, RowMajor, or DiagMajor

For this class, we have the additional storage possibility: DiagMajor, which has unit step along the diagonals. It stores the values in memory starting with the lowest sub-diagonal and progressing up to the highest super-diagonal. The default attributes are CStyle and ColMajor.

For each type of storage, we require that the step size in each direction be uniform within a given row, column or diagonal. This means that we require a few extra elements of memory that are not actually used. To demonstrate the different storage orders and why extra memory is required, here are three $6 \times 6$ band-diagonal matrices, each with $nlo = 2$ and $nhi = 3$ in each of the different storage types. The number in each place indicates the offset in memory from the top left element.

$$\text{ColMajor:} \quad \begin{pmatrix} 0 & 5 & 10 & 15 & & \\ 1 & 6 & 11 & 16 & 21 & \\ 2 & 7 & 12 & 17 & 22 & 27 \\ & 8 & 13 & 18 & 23 & 28 \\ & & 14 & 19 & 24 & 29 \\ & & & 20 & 25 & 30 \end{pmatrix}$$

$$\text{RowMajor:} \quad \begin{pmatrix} 0 & 1 & 2 & 3 & & \\ 5 & 6 & 7 & 8 & 9 & \\ 10 & 11 & 12 & 13 & 14 & 15 \\ & 16 & 17 & 18 & 19 & 20 \\ & & 22 & 23 & 24 & 25 \\ & & & 28 & 29 & 30 \end{pmatrix}$$

$$\text{DiagMajor:} \quad \begin{pmatrix} 0 & 6 & 12 & 18 & & \\ -5 & 1 & 7 & 13 & 19 & \\ -10 & -4 & 2 & 8 & 14 & 20 \\ & -9 & -3 & 3 & 9 & 15 \\ & & -8 & -2 & 4 & 10 \\ & & & -7 & -1 & 5 \end{pmatrix}$$

First, notice that all three storage methods require 4 extra locations in memory, which do not hold any actual matrix data. (They require a total of 31 memory addresses for only 27 that are used.) This is because we want to have the same step size between consecutive row elements for every row. Likewise for the columns (which in turn implies that it is also true for the diagonals).

For $N \times N$ square matrices, the total memory needed is $(N - 1) * (nlo + nhi + 1) + 1$, which wastes $(nlo - 1) * nlo/2 + (nhi - 1) * nhi/2$ locations. For non-square matrices, the formula is more complicated, and changes slightly between the three storages. If you want to know the memory used by a `BandMatrix`, we provide the routine:

```
int BandStorageLength(StorageType stor, int nrows, int ncols, int nlo,
      int nhi)
```

For square matrices, all three methods always need the same amount of memory (and for non-square, they aren't very different), so the decision about which method to use should generally be based on performance considerations rather than memory usage. The speed of the various matrix operations are different for the different storage types. If the matrix calculation speed is important, it may be worth trying all three to see which is fastest for the operations you are using.

Second, notice that the `DiagMajor` storage doesn't start with the upper left element as usual. Rather, it starts at the start of the lowest sub-diagonal. This mostly just impacts the `BandMatrixViewOf` function. If the storage is given as `DiagMajor`, then the start of the array needs to be at the start of the lowest sub-diagonal.

Most functions and methods for `BandMatrix` work the same as they do for `Matrix`. In these cases, we will just list the functions that are allowed with the effect understood to be the same as for a regular `Matrix`. Of course, there are almost always algorithmic speed-ups, which the code will use to take advantage of the banded structure. Whenever there is a difference in how a function works, we will explain the difference.

## 8.1 Constructors

As usual, the optional `A` template argument specifies attributes about the `DiagMatrix`. The default attributes if not specified are `CStyle|ColMajor`.

- `tmv::BandMatrix<T,A> b()`

  Makes a `BandMatrix` with zero size. You would normally use the `resize` function later to change the size to some useful value.

- `tmv::BandMatrix<T,A> b(int nrows, int ncols, int nlo, int nhi)`

  Makes a `BandMatrix` with `nrows` rows, `ncols` columns, `nlo` sub-diagonals, and `nhi` super-diagonals with *uninitialized* values. If extra debugging is turned on (with the compiler flag `-DTMV_EXTRA_DEBUG`), then the values are initialized to 888.

- `tmv::BandMatrix<T,A> b(int nrows, int ncols, int nlo, int nhi, T x)`

  Makes a `BandMatrix` with all values equal to `x`.

-
  ```
  tmv::BandMatrix<T,A> b(const Matrix<T,A2>& m, int nlo, int nhi)
  tmv::BandMatrix<T,A> b(const BandMatrix<T,A2>& m, int nlo, int nhi)
  tmv::BandMatrix<T,A> b(const UpperTriMatrix<T,A2>& m, int nhi)
  tmv::BandMatrix<T,A> b(const LowerTriMatrix<T,A2>& m, int nlo)
  ```

  Make a `BandMatrix` the same size as `m`, which copies the values of `m` that are within the band defined by `nlo` and `nhi` For the second one, `nlo` and `nhi` must not be larger than those for `m`. For the last two, `nlo` and `nhi` (respectively) are taken to be 0.

-
  ```
  tmv::BandMatrix<T,tmv::DiagMajor> m = UpperBiDiagMatrix(
        const Vector<T>& v1, const Vector<T>& v2)
  tmv::BandMatrix<T,tmv::DiagMajor> m = LowerBiDiagMatrix(
        const Vector<T>& v1, const Vector<T>& v2)
  tmv::BandMatrix<T,tmv::DiagMajor> m = TriDiagMatrix(
        const Vector<T>& v1,  const Vector<T>& v2, const Vector<T>& v3)
  ```

Shorthand to create bi- or tri-diagonal `BandMatrix`es if you already have the `Vectors`. The `Vectors` are in order from bottom to top in each case.

-
  ```
  tmv::BandMatrix<T,A> b1(const BandMatrix<T2,A2>& b2)
  b1 = b2
  ```

  Copy the `BandMatrix b2`, which may be of any type `T2` so long as values of type `T2` are convertible into type `T`. The assignment operator has the same flexibility.

-
  ```
  tmv::BandMatrixView<T,A> b =
        BandMatrixViewOf(MatrixView<T> m, int nlo, int nhi)
  tmv::BandMatrixView<T,A> b =
        BandMatrixViewOf(BandMatrixView<T> m, int nlo, int nhi)
  tmv::BandMatrixView<T,A> b =
        BandMatrixViewOf(DiagMatrixView<T> m)
  tmv::BandMatrixView<T,A> b =
        BandMatrixViewOf(UpperTriMatrixView<T> m)
  tmv::BandMatrixView<T,A> b =
        BandMatrixViewOf(UpperTriMatrixView<T> m, int nhi)
  tmv::BandMatrixView<T,A> b =
        BandMatrixViewOf(LowerTriMatrixView<T> m)
  tmv::BandMatrixView<T,A> b =
        BandMatrixViewOf(LowerTriMatrixView<T> m, int nlo)
  ```

  Make an `BandMatrixView` of the corresponding portion of `m`. There are also `ConstBandMatrixView` versions of all of these.

-
  ```
  tmv::BandMatrixView<T,A> b(MatrixView<T> m, int nlo, int nhi)
  tmv::BandMatrixView<T,A> b(BandMatrixView<T> m, int nlo, int nhi)
  ```

  For square matrices `m`, these (and the corresponding `ConstBandMatrixView` versions) work the same as the above `BandMatrixViewOf` commands. However, this version preserves the values of `nrows` and `ncols` from `m` even if some of the rows or columns do not include any of the new band. This is only important if `m` is not square.

  For example, if `m` is $10 \times 8$, then

  ```
  tmv::BandMatrixView<T> b1(m,0,2);
  ```

  will create a $10 \times 8$ `BandMatrixView` of `m`'s diagonal plus two super-diagonals, but

  ```
  tmv::BandMatrixView<T> b2 = BandMatrixViewOf(m,0,2);
  ```

  will instead create an $8 \times 8$ `BandMatrixView` of the same portion of `m`.

  Note that the same difference holds for the `BandMatrix` constructor:

  ```
  tmv::BandMatrix<T> b1(m,0,2);
  ```

  will create a $10 \times 8$ `BandMatrix`, but

  ```
  tmv::BandMatrix<T> b2 = BandMatrixViewOf(m,0,2);
  ```

  will create an $8 \times 8$ `BandMatrix`.

-
  ```
  tmv::BandMatrixView<T> b =
        tmv::BandMatrixViewOf(T* vv, int ncols, int nrows,
            int nlo, int nhi, StorageType stor)
  tmv::ConstBandMatrixView<T> b =
        tmv::BandMatrixViewOf(const T* vv, int ncols, int nrows,
            int nlo, int nhi, StorageType stor)
  ```

  Make a `BandMatrixView` of the actual memory elements, `vv`. The length of the data array should be `BandStorageLength(stor,ncols,nrows,nlo,nhi)`. One important point: The `DiagMajor` storage order starts at the beginning of the lowest sub-diagonal, so `vv` is the memory address of `b(nlo,0)`, not `b(0,0)`.

-
  ```
  tmv::BandMatrixView<T> b =
        tmv::BandMatrixViewOf(T* vv, int ncols, int nrows,
            int nlo, int nhi, int stepi, int stepj)
  tmv::ConstBandMatrixView<T> b =
        tmv::BandMatrixViewOf(const T* vv, int ncols, int nrows,
            int nlo, int nhi, int stepi, int stepj)
  ```

  Make a `BandMatrixView` of the actual memory elements, `vv`. These allow for arbitrary steps through the data. Also, the value `*vv` is always `b(0,0)` even if the elements are laid out in `DiagMajor` order in memory.

## 8.2 Access

-
  ```
  b.nrows() = b.colsize()
  b.ncols() = b.rowsize()
  b.resize(int new_nrows, int new_ncols, int new_nlo, int new_nhi)
  b(i,j)
  b.cref(i,j)
  b.ref(i,j)
  ```

  For the mutable `b(i,j)` version, the position must fall within the band. If `b` is not mutable, then `b(i,j)` return 0 for positions outside of the band.

-
  ```
  b.row(int i, int j1, int j2)
  b.col(int j, int i1, int i2)
  b.diag()
  b.diag(int i)
  b.diag(int i, int k1, int k2)
  ```

  The versions of `row` and `col` with only one argument are missing, since the full row or column isn't accessible as a `VectorView`. You must specify a valid range within the row or column that you want, given the banded storage of `b`.

-
  ```
  ConstVectorView<T> b.constLinearView() const
  VectorView<T> b.linearView()
  bool b.canLinearize()
  ```

  These are similar to the regular `Matrix` version in that they return a view to the elements of a `BandMatrix` as a single vector. However, in this case, there are a few elements in memory that are not necessarily defined, since they lie outside of the actual band structure, so some care should be used depending on the application of the returned vector views. They are always allowed for an actual `BandMatrix`. For a `BandMatrixView` (or `ConstBandMatrixView`), they are only allowed if all of the elements in the view are in one contiguous block of memory. The helper function `b.canLinearize()` returns whether or not the first two methods will work.

-
  ```
  BandMatrix<T>::rowmajor_iterator b.rowmajor_begin()
  BandMatrix<T>::rowmajor_iterator b.rowmajor_end()
  BandMatrix<T>::const_rowmajor_iterator b.rowmajor_begin() const
  BandMatrix<T>::const_rowmajor_iterator b.rowmajor_end() const
  BandMatrix<T>::colmajor_iterator b.colmajor_begin()
  BandMatrix<T>::colmajor_iterator b.colmajor_end()
  BandMatrix<T>::const_colmajor_iterator b.colmajor_begin() const
  BandMatrix<T>::const_colmajor_iterator b.colmajor_end() const
  BandMatrix<T>::diagmajor_iterator b.diagmajor_begin()
  BandMatrix<T>::diagmajor_iterator b.diagmajor_end()
  BandMatrix<T>::const_diagmajor_iterator b.diagmajor_begin() const
  BandMatrix<T>::const_diagmajor_iterator b.diagmajor_end() const
  ```

  Here we have added a new set of iterators that traverse the matrix in diagonal-major order. Unlike the others, they do not start with $m(0,0)$. They instead start at the beginning of the lowest subdiagonal in the band and proceed up to the highest super-diagonal. Note: If you want to iterate along the diagonals starting with the highest super-diagonal and proceeding down, you can use `b.transpose().diagmajor_begin()`.

-
  ```
  T* b.ptr()
  const T* b.cptr() const
  int b.stepi() const
  int b.stepj() const
  int b.diagstep() const
  bool b.isconj() const
  bool b.isrm() const
  bool b.iscm() const
  bool b.isdm() const
  ```

  These methods allow for direct memory access of a `BandMatrix`. There are two new methods here: `m.diagstep() = m.stepi()+m.stepj()` returns the step size along the diagonal directions, and `isdm()` is another convenience function that returns whether `b` is `DiagMajor`.

-
  ```
  b.subVector(int i, int j, int istep, int jstep, int size)
  b.subMatrix(int i1, int i2, int j1, int j2)
  b.subMatrix(int i1, int i2, int j1, int j2, int istep, int jstep)
  ```

  These work the same as for a `Matrix` (See 5.3), except that the entire subvector or submatrix must be completely within the band.

-
  ```
  b.subBandMatrix(int i1, int i2, int j1, int j2)
  b.subBandMatrix(int i1, int i2, int j1, int j2, int newnlo, int newnhi)
  b.subBandMatrix(int i1, int i2, int j1, int j2, int newnlo, int newnhi,
        int istep, int jstep)
  ```

  This returns a `BandMatrixView` that is a subset of a `BandMatrix`. The first version returns the full band matrix that fits within the rows `i1..i2` and the columns `j1..j2`. If `i1 == j1` and the range is relatively large, then the values of `nlo` and `nhi` for the new submatrix will match the values of the original matrix `b`.

However, if the new size is too small, then the number of bands may be smaller if some off-diagonals would be outside of the range. Likewise, if `i1 != j1`, then `nlo + nhi` will (typically) be preserved, but some sub-diagonals may become super-diagonals or vice versa.

If you want more control over the number of off-diagonals, then the next two versions allow you to specify them explicitly. The final version also allows a non-unit step in each direction.

For example, if `b` is a $6 \times 6$ `BandMatrix` with 2 sub-diagonals and 3 super-diagonals (like our example above), the 3 super-diagonals may be viewed with `b.subBandMatrix(0,5,1,6,0,2)` with `CStyle` indexing or `b.subBandMatrix(1,5,2,6,0,2)` with `FortranStyle` indexing.

-
  ```
  b.rowRange(int i1, int i2)
  b.colRange(int j1, int j2)
  b.diagRange(int k1, int k2)
  ```

  These return a `BandMatrixView` of the parts of these rows, columns or diagonals that appear within the original banded structure. For our example of viewing just the super-diagonals of a $6 \times 6$ `BandMatrix` with 2 sub- and 3 super-diagonals, we could instead use `b.diagRange(1,4)` with `CStyle` indexing or `b.diagRange(1,3)` with `FortranStyle` indexing. The last 3 rows would be `b.rowRange(3,6)` or `b.rowRange(4,6)` respectively. Note that this wold be a $3 \times 5$ matrix with 0 sub-diagonals and 4 super-diagonals. These routines calculate the appropriate changes in the size and shape to include all of the relevant parts of the rows or columns.

-
  ```
  b.upperBand()
  b.lowerBand()
  ```

  These return a `BandMatrixView` including the main diagonal and either the super- or sub-diagonals. The size is set automatically to include the entire band. (This is only non-trivial for non-square band matrices.)

-
  ```
  b.upperBandOff()
  b.lowerBandOff()
  ```

  These return a `BandMatrixView` of only the off-diagonals of either the upper or lower half of the matrix. They are inspired by analogy with the combination `m.upperTri().offDiag()`. Since `BandMatrix` does not have the method `offDiag`, these provide the same functionality.

-
  ```
  b.transpose()
  b.conjugate()
  b.adjoint()
  b.view()
  b.cView()
  b.fView()
  b.realPart()
  b.imagPart()
  ```

  These return `BandMatrixViews`.

## 8.3 Functions

```
RT b.norm1() = Norm1(b)
RT b.norm2() = Norm2(b)
RT b.normInf() = NormInf(b)
RT b.normF() = NormF(b) = b.norm() = Norm(b)
```

```
RT b.normSq() = NormSq(b)
RT b.normSq(RT scale)
RT b.maxAbsElement() = MaxAbsElement(b)
RT b.maxAbs2Element() = MaxAbs2Element(b)
T b.trace() = Trace(b)
T b.sumElements() = SumElements(b)
RT b.sumAbsElements() = SumAbsElements(b)
RT b.sumAbs2Elements() = SumAbs2Elements(b)
T b.det() = Det(b)
RT b.logDet(T* sign=0) = LogDet(b)
bool b.isSingular()
RT b.condition()
RT b.doCondition()
minv = b.inverse() = Inverse(b)
b.makeInverse(Matrix<T>& minv)
b.makeInverseATA(Matrix<T>& cov)
```

The inverse of a `BandMatrix` is not (in general) banded. So `minv` here must be a regular `Matrix`.

```
b.setZero()
b.setAllTo(T x)
b.addToAll(T x)
b.clip(RT thresh)
b.setToIdentity(T x = 1)
b.conjugateSelf()
b.transposeSelf()
Swap(b1,b2)
```

## 8.4 Arithmetic

In addition to `x`, `v`, and `m` from before, we now add `b` for a `BandMatrix`.

```
b2 = -b1
b2 = x * b1
b2 = b1 [*/] x
b3 = b1 [+-] b2
m2 = m1 [+-] b
m2 = b [+-] m1
b [*/]= x
b2 [+-]= b1
m [+-]= b
v2 = b * v1
v2 = v1 * b
v *= b
b3 = b1 * b2
b3 = ElemProd(b1,b2)
m2 = b * m1
m2 = m1 * b
m *= b
b2 = b1 [+-] x
b2 = x [+-] b1
b [+-]= x
```

```
b = x
```

## 8.5 Division

The division operations are:

```
v2 = v1 [/%] b
m2 = m1 [/%] b
m2 = b [/%] m1
m = b1 [/%] b2
m = x [/%] b
v [/%]= b
m [/%]= b
```

`BandMatrix` has three possible choices for the decomposition to use for division:

1. `b.divideUsing(tmv::LU)` does a normal LU decomposition, taking into account the band structure of the matrix, which greatly speeds up the calculation into the lower and upper (banded) triangles. This is the default decomposition to use for a square `BandMatrix` if you don't specify anything.

   This decomposition can only really be done in place if either `nlo` or `nhi` is 0, in which case it is automatically done in place, since the `BandMatrix` is already lower or upper triangle. Thus, there is usually no reason to use the `divideInPlace()` method.

   If this is not the case, and you really want to do the decomposition in place, you can declare a matrix with a wider band and view the portion that represents the matrix you actually want. This view then can be divided in place. More specifically, you need to declare the wider `BandMatrix` with `ColMajor` storage, with the smaller of {`nlo`, `nhi`} as the number of sub-diagonals, and with (`nlo` + `nhi`) as the number of super-diagonals. Then you can use `BandMatrixViewOf` to view the portion you want, transposing it if necessary. On the other hand, you are probably not going to get much of a speed gain from all of this finagling, so unless you are really memory starved, it's probably not worth it.

   To access this decomposition, use:

   ```
   bool b.lud().isTrans()
   tmv::LowerTriMatrix<T,UnitDiag> b.lud().getL()
   tmv::ConstBandMatrixView<T> b.lud().getU()
   const Permutation& b.lud().getP()
   ```

   The following should result in a matrix numerically very close to `b`.

   ```
   tmv::Matrix<T> m2 = b.lud().getP() * b.lud().getL() * b.lud().getU();
   if (b.lud().isTrans()) m2.transposeSelf();
   ```

2. `b.divideUsing(tmv::QR)` will perform a QR decomposition. This is the default method for a non-square `BandMatrix`.

   The same kind of machinations need to be done to perform this in place as for the LU decomposition except that the wider matrix you provide should have at least as many rows as columns. Only if the matrix is square should you make sure that the number of subdiagonals is the smaller of {`nlo`, `nhi`}.

   To access this decomposition, use:[12]

[12] I have not yet made a version of the `PackedQ` class for `BandMatrix`. So unfortunately, here `getQ()` creates the matrix directly and is thus rather inefficient.

```
bool b.qrd().isTrans()
tmv::Matrix<T> b.qrd().getQ()
tmv::ConstBandMatrixView<T> b.qrd().getR()
```

The following should result in a matrix numerically very close to `b`.

```
tmv::Matrix<T> m2(b.nrows,b.ncols);
tmv::MatrixView<T> m2v =
      b.qrd().isTrans() ? b2.transpose() : b2.view();
m2v = b.qrd().getQ() * b.qrd().getR();
```

3. `b.divideUsing(tmv::SV)` will perform a singular value decomposition.

   This cannot be done in place.

   To access this decomposition, use:

   ```
   tmv::ConstMatrixView<T> b.svd().getU()
   tmv::ConstDiagMatrixView<RT> b.svd().getS()
   tmv::ConstMatrixView<T> b.svd().getVt()
   ```

   The product of these three should result in a matrix numerically very close to `b`.

   There are the same control and access routines as for a regular SVD (See 5.8.7),

   ```
   b.svd().thresh(RT thresh)
   b.svd().top(int nsing)
   RT b.svd().condition()
   int b.svd().getKMax()
   ```

The routines

```
b.saveDiv()
b.setDiv()
b.resetDiv()
b.unsetDiv()
bool b.divIsSet()
```

work the same as for regular `Matrix`es. (See 5.8.5.)

And just as for a regular `Matrix`, the functions `b.det()`, `b.logDet()`, and `b.isSingular()` use whichever decomposition is currently set with `b.divideUsing(dt)`, unless `b`'s data type is an integer type, in which case Bareiss's algorithm for the determinant is used.

## 8.6 I/O

The simplest I/O syntax is the usual:

```
os << b;
is >> b;
```

The output format is the same as for a `Matrix`, including all the 0's. (See 5.9.) On input, if any of the elements outside of the band structure are not 0, a `tmv::ReadError` is thrown.

There is also a compact I/O style that puts all the elements that aren't trivially 0 all on a single line and skips the parentheses.

```
os << tmv::CompactIO() << b;
is >> tmv::CompactIO() >> b;
```

This has the extra advantage that it also outputs the number of sub- and super-diagonals, so the `BandMatrix` can be resized correctly if it is not the right size already. The normal I/O style only includes the number of rows and columns, so the number of diagonals is assumed to be correct if the matrix needs to be resized. The compact I/O style can adjust both the size and the number of diagonals.

One can also write small values as 0 with

```
os << tmv::ThreshIO(thresh) << b;
os << tmv::CompactIO().setThresh(thresh) << b;
```

See §15 for more information about specifying custom I/O styles, including features like using brackets instead of parentheses, or putting commas between elements, or specifying an output precision.

# 9 Symmetric and hermitian matrices

The `SymMatrix` class is our symmetric matrix class. A symmetric matrix is one for which $m = m^T$. We also have a class called `HermMatrix`, which is our hermitian matrix class. A hermitian matrix is one for which $m = m^\dagger$. The two are exactly the same if `T` is real, but for complex `T`, they are different.

One general caveat about complex `HermMatrix` calculations is that the diagonal elements should all be real. Some calculations assume this, and only use the real part of the diagonal elements. Other calculations use the full complex value as it is in memory. Therefore, if you set a diagonal element to a non-real value at some point, the results will likely be wrong in unpredictable ways. Plus of course, your matrix won't actually be hermitian any more, so the right answer is undefined in any case.

All the `SymMatrix` and `HermMatrix` routines are included by:

```
#include "TMV_Sym.h"
```

In addition to the data type template parameter (indicated here by `T` as usual), there is also a second template parameter that specifies attributes of the `SymMatrix` or `HermMatrix`. The attributes that are allowed are:

- `CStyle` or `FortranStyle`
- `ColMajor` or `RowMajor`
- `Lower` or `Upper`

The final option refers to which triangle the data are actually stored in, since the other half of the values are identical, so we do not need to reference them. The default attributes are `CStyle`, `ColMajor` and `Lower`.

The storage of a `SymMatrix` takes $N \times N$ elements of memory, even though approximately half of them are never used. Someday, I'll write the packed storage versions, which will allow for efficient storage of the matrices.

Usually, the symmetric/hermitian distinction does not affect the use of the classes. (It does affect the actual calculations performed of course.) So we will use `s` for both, and just point out whenever a `HermMatrix` acts differently from a `SymMatrix`.

Most functions and methods for `SymMatrix` and `HermMatrix` work the same as they do for `Matrix`. In these cases, we will just list the functions that are allowed with the effect understood to be the same as for a regular `Matrix`. Of course, there are almost always algorithmic speed-ups, which the code will use to take advantage of the symmetric or hermitian structure. Whenever there is a difference in how a function works, we will explain the difference.

## 9.1 Constructors

As usual, the optional `A` template argument specifies attributes about the `SymMatrix`. The default attributes if not specified are `CStyle|ColMajor|Lower`.

-
  ```
  tmv::SymMatrix<T,A> s()
  tmv::HermMatrix<T,A> h()
  ```

  Makes a `SymMatrix` or `HermMatrix` with zero size. You would normally use the `resize` function later to change the size to some useful value.

-
  ```
  tmv::SymMatrix<T,A> s(int n)
  tmv::HermMatrix<T,A> h(int n)
  ```

  Makes an `n` $\times$ `n` `SymMatrix` or `HermMatrix` with *uninitialized* values. If extra debugging is turned on (with the compiler flag `-DTMV_EXTRA_DEBUG`), then the values are in fact initialized to 888.

-
  ```
  tmv::SymMatrix<T,A> s(int n, T x)
  tmv::HermMatrix<T,A> h(int n, RT x)
  ```

Makes an n × n SymMatrix or HermMatrix with all values equal to x. For the HermMatrix version of this, x must be real.

-
  ```
  tmv::SymMatrix<T,A> s(const Matrix<T,A2>& m)
  tmv::HermMatrix<T,A> h(const Matrix<T,A2>& m)
  tmv::SymMatrix<T,A> s(const DiagMatrix<T,A2>& d)
  tmv::HermMatrix<T,A> h(const DiagMatrix<T,A2>& d)
  ```

  Makes a SymMatrix or HermMatrix which copies the corresponding values of m.

-
  ```
  tmv::SymMatrix<T,A> s1(const SymMatrix<T2,A2>& s2)
  tmv::HermMatrix<T,A> h1(const HermMatrix<T2,A2>& h2)
  s1 = s2
  ```

  Copy s2 or h2 respectively, which may be of any type T2 so long as values of type T2 are convertible into type T. The assignment operator has the same flexibility.

-
  ```
  tmv::SymMatrixView<T,A> s =
        tmv::SymMatrixViewOf(MatrixView<T> m, UpLoType uplo)
  tmv::ConstSymMatrixView<T,A> s =
        tmv::SymMatrixViewOf(ConstMatrixView<T> m, UpLoType uplo)
  tmv::SymMatrixView<T,A> h =
        tmv::HermMatrixViewOf(MatrixView<T>& m, UpLoType uplo)
  tmv::ConstSymMatrixView<T,A> h =
        tmv::HermMatrixViewOf(ConstMatrixView<T>& m, UpLoType uplo)
  ```

  Make a SymMatrixView of the corresponding portion of m. uplo should be either Lower or Upper to indicate which portion of m you want to view as symmetric.

-
  ```
  tmv::SymMatrixView<T,A> s =
        tmv::SymMatrixViewOf(T* vv, int n, UpLoType uplo,
        StorageType stor)
  tmv::ConstSymMatrixView<T,A> s =
        tmv::SymMatrixViewOf(const T* vv, int n, UpLoType uplo,
        StorageType stor)
  tmv::SymMatrixView<T,A> h =
        tmv::HermMatrixViewOf(T* vv, int n, UpLoType uplo,
        StorageType stor)
  tmv::ConstSymMatrixView<T,A> h =
        tmv::HermMatrixViewOf(const T* vv, int n, UpLoType uplo,
        StorageType stor)
  ```

  Make a SymMatrixView of the actual memory elements, vv, in either the upper or lower triangle. vv must be of length n × n, even though only about half of the values are actually used,

-
  ```
  tmv::SymMatrixView<T,A> s =
        tmv::SymMatrixViewOf(T* vv, int n, UpLoType uplo, int stepi,
        int stepj)
  tmv::ConstSymMatrixView<T,A> s =
        tmv::SymMatrixViewOf(const T* vv, int n, UpLoType uplo,
        int stepi, int stepj)
  tmv::SymMatrixView<T,A> h =
  ```

```
      tmv::HermMatrixViewOf(T* vv, int n, UpLoType uplo, int stepi,
      int stepj)
tmv::ConstSymMatrixView<T,A> h =
      tmv::HermMatrixViewOf(const T* vv, int n, UpLoType uplo,
      int stepi, int stepj)
```

Make a `SymMatrixView` of the actual memory elements, `vv`, in either the upper or lower triangle. This allows for arbitrary steps through the data.

## 9.2 Access

-
  ```
  s.nrows() = s.ncols() = s.colsize() = s.rowsize() = s.size()
  s.resize(int new_size)
  s(i,j)
  s.cref(i,j)
  s.ref(i,j)
  ```

  The only difference here from the corresponding methods for a `Matrix` is that a complex `HermMatrix` has a special reference type, since it needs to check whether its value is the conjugate of the value actually stored in memory. This will necessarily be true for values in either the upper or lower triangle.

-
  ```
  s.row(int i, int j1, int j2)
  s.col(int i, int j1, int j2)
  s.diag()
  s.diag(int i)
  s.diag(int i, int k1, int k2)
  ```

  As for triangle matrices, the versions of `row` and `col` with only one argument are missing, since the full row or column isn't accessible as a `VectorView`. You must specify a valid range within the row or column that you want, given the upper or lower storage of `s`. The diagonal element may be in a `VectorView` with either elements in the lower triangle or the upper triangle, but not both. To access a full row, you would therefore need to use two steps:

  ```
  s.row(i,0,i) = ...
  s.row(i,i,ncols) = ...
  ```

-
  ```
  T* s.ptr()
  const T* s.cptr() const
  int s.stepi() const
  int s.stepj() const
  bool s.isconj() const
  bool s.issym() const
  bool s.isherm() const
  bool s.isupper() const
  bool s.isrm() const
  bool s.iscm() const
  ```

  These methods allow for direct memory access of a `SymMatrix`. There are a few new methods here: `issym()` returns whether `s` is symmetric (as opposed to hermitian), `isherm()` returns whether it is hermitian (as opposed to symmetric), and `isupper()` returns whether the the elements actually stored in memory are for the upper triangle (as opposed to the lower triangle). Note that for a real `SymMatrix` both `issym()` and `isherm()` will return `true`.

- `s.subVector(int i, int j, int istep, int jstep, int size)`
  `s.subMatrix(int i1, int i2, int j1, int j2)`
  `s.subMatrix(int i1, int i2, int j1, int j2, int istep, int jstep)`

  These work the same as for a `Matrix` (See 5.3.) except that the entire subvector or submatrix must be completely within either the upper or lower triangle.

- `s.subSymMatrix(int i1, int i2, int istep = 1)`

  This returns a `SymMatrixView` of `s` whose upper-left corner is `s(i1,i1)`, and whose lower-right corner is `s(i2-istep,i2-istep)` for `CStyle` or `s(i2,i2)` for `FortranStyle`. If `istep` $\neq 1$, then it is the step in both the `i` and `j` directions.

- `s.upperTri(DiagType dt=NonUnitDiag)`
  `s.lowerTri(DiagType dt=NonUnitDiag)`
  `s.unitUpperTri()`
  `s.unitLowerTri()`

  All of these are valid, regardless of which triangle stores the actual data for `s`.

- `s.transpose()`
  `s.conjugate()`
  `s.adjoint()`
  `s.view()`
  `s.cView()`
  `s.fView()`
  `s.realPart()`
  `s.imagPart()`

  These return `SymMatrixViews`. Note that the imaginary part of a complex hermitian matrix is skew-symmetric, so `s.imagPart()` is illegal for a `HermMatrix`. If you need to deal with the imaginary part of a `HermMatrix`, you can view them as a triangle matrix with `s.upperTri().imagPart()` [13].

Note that there are no iterators provided for `SymMatrix` or `HermMatrix`. The recommended way to iterate over their stored values is to use the `upperTri()` or `lowerTri()` methods and iterate over those portions of the matrix directly.

## 9.3 Functions

```
RT s.norm1() = Norm1(s)
RT s.norm2() = Norm2(s)
RT s.normInf() = NormInf(s)
RT s.normF() = NormF(s) = s.norm() = Norm(s)
RT s.normSq() = NormSq(s)
RT s.normSq(RT scale)
RT s.maxAbsElement() = MaxAbsElement(s)
RT s.maxAbs2Element() = MaxAbs2Element(s)
T s.trace() = Trace(s)
T s.sumElements() = SumElements(s)
RT s.sumAbsElements() = SumAbsElements(s)
```

[13]Or, since the diagonal elements are all real. you could also use `s.upperTri().offDiag().imagPart()`.

```
RT s.sumAbs2Elements() = SumAbs2Elements(s)
T s.det() = Det(s)
RT s.logDet(T* sign=0) = LogDet(s)
bool s.isSingular()
RT s.condition()
RT s.doCondition()
sinv = s.inverse() = Inverse(s)
s.makeInverse(Matrix<T>& sinv)
s.makeInverse(SymMatrix<T>& sinv)
s.makeInverseATA(Matrix<T>& cov)
```

Since the inverse of an `SymMatrix` is also symmetric, the object returned by `s.inverse()` is assignable to a `SymMatrix`. Of course you can also assign it to a regular `Matrix` if you prefer. Similarly, there are versions of `s.makeInverse(minv)` for both argument types.

```
s.setZero()
s.setAllTo(T x)
s.addToAll(T x)
s.clip(RT thresh)
s.setToIdentity(T x = 1)
s.conjugateSelf()
s.transposeSelf()
Swap(s1,s2)
```

The method `transposeSelf()` does nothing to a `SymMatrix` and is equivalent to `conjugateSelf()` for a `HermMatrix`.

```
s.swapRowsCols(int i1, int i2)
```

The new method, `swapRowsCols`, would be equivalent to

```
s.swapRows(i1,i2).swapCols(i1,i2);
```

except that neither of these functions are allowed for a `SymMatrix`, since they result in non-symmetric matrices. Only the combination of both maintains the symmetry of the matrix. So this combination is included as a method.

## 9.4 Arithmetic

In addition to `x`, `v`, and `m` from before, we now add `s` for a `SymMatrix`.

```
s2 = -s1
s2 = x * s1
s2 = s1 [*/] x
s3 = s1 [+-] s2
m2 = m1 [+-] s
m2 = s [+-] m1
s [*/]= x
s2 [+-]= s1
m [+-]= s
v2 = s * v1
v2 = v1 * s
v *= s
m = s1 * s2
s3 = ElemProd(s1,s2)
m2 = s * m1
```

```
m2 = m1 * s
m *= s
s2 = s1 [+-] x
s2 = x [+-] s1
s [+-]= x
s = x
s = v ^ v
s [+-]= v ^ v
s = m * m.transpose()
s [+-]= m * m.transpose()
s = U * U.transpose()
s [+-]= U * U.transpose()
s = L * L.transpose()
s [+-]= L * L.transpose()
```

For outer products, both `v`'s need to be the same actual data. If `s` is complex hermitian, then it should actually be `s = v ^ v.conjugate()`. Likewise for the next three (called "rank-k updates"), the `m`'s, `L`'s and `U`'s need to be the same data, and for a complex hermitian matrix, `transpose()` should be replaced with `adjoint()`.

## 9.5 Delayed evaluation

As usual, the above arithmetic operations try to delay the evaluation of the expression until the storage location is known. Only if the expression gets sufficiently complicated will TMV use a temporary object to store an intermediate result.

However, there is a minor issue with `SymMatrix` arithmetic that is worth knowing about. Because TMV uses the same base class for both hermitian and symmetric matrices, the compiler cannot tell the difference between them in arithmetic operations. (The difference is stored in a variable, so the code knows which kind of matrix it is at runtime.) This can become an issue when doing complicated arithmetic operations with complex hermitian or symmetric matrices.

For some operations, the compiler cannot tell whether the result is necessarily another `SymMatrix` or whether it is merely a regular `Matrix`. For example, a complex scalar times a complex `HermMatrix` is not hermitian (or symmetric), but a complex scalar times a `SymMatrix` is still symmetric. However, since the base classes are the same, and the arithmetic is done in terms of the base classes, the compiler cannot figure out whether the result is symmetric. Likewise the sum of a complex `SymMatrix` and a complex `HermMatrix` is not hermitian or symmetric.

Most of the time, this won't matter, since you will generally assign the result to either a `SymMatrix` or a `Matrix` as appropriate, and the code can check whether the assignment is valid at runtime. However, if you let your expression get more complicated than a single matrix addition, multiplication, etc., then some things that should be allowed give compiler errors. For example:

```
s3 += x*s1+s2;
```

is not legal for complex symmetric `s1, s2, s3`, even though this is valid mathematically. This is because there is no 3-matrix `Add` function. (TMV's `AddMM` function just adds a multiple of one matrix to another.) So the right hand side needs to be instantiated before being added to the left side, and it will instantiate as a regular `Matrix`, which cannot be added to a `SymMatrix`. If the "+=" had been just "=", then we wouldn't have any problem, since the composite object that stores `x*s1+s2` is assignable to a `SymMatrix`.

One work-around is to explicitly tell the compiler to instantiate the right hand side as a `SymMatrix`:

```
s3 += SymMatrix<T>(x*s1+s2);
```

Another work-around, which I suspect will usually be preferred, is to break the equation into multiple statements, each of which are simple enough to not require any instantiation:

```
s3 += x*s1;
s3 += s2;
```

The forthcoming version 0.90 which is currently in development has a more sophisticated way of determining how to instantiate a composite object, which will fix this problem.

## 9.6 Division

The division operations are:

```
v2 = v1 [/%] s
m2 = m1 [/%] s
m2 = s [/%] m1
m = s1 [/%] s2
s1 = x [/%] s2
v [/%]= s
m [/%]= s
```

`SymMatrix` has three possible choices for the decomposition to use for division:

1. `m.divideUsing(tmv::LU)` will perform something similar to the LU decomposition for regular matrices. But in fact, it does what is called an LDL or Bunch-Kaufman decomposition.

   A permutation of `m` is decomposed into a lower triangle matrix ($L$) times a symmetric block diagonal matrix ($D$) times the transpose of $L$. $D$ has either 1x1 and 2x2 blocks down the diagonal. For hermitian matrices, the third term is the adjoint of $L$ rather than the transpose.

   This is the default decomposition to use if you don't specify anything.

   To access this decomposition, use:

   ```
   ConstLowerTriMatrixView<T> s.lud().getL()
   BandMatrix<T> s.lud().getD()
   const Permutation& s.lud().getP()
   ```

   The following should result in a matrix numerically very close to `s`.

   ```
   Matrix<T> m2 = s.lud().getP() * s.lud().getL() * s.lud().getD() *
         s.lud().getL().transpose() * s.lud().getP().transpose();
   ```

   For a complex hermitian `s`, you would need to replace `transpose` with `adjoint`.

2. `s.divideUsing(tmv::CH)` will perform a Cholesky decomposition. The matrix `s` must be hermitian (or real symmetric) to use `CH`, since that is the only kind of matrix that has a Cholesky decomposition.

   It is also similar to an LU decomposition, where $U$ is the adjoint of $L$, and there is no permutation. It can be a bit dangerous, since not all hermitian matrices have such a decomposition, so the decomposition could fail. Only so-called "positive-definite" hermitian matrices have a Cholesky decomposition. A positive-definite matrix has all positive real eigenvalues. In general, hermitian matrices have real, but not necessarily positive eigenvalues.

   One example of a positive-definite matrix is $s = A^\dagger A$ where $A$ is any matrix. Then $s$ is guaranteed to be positive-semi-definite (which means some of the eigenvalues may be 0, but not negative). In this case, the routine will usually work, but still might fail from numerical round-off errors if $s$ is nearly singular.

   When the decomposition fails, it throws an object of type `NonPosDef`.

   See §12.4 for some more discussion about positive-definite matrices.

The only advantage of Cholesky over Bunch-Kaufman is speed. (And only about 20 to 30% at that.) If you know your matrix is positive-definite, the Cholesky decomposition is the fastest way to do division.

To access this decomposition, use:

```
ConstLowerTriMatrixView<T> s.chd().getL()
```

The following should result in a matrix numerically very close to `s`.

```
Matrix<T> m2 = s.chd().getL() * s.chd().getL().adjoint()
```

3. `s.divideUsing(tmv::SV)` will perform either an eigenvalue decomposition (for hermitian and real symmetric matrices) or a regular singular value decomposition (for complex symmetric matrices).

   For hermitian matrices (including real symmetric matrices), the eigenvalue decomposition is $H = USU^\dagger$, where $U$ is unitary and $S$ is diagonal. So this would be identical to a singular value decomposition where $V = U^\dagger$, except that the elements of $S$, the eigenvalues of $H$, may be negative.

   However, this decomposition is just as useful for division, dealing with singular matrices just as elegantly. It just means that internally, we allow the values of $S$ to be negative, taking the absolute value when necessary (e.g. for `norm2`). The below access commands finish the calculation of $S$ and $V$ so that the $S(i)$ values are positive.

   To access this decomposition, use:

```
ConstMatrixView<T> s.svd().getU()
DiagMatrix<RT> s.svd().getS()
Matrix<T> s.svd().getVt()
```

   The following should result in a matrix numerically very close to `s`.

```
Matrix<T> m2 = s.svd().getU() * s.svd().getS() * s.svd().getVt()
```

   For a complex symmetric `s`, the situation is not as convenient, since $V \neq U^\dagger$. So for complex symmetric matrices, we just do the normal SVD: $s = USV^\dagger$, although the algorithm does use the symmetry of the matrix to speed up portions of the algorithm relative that that for a general matrix.

   The access is also necessarily different, since the object returned by `s.svd()` implicitly assumes that $V = U$ (modulo some sign changes), so we need a new accessor: `s.symsvd()`. Its `getS` and `getVt` methods return Views rather than instantiated matrices.

   Both versions also have the same control and access routines as a regular SVD (See 5.8.7):

```
s.svd().thresh(RT thresh)
s.svd().top(int nsing)
RT s.svd().condition()
int s.svd().getKMax()
```

   (Likewise for `s.symsvd()`.)

The routines

```
s.saveDiv()
s.setDiv()
s.resetDiv()
s.unsetDiv()
bool s.divIsSet()
s.divideInPlace()
```

work the same as for regular `Matrix`es. (See 5.8.5.) However, it should be noted that for `tmv::SV`, the `divideInPlace` option will use both the memory that is used for the `SymMatrix` as well as the other half of the memory that is normally not used. So if you've done something clever to use the other half of the matrix for something, that will get clobbered too.

And just as for a regular `Matrix`, the functions `s.det()`, `s.logDet()`, and `s.isSingular()` use whichever decomposition is currently set with `s.divideUsing(dt)`, unless `s`'s data type is an integer type, in which case Bareiss's algorithm for the determinant is used.

## 9.7 I/O

The simplest I/O syntax is the usual:

```
os << s;
is >> s;
```

The output format is the same as for a `Matrix`. (See 5.9.) On input, if the matrix read in is not symmetric (or Hermitian as appropriate), then a `tmv::ReadError` is thrown.

There is also a compact I/O style that puts all the elements in the lower triangle all on a single line and skips the parentheses.

```
os << tmv::CompactIO() << s;
is >> tmv::CompactIO() >> s;
```

One can also write small values as 0 with

```
os << tmv::ThreshIO(thresh) << s;
os << tmv::CompactIO().setThresh(thresh) << s;
```

See §15 for more information about specifying custom I/O styles, including features like using brackets instead of parentheses, or putting commas between elements, or specifying an output precision.

## 9.8 Other operations

There are three more arithmetic routines that we provide for `SymMatrix`, which do not have any corresponding shorthand with the usual arithmetic operators.

The first two are:

```
Rank2Update<bool add>(T x, const Vector<T1>& v1, const Vector<T2>& v2,
      SymMatrix<T>& s)
Rank2KUpdate<bool add>(T x, const Matrix<T1>& m1, const Matrix<T2>& m2,
      SymMatrix<T>& s)
```

They are similar to the `Rank1Update` and `RankKUpdate` routines, which are implemented in TMV with the expressions `s += x * v ^ v` and `s += x * m * m.transpose()`.

A rank-2 update calculates

```
s (+=) x * ((v1 ^ v2) + (v2 ^ v1))
s (+=) x * (v1 ^ v2.conjugate()) + conj(x) * (v2 ^ v1.conjugate())
```

for a symmetric or hermitian `s` respectively, where "(+=)" means "+=" if `add` is `true` and "=" if `add` is `false`. Likewise, a rank-2k update calculates:

```
s (+=) x * (m1 * m2.transpose() + m2 * m1.transpose())
s (+=) x * m1 * m2.adjoint() + conj(x) * m2 * m1.adjoint()
```

for a symmetric or hermitian `s` respectively.

We don't have an arithmetic operator shorthand for these, because, as you can see, the operator overloading required would be quite complicated. And since they are pretty rare, I decided to just let the programmer call the routines explicitly.

The other routine is:

```
SymMultMM<bool add>(T x, const Matrix<T>& m1, const Matrix<T>& m2,
      SymMatrix<T>& s)
```

This calculates the usual generalized matrix product: `s (+=) x * m1 * m2`, but it basically asserts that the product `m1 * m2` is symmetric (or hermitian as appropriate).

Since a matrix product is not in general symmetric, I decided not to allow this operation with just the usual operators to prevent the user from doing this accidentally. However, there are times when the programmer can know that the product should be (at least numerically close to) symmetric and that this calculation is ok. Therefore it is provided as a subroutine.

Note: All three of these functions may also take views as their arguments. They don't have to be instantiated objects.

# 10 Symmetric and hermitian band matrices

The `SymBandMatrix` class is our symmetric band matrix, which combines the properties of `SymMatrix` and `BandMatrix`; it has a banded structure and $m = m^T$. Likewise `HermBandMatrix` is our hermitian band matrix for which $m = m^\dagger$.

As with the documentation for `SymMatrix`/`HermMatrix`, the descriptions below will only be written for `SymBandMatrix` with the implication that a `HermBandMatrix` has the same functionality, but with the calculations appropriate for a hermitian matrix, rather than symmetric.

One general caveat about complex `HermBandMatrix` calculations is that the diagonal elements should all be real. Some calculations assume this, and only use the real part of the diagonal elements. Other calculations use the full complex value as it is in memory. Therefore, if you set a diagonal element to a non-real value at some point, the results will likely be wrong in unpredictable ways. Plus of course, your matrix will not actually be hermitian any more, so the right answer is undefined in any case.

All the `SymBandMatrix` and `HermBandMatrix` routines are included by:

```
#include "TMV_SymBand.h"
```

In addition to the data type template parameter (indicated here by `T` as usual), there is also a second template parameter that specifies attributes of the `SymBandMatrix` or `HermBandMatrix`. The attributes that are allowed are:

- `CStyle` or `FortranStyle`
- `ColMajor`, `RowMajor`, or `DiagMajor`
- `Lower` or `Upper`

The default attributes are `CStyle`, `ColMajor` and `Lower`.

The storage size required is the same as for the `BandMatrix` of the upper or lower band portion. (See 8.) As with square band matrices, all three storage methods always need the same amount of memory, so the decision about which method to use should generally be based on performance considerations rather than memory usage. The speed of the various matrix operations are different for the different storage types. If the matrix calculation speed is important, it may be worth trying all three to see which is fastest for the operations you are using.

Also, as with `BandMatrix`, the storage for `Lower`, `DiagMajor` does not start with the upper left element as usual. Rather, it starts at the start of the lowest sub-diagonal.

Most functions and methods for `SymBandMatrix` and `HermBandMatrix` work the same as they do for `Matrix`. In these cases, we will just list the functions that are allowed with the effect understood to be the same as for a regular `Matrix`. Of course, there are almost always algorithmic speed-ups, which the code will use to take advantage of the symmetric (or hermitian) banded structure. Whenever there is a difference in how a function works, we will explain the difference.

## 10.1 Constructors

As usual, the optional `A` template argument specifies attributes about the `SymBandMatrix`. The default attributes if not specified are `CStyle|ColMajor|Lower`.

-
  ```
  tmv::SymBandMatrix<T,A> sb()
  tmv::HermBandMatrix<T,A> hb()
  ```

  Makes a `SymBandMatrix` or `HermBandMatrix` with zero size. You would normally use the `resize` function later to change the size to some useful value.

-
  ```
  tmv::SymBandMatrix<T,A> sb(int n, int nlo)
  tmv::HermBandMatrix<T,A> hb(int n, int nlo)
  ```

Makes an `n` $\times$ `n` `SymBandMatrix` or `HermBandMatrix` with `nlo` off-diagonals and with *uninitialized* values. If extra debugging is turned on (with the compiler flag `-DTMV_EXTRA_DEBUG`), then the values are initialized to 888.

-
  ```
  tmv::SymBandMatrix<T,A> sb(int n, int nlo, T x)
  tmv::HermBandMatrix<T,A> hb(int n, int nlo, RT x)
  ```

  Makes an `n` $\times$ `n` `SymBandMatrix` or `HermBandMatrix` with `nlo` off-diagonals and with all values equal to `x`. For the `HermBandMatrix` version of this, `x` must be real.

-
  ```
  tmv::SymBandMatrix<T,A> sb(const Matrix<T,A2>& m, int nlo)
  tmv::SymBandMatrix<T,A> sb(const SymMatrix<T,A2>& m, int nlo)
  tmv::SymBandMatrix<T,A> sb(const BandMatrix<T,A2>& m, int nlo)
  tmv::SymBandMatrix<T,A> sb(const SymBandMatrix<T,A2>& m, int nlo)
  ```

  Makes a `SymBandMatrix` which copies the corresponding values of `m`. For the last two, `nlo` must not be larger than the number of upper or lower bands in `m`. There are similar constructors for `HermBandMatrix`.

-
  ```
  tmv::SymBandMatrix<T,DiagMajor> sb = SymTriDiagMatrix(
        const Vector<T>& v1, const Vector<T>& v2)
  tmv::SymBandMatrix<T,DiagMajor> sb = SymTriDiagMatrix(
        const Vector<T>& v1, const Vector<T>& v2)
  tmv::HermBandMatrix<T,DiagMajor> hb = HermTriDiagMatrix(
        const Vector<T>& v1, const Vector<T>& v2,
        UpLoType uplo)
  tmv::HermBandMatrix<T,DiagMajor> hb = HermTriDiagMatrix(
        const Vector<T>& v1, const Vector<T>& v2,
        UpLoType uplo)
  ```

  Shorthand to create a symmetric tri-diagonal band matrix if you already have the `Vectors`. The main diagonal is `v1` and the off-diagonal is `v2`.

  With a `HermTriDiagMatrix`, `v1` should be real, although it may be either a real-valued `Vector` or a complex-valued `Vector` whose imaginary components are all zero. Also, `HermTriDiagMatrix` takes an extra parameter, `uplo`, indicating whether `v2` should be used as the upper or lower off-diagonal (since they differ by a conjugation).

-
  ```
  tmv::SymBandMatrix<T,A> sb1(const SymBandMatrix<T2,A2>& sb2)
  sb1 = sb2
  ```

  Copy the `SymBandMatrix m2`, which may be of any type `T2` so long as values of type `T2` are convertible into type `T`. The assignment operator has the same flexibility.

-
  ```
  tmv::SymBandMatrixView<T,A> sb =
        SymBandMatrixViewOf(MatrixView<T> m, UpLoType uplo, int nlo)
  tmv::SymBandMatrixView<T,A> sb =
        SymBandMatrixViewOf(SymMatrixView<T> m, int nlo)
  tmv::SymBandMatrixView<T,A> sb =
        SymBandMatrixViewOf(BandMatrixView<T> m, UpLoType uplo, int nlo)
  tmv::SymBandMatrixView<T,A> hb =
        HermBandMatrixViewOf(MatrixView<T> m, UpLoType uplo, int nlo)
  tmv::SymBandMatrixView<T,A> hb =
        HermBandMatrixViewOf(SymMatrixView<T> m, int nlo)
  ```

```
tmv::SymBandMatrixView<T,A> hb =
      HermBandMatrixViewOf(BandMatrixView<T> m, UpLoType uplo, int nlo)
```

Make an `SymBandMatrixView` of the corresponding portion of `m`. To view these as a hermitian band matrix, use the command, `HermBandMatrixViewOf` instead. For the view of a `BandMatrix`, the parameter `nlo` may be omitted, in which case either `m.nhi()` or `m.nlo()` is used according to whether `uplo` is `Upper` or `Lower` respectively. There are also `ConstSymBandMatrixView` versions of these.

-
  ```
  tmv::SymBandMatrixView<T,A> sb =
        tmv::SymBandMatrixViewOf(T* vv, int n, int nlo,
            UpLoType uplo, StorageType stor)
  tmv::ConstSymBandMatrixView<T,A> sb =
        tmv::SymBandMatrixViewOf(const T* vv, int n, int nlo,
            UpLoType uplo, StorageType stor)
  tmv::SymBandMatrixView<T,A> hb =
        tmv::HermBandMatrixViewOf(T* vv, int n, int nlo,
            UpLoType uplo, StorageType stor)
  tmv::ConstSymBandMatrixView<T,A> hb =
        tmv::HermBandMatrixViewOf(const T* vv, int n, int nlo,
            UpLoType uplo, StorageType stor)
  ```

  Make a `SymBandMatrixView` of the actual memory elements, `vv`, in either the upper or lower band. The length of the data array should be `BandStorageLength(stor,n,n,nlo,0)`. Also, as with the corresponding `BandMatrixViewOf` functions, if `uplo` is `Lower` and `stor` is `DiagMajor` then `vv` should be the memory address of `b(nlo,0)`, not `b(0,0)`.

-
  ```
  tmv::SymBandMatrixView<T,A> sb =
        tmv::SymBandMatrixViewOf(T* vv, int n, int nlo,
            UpLoType uplo, int stepi, int stepj)
  tmv::ConstSymBandMatrixView<T,A> sb =
        tmv::SymBandMatrixViewOf(const T* vv, int n, int nlo,
            UpLoType uplo, int stepi, int stepj)
  tmv::SymBandMatrixView<T,A> hb =
        tmv::HermBandMatrixViewOf(T* vv, int n, int nlo,
            UpLoType uplo, int stepi, int stepj)
  tmv::ConstSymBandMatrixView<T,A> hb =
        tmv::HermBandMatrixViewOf(const T* vv, int n, int nlo,
            UpLoType uplo, int stepi, int stepj)
  ```

  Make a `SymBandMatrixView` of the actual memory elements, `vv`, in either the upper or lower band. This allows for arbitrary steps through the data.

## 10.2 Access

-
  ```
  sb.nrows() = sb.ncols() = sb.colsize() = sb.rowsize() = sb.size()
  sb.nlo() = sb.nhi()
  sb.resize(int new_size, int new_nlo)
  sb(i,j)
  sb.cref(i,j)
  sb.ref(i,j)
  ```

As with a complex `HermMatrix`, a complex `HermBandMatrix` has a special reference type, since it needs to check whether its value is the conjugate of the value actually stored in memory. This will necessarily be true for values in either the upper or lower band. Also, for the mutable `sb(i,j)` version, the position must fall within the band. If `sb` is not mutable, then `sb(i,j)` returns 0 for positions outside of the band.

-
  ```
  sb.row(int i, int j1, int j2)
  sb.col(int i, int j1, int j2)
  sb.diag()
  sb.diag(int i)
  sb.diag(int i, int k1, int k2)
  ```

  Again, the versions of `row` and `col` with only one argument are missing, since the full row or column isn't accessible as a `VectorView`. You must specify a valid range within the row or column that you want, given the banded storage of `sb`. And, like for `SymMatrix`, a full row must be accessed in its two parts, one on each side of the diagonal.

-
  ```
  T* sb.ptr()
  const T* sb.cptr() const
  int sb.stepi() const
  int sb.stepj() const
  int sb.diagstep() const
  bool sb.isconj() const
  bool sb.issym() const
  bool sb.isherm() const
  bool sb.isupper() const
  bool sb.isrm() const
  bool sb.iscm() const
  bool sb.isdm() const
  ```

  These methods allow for direct memory access of a `SymBandMatrix`.

-
  ```
  sb.subVector(int i, int j, int istep, int jstep, int size)
  sb.subMatrix(int i1, int i2, int j1, int j2)
  sb.subMatrix(int i1, int i2, int j1, int j2, int istep, int jstep)
  sb.subBandMatrix(int i1, int i2, int j1, int j2)
  sb.subBandMatrix(int i1, int i2, int j1, int j2, int newnlo, int newhi)
  sb.subBandMatrix(int i1, int i2, int j1, int j2, int newnlo, int newhi,
        int istep, int jstep)
  sb.diagRange(int k1, int k2)
  sb.upperBand()
  sb.lowerBand()
  sb.upperBandOff()
  sb.lowerBandOff()
  ```

  These work the same as for a `BandMatrix` (See 8.2), except that the entire subvector or submatrix must be completely within the upper or lower band.

-
  ```
  sb.subSymMatrix(int i1, int i2)
  sb.subSymMatrix(int i1, int i2, int istep)
  sb.subSymBandMatrix(int i1, int i2, int newnlo=m.nlo())
  sb.subSymBandMatrix(int i1, int i2, int newnlo, int istep)
  ```

These return a view of the `SymMatrix` or `SymBandMatrix` which runs from `i1` to `i2` along the diagonal with an optional step, and includes the off-diagonals in the same rows/cols. For the first two, the `SymMatrix` must be completely with the band.

- `sb.symDiagRange(int newnlo)`

  Since `diagRange` returns a regular `BandMatrixView`, it must be completely within either the upper or lower band. This routine returns a `SymBandMatrixView` which straddles the diagonal with `newnlo` super- and sub-diagonals.

-
  ```
  sb.transpose()
  sb.conjugate()
  sb.adjoint()
  sb.view()
  sb.cView()
  sb.fView()
  sb.realPart()
  sb.imagPart()
  ```

  These return `SymBandMatrixViews`. Note that the imaginary part of a complex hermitian band matrix is skew-symmetric, so `sb.imagPart()` is illegal for a `HermBandMatrix`. If you need to manipulate the imaginary part of a `HermMatrix`, you could use `sb.upperBandOff().imagPart()` (since all the diagonal elements are real).

Note that there are no iterators provided for `SymBandMatrix` or `HermBandMatrix`. The recommended way to iterate over their stored values is to use the `upperBand()` or `lowerBand()` methods and iterate over those portions of the matrix directly.

## 10.3 Functions

```
RT sb.norm1() = Norm1(sb)
RT sb.norm2() = Norm2(sb)
RT sb.normInf() = NormInf(sb)
RT sb.normF() = NormF(sb) = sb.norm() = Norm(sb)
RT sb.normSq() = NormSq(sb)
RT sb.normSq(RT scale)
RT sb.maxAbsElement() = MaxAbsElement(sb)
RT sb.maxAbs2Element() = MaxAbs2Element(sb)
T sb.trace() = Trace(sb)
T sb.sumElements() = SumElements(sb)
RT sb.sumAbsElements() = SumAbsElements(sb)
RT sb.sumAbs2Elements() = SumAbs2Elements(sb)
T sb.det() = Det(sb)
RT sb.logDet(T* sign=0) = LogDet(sb)
sinv = sb.inverse() = Inverse(sb)
bool sb.isSingular
RT sb.condition()
RT sb.doCondition()
sb.makeInverse(Matrix<T>& minv)
sb.makeInverse(SymMatrix<T>& sinv)
sb.makeInverseATA(Matrix<T>& cov)
```

The inverse of a SymBandMatrix is not (in general) banded. However, it is symmetric (or hermitian). So sb.inverse() may be assigned to either a Matrix or a SymMatrix.

```
sb.setZero()
sb.setAllTo(T x)
sb.addToAll(T x)
sb.clip(RT thresh)
sb.setToIdentity(T x = 1)
sb.conjugateSelf()
sb.transposeSelf()
Swap(sb1,sb2)
```

As with SymMatrix, the method transposeSelf() does nothing to a SymBandMatrix and is equivalent to conjugateSelf() for a HermBandMatrix.

## 10.4 Arithmetic

In addition to x, v, m, b and s from before, we now add sb for a SymBandMatrix.

```
sb2 = -sb1
sb2 = x * sb1
sb2 = sb1 [*/] x
sb3 = sb1 [+-] sb2
m2 = m1 [+-] sb
m2 = sb [+-] m1
b2 = b1 [+-] sb
b2 = sb [+-] b1
s2 = s1 [+-] sb
s2 = sb [+-] s1
sb [*/]= x
sb2 [+-]= sb1
m [+-]= sb
b [+-]= sb
s [+-]= sb
v2 = sb * v1
v2 = v1 * sb
v *= sb
b = sb1 * sb2
sb3 = ElemProd(sb1,sb2)
m2 = sb * m1
m2 = m1 * sb
m *= sb
b2 = sb * b1
b2 = b1 * sb
b *= sb
m2 = sb * s1
m2 = s1 * sb
sb2 = sb1 [+-] x
sb2 = x [+-] sb1
sb [+-]= x
sb = x
```

## 10.5 Division

The division operations are:

```
v2 = v1 [/%] sb
m2 = m1 [/%] sb
m2 = sb [/%] m1
m = sb1 [/%] sb2
s = x [/%] sb
v [/%]= sb
m [/%]= sb
```

`SymBandMatrix` has three possible choices for the division decomposition:

1. `m.divideUsing(tmv::LU)` actually does the `BandMatrix` version of LU, rather than a Bunch-Kaufman algorithm like for `SymMatrix`. The reason is that the pivots in the Bunch-Kaufman algorithm can arbitrarily expand the band width required to hold the information. The generic banded LU algorithm is limited to 3*`nlo`+1 bands.

   To access this decomposition, use:

   ```
   bool sb.lud().isTrans()
   tmv::LowerTriMatrix<T,UnitDiag> sb.lud().getL()
   tmv::ConstBandMatrixView<T> sb.lud().getU()
   const Permutation& sb.lud().getP()
   ```

   The following should result in a matrix numerically very close to `sb`.

   ```
   tmv::Matrix<T> m2(sb.nrows(),sb.ncols);
   tmv::MatrixView<T> m2v =
         sb.lud().isTrans() ? m2.transpose() : m2.view();
   m2v = sb.lud().getP() * sb.lud().getL() * sb.lud().getU();
   ```

2. `sb.divideUsing(tmv::CH)` will perform a Cholesky decomposition. `sb` must be hermitian (or real symmetric) to use `CH`, since that is the only kind of matrix that has a Cholesky decomposition.

   As with a regular `SymMatrix`, the only real advantage of Cholesky over LU decomposition is speed. If you know your matrix is positive-definite, the Cholesky decomposition is the fastest way to do division.

   If `sb` is tri-diagonal (i.e. `nlo` = 1), then we use a slightly different algorithm, which avoids the square roots required for a Cholesky decomposition. Namely, we form the decomposition $sb = LDL^\dagger$, where $L$ is a unit-diagonal lower banded matrix with 1 sub-diagonal, and $D$ is diagonal.

   If `sb` has `nlo` $> 1$, then we just use a normal Cholesky algorithm where $sb = LL^\dagger$ and $L$ is lower banded with the same `nlo` as `sb`.

   Both versions of the algorithm are accessed with the same methods:

   ```
   BandMatrix<T> sb.chd().getL()
   DiagMatrix<T> sb.chd().getD()
   ```

   with $L$ being made unit-diagonal or $D$ being set to the identity matrix as appropriate. (Obviously, `getL()` contains all of the information for the non-tridiagonal version.)

   The following should result in a matrix numerically very close to `sb`.

   ```
   Matrix<T> m2 = sb.chd().getL() * sb.chd().getD() *
         sb.chd().getL().adjoint()
   ```

3. `sb.divideUsing(tmv::SV)` will perform either an eigenvalue decomposition (for hermitian band and real symmetric band matrices) or a regular singular value decomposition (for complex symmetric band matrices).

   To access this decomposition, use:

   ```
   ConstMatrixView<T> sb.svd().getU()
   DiagMatrix<RT> sb.svd().getS()
   Matrix<T> sb.svd().getVt()
   ```

   (As for `SymMatrix`, a complex symmetric matrix needs to use the accessor `symsvd()` instead, whose `getS` and `getVt` methods return Views rather than instantiated matrices.)

   The following should result in a matrix numerically very close to `sb`.

   ```
   Matrix<T> m2 = sb.svd().getU() * sb.svd().getS() * sb.svd().getVt()
   ```

   Both versions also have the same control and access routines as a regular SVD: (See 5.8.7):

   ```
   sb.svd().thresh(RT thresh)
   sb.svd().top(int nsing)
   RT sb.svd().condition()
   int sb.svd().getKMax()
   ```

   (Likewise for `sb.symsvd()`.)

The routines

```
sb.saveDiv()
sb.setDiv()
sb.resetDiv()
sb.unsetDiv()
bool sb.divIsSet()
sb.divideInPlace()
```

work the same as for regular `Matrix`es. (See 5.8.5.) However, `divideInPlace` only actually works for the `CH` algorithm.

And just as for a regular `Matrix`, the functions `sb.det()`, `sb.logDet()`, and `sb.isSingular()` use whichever decomposition is currently set with `sb.divideUsing(dt)`, unless `sb`'s data type is an integer type, in which case Bareiss's algorithm for the determinant is used.

## 10.6 I/O

The simplest I/O syntax is the usual:

```
os << sb;
is >> sb;
```

The output format is the same as for a `Matrix`. (See 5.9.) On input, if the matrix read in is not symmetric (or Hermitian as appropriate), then a `tmv::ReadError` is thrown. Likewise if any of the elements outside of the band structure are not 0.

There is also a compact I/O style that puts all the elements in the lower band all on a single line and skips the parentheses.

```
os << tmv::CompactIO() << sb;
is >> tmv::CompactIO() >> sb;
```

This has the extra advantage that it also outputs the number of off-diagonals, so the `SymBandMatrix` can be resized correctly if it is not the right size already. The normal I/O style only includes the number of rows and columns, so the number of diagonals is assumed to be correct if the matrix needs to be resized. The compact I/O style can adjust both the size and the number of diagonals.

One can also write small values as 0 with

```
os << tmv::ThreshIO(thresh) << sb;
os << tmv::CompactIO().setThresh(thresh) << sb;
```

See §15 for more information about specifying custom I/O styles, including features like using brackets instead of parentheses, or putting commas between elements, or specifying an output precision.

# 11 Permutations

The `Permutation` class is our permutation matrix class. A permutation matrix is a square matrix with exactly one 1 in each row and column, and all the rest of the elements equal to 0.

However, internally we do not store a permutation this way. Instead, we treat a permutation as a series of pair-wise interchanges. This seems to be the fastest way to apply a permutation to a matrix or vector, rather than using an index-based method.

Also, I didn't bother to have a `PermutationView` class. Instead, the `Permutation` object keeps track of whether it owns its data or is just referencing values kept somewhere else. Whenever you perform a mutable action on the object, it copies the values if necessary. So you cannot indirectly modify another `Permutation` the way you can with `MatrixView`.

## 11.1 Constructors

- `tmv::Permutation p()`

  Makes a `Permutation` with zero size. You would normally use the `resize` function later to change the size to some useful value.

- `tmv::Permutation p(int n)`

  Makes an `n` × `n` `Permutation` set initially to the identity matrix.

- `tmv::Permutation p(int n, const ptrdiff_t* pp, bool isinv=false)`

  Makes an `n` × `n` `Permutation` using the provided values as the list of interchanges. The meaning of `pp` is that `v=p*v` is equivalent to

  ```
  if (isinv) {
      for(int i=n-1; i>=0; --i) v.swap(i,pp[i]);
  } else {
      for(int i=0; i<n; ++i) v.swap(i,pp[i]);
  }
  ```

  If `isinv` is omitted, it is taken to be `false`.

  Note: most of the parameter values to TMV methods that are listed as `int` are actually `ptrdiff_t`. This might be the same as `int` on your system, or it might be equivalent to `long`. But the difference is normally completely transparent to the user, since the compiler will seemlessly convert between integer types as needed. However, since we are dealing with an array of index values, the compiler can do the conversion. You need to correctly use `ptrdiff_t` rather than `int`.

  The reason we use `ptrdiff_t` is in case `int` is only 32 bits and you allocate a matrix with more than $2^{31}$ elelements, then `int` would overflow, but `ptrdiff_t` is guaranteed to be safe. So we use `ptrdiff_t` for all memory offsets in the code, and to be consistent, all the parameters that deal with indices are `ptrdiff_t` as well. I think this constructor is the only case where you actually need to know about this fact.

## 11.2 Access

```
p.resize(int new_size)

p.nrows() = p.ncols() = p.colsize() = p.rowsize() = p.size()
p(i,j)
p.cref(i,j)
```

Note: Because of the way the permutation is stored, `p(i,j)` is not terribly efficient. It takes $O(N)$ time to calculate. Also, there is no mutable version like there is for most matrices.

```
p.transpose() = p.inverse()
```

These are the same, and they do not create new storage. So statements like `v = p.transpose() * v` are efficient.

```
const ptrdiff_t* p.getValues()
```

Get the indices of the interchanges. These are equivalent to the `pp` values described above for the constructor.

```
bool p.isInverse()
```

Returns true the the interchange values are taken in the reverse order (last to first) or false if not. This is equivalent to the `isinv` parameter in the constructor from the `pp` values.

## 11.3 Functions

Most of these functions aren't very interesting, since most of them have trivial values like 1 or n. But we provide them all for consistency with the functions that other matrices provide.

```
int p.norm1() = Norm1(p) = 1
int p.norm2() = Norm2(p) = 1
int p.normInf() = NormInf(p) = 1
int p.maxAbsElement() = MaxAbsElement(p) = 1
int p.maxAbs2Element() = MaxAbs2Element(p) = 1
double p.normF() = NormF(p) = p.norm() = Norm(p) = sqrt(n)
int p.normSq() = NormSq(p) = n
double p.normSq(double scale) = n*scale^2
int p.trace() = Trace(p)
int p.sumElements() = SumElements(p) = n
int p.sumAbsElements() = SumAbsElements(p) = n
int p.sumAbs2Elements() = SumAbs2Elements(p) = n
int p.det() = Det(p)
int p.logDet(int* sign=0) = LogDet(p)
bool p.isSingular() = false
int p.condition() = 1
int p.doCondition() = 1
pinv = p.inverse() = Inverse(p)
p.makeInverse(Matrix<T>& minv)
p.makeInverseATA(Matrix<T>& cov)

p.setToIdentity()
p.transposeSelf()
p.invertSelf()
Swap(p1,p2)
```

## 11.4 Arithmetic

```
v2 = p * v1
v2 = v1 * p
v2 = v1 / p
v2 = v1 % p
```

```
v *= p
v /= p
v %= p
m2 = p * m1
m2 = m1 * p
m2 = m1 / p
m2 = m1 % p
m *= p
m /= p
m %= p
p1 == p2
p1 != p2
```

## 11.5 I/O

The simplest output syntax is the usual:

```
os << p;
```

The output format is the same as for a `Matrix`, including all the 0's. (See 5.9.) Unlike most matrix types, this format cannot be read back into a `Permutation` object.

If you need to be able to read the values back in, you should use the compact I/O style that writes out the permutation as a list of indices for the interchanges.

```
os << tmv::CompactIO() << p;
is >> tmv::CompactIO() >> p;
```

On input, the `Permutation p` will be resized if necessary based on the size information read in.

See §15 for more information about specifying custom I/O styles, including features like using brackets instead of parentheses, or putting commas between elements, or specifying an output precision.

# 12 Errors and exceptions

There are two kinds of errors that the TMV library looks for. The first are coding errors. Some examples are:

- Trying to access elements outside the range of a `Vector` or `Matrix`.
- Trying to add to `Vector`s or `Matrix`es that are different sizes.
- Trying to multiply a `Matrix` by a `Vector` where the number of columns in the `Matrix` doesn't match the size of the `Vector`.
- Viewing a `Matrix` as a `HermMatrix` when the diagonal isn't real.

I check for all of these (and similar) errors using assert statements. If these asserts fail, it should mean that the programmer made a mistake in the code. (Unless I've made a mistake in the TMV code, of course.)

Once the code is working, you can make the code slightly faster by compiling with either `-DNDEBUG` or `-DTMV_NDEBUG`. I say slightly, since most of these checks are pretty innocuous. And most of the computing time is usually in the depths of the various algorithms, not in these $O(1)$ time checks of the dimensions and such. There are a few checks which take $O(N)$ time, such as the last item listed above, but these are not enabled by default. They are only turned on when using `-DTMV_EXTRA_DEBUG`.

The other kind of error checked for by the code is where the data don't behave in the way the programmer expected. Here is a (complete) list of these errors:

- A singular matrix is encountered in a division routine that cannot handle it.
- An input file has the wrong format.
- A Cholesky decomposition is attempted for a hermitian matrix that isn't positive definite.
- A QR downdate failed because the resulting $A^\dagger A - X^\dagger X$ was found not to be positive definite.

These errors are always checked for even if `-DNDEBUG` or `-DTMV_NDEBUG` is used. That's because they are not problems in the code per se, but rather are problems with the data or files used by the code. So they could still happen even after the code has been thoroughly tested.

All errors in the TMV library are indicated by throwing an object of type `tmv::Error`. If you decide to catch it, you can determine what went wrong by printing it:

```
catch (tmv::Error& e) {
    std::cerr << e << std::endl;
}
```

If you catch the error by value rather than by reference, it will print out a single line description. If you catch it by reference (as above), it may print out more information about the problem.

Also, `tmv::Error` derives from `std::exception` and overrides the `what()` method, so any program that catches these will catch `tmv::Error` as well.

If you want to be more specific, there are a number of classes that derive from `Error`:

## 12.1 FailedAssert

The `tmv::FailedAssert` exception indicates that one of the assert statements failed. Since these are coding errors, if you catch this one, you'll probably just want to print out the error and abort the program so you can fix the bug. In addition to printing the text of the assert statement that failed, if you catch by reference it will also indicate the file and line number as normal assert macros do. Unfortunately, it gives the line number in the TMV code, rather than in your own code, but hopefully seeing which function in TMV found the problem will help you figure out which line in your own code was incorrect.

If you believe that the assert failed due to a bug in the TMV code rather than your own code, please post a bug report at `https://github.com/rmjarvis/tmv/issues`.

## 12.2 Singular

The `tmv::Singular` exception indicates that you tried to invert or divide by a matrix that is (numerically) singular. This may be useful to catch specifically, since you may want to do something different when you encounter a singular matrix. Note however that this only detects exactly singular matrices. If a matrix is numerically close to singular, but no actual zeros are found, then no error will be thrown. Your results will just be unreliable.

## 12.3 ReadError

The `tmv::ReadError` exception indicates that there was some problem reading in a matrix or vector from an `istream` input. If you catch this by reference and write it, it will give you a fairly specific description of what the problem was as well as writing the part of the matrix or vector that was read in successfully.

## 12.4 NonPosDef

The `tmv::NonPosDef` exception indicates that you tried to do some operation that requires a matrix to be positive definite, and it turned out not to be positive definite. The most common example would be performing a Cholesky decomposition on a hermitian matrix. I suspect that this is the most useful exception to catch specifically, as opposed to just via the `tmv::Error` base class.

For example, the fastest algorithm for determining whether a matrix is (at least numerically) positive definite is to try the Cholesky decomposition and catch this exception. To wit:

```
bool IsPosDef(const tmv::HermMatrix<T>& m)
{
    try {
        m.view().chd();
    } catch (tmv::NonPosDef) {
        return false;
    }
    return true;
}
```

Or you might want to use Cholesky for division when possible and Bunch-Kaufman otherwise:

```
try {
    m.divideUsing(tmv::CH);
    m.setDiv();
} catch (tmv::NonPosDef) {
    m.divideUsing(tmv::LU);
    m.setDiv();
}
x = b/m;
```

Note, however, that the speed difference between the two algorithms is only about 20% - 30% or so for typical matrices. So if a significant fraction of your matrices are not positive definite, you are probably better off always using the Bunch-Kaufman (`tmv::LU`) algorithm. Code like that given above would probably be most useful when all of your matrices should be positive definite in exact arithmetic, but you want to guard against one failing the Cholesky decomposition due to round-off errors.

It is also worth mentioning that the routine `QR_Downdate` described in §14.11 below will also throw the exception `NonPosDef` when it fails.

## 12.5 Warnings

There are also a few things that are not necessarily errors, but indicate that something unexpected happened that TMV was able to handle and deal with. The user might be interested in knowing about them. Here is a complete list of these situations:

- The divide and conquer SVD or Eigen algorithm had trouble converging on a solution during the conquer stage. Usually this only happens for matrices that happen to have eigen values that are extremely close together and also an extremely large dynamic range for the eigenvalues. (In fact, the algorithm succeeds for all my tests of extreme matrices, but carefully constructed matrices could probably still trigger the problem, so I've left in the warning.)
- A `bad_alloc` was caught in an algorithm that tried to allocate extra temporary memory, and a slower algorithm that doesn't allocate new memory was used instead.
- The LAPACK function `dstegr` (or `sstegr`) had an error, and the function `dstedc` (or `sstedc`) was called instead. The `dstegr` calculation was wasted, although I don't think it is generally knowable *a priori* that this might happen.
- A LAPACK function requested more workspace than was provided. This shouldn't happen anymore, since I switched to using the LAPACK workspace queries. But the code to check for this is still there.

The default way of handling these situations is to do nothing. After all, TMV was able to handle the situation and do something sensible. However, these are situations that the programmer may want to know about. So, if you want, you can have warnings written to an output stream of your choice using the function:

```
std::ostream* tmv::WriteWarningsTo(std::ostream* os)
```

The `os` could be `std::cout` or some log file that you look at later. Or it could even be a `stringstream` so you can examine the text of the warnings within your code and take appropriate action.

The function `WriteWarningsTo` returns a pointer to the old warning stream in case you only want to change the warning output stream temporarily. Also if you give `WriteWarningsTo` a null pointer, rather than an actual `ostream`, then this will turn off the warnings.

There is also a quick shorthand for turning off warnings.[14]

```
void tmv::NoWarnings();
```

This is functionally equivalent to `tmv::WriteWarningsTo(0);`

[14] This is mostly for backwards compatibility. In the past, the default behavior had been to output warnings to `std::cout`, so it was important to have an easy way to turn them off.

## 13 Eigenvalues and eigenvectors

The eigenvalues of a matrix are important quantities in many matrix applications. A number, $\lambda$, is an eigenvalue of a square matrix, $A$, if for some non-zero vector $v$,

$$Av = \lambda v$$

in which case $v$ is the called an eigenvector corresponding to the eigenvalue $\lambda$. Since any arbitrary multiple of $v$ also satisfies this equation, it is common practice to scale the eigenvectors so that $||v||_2 = 1$. If $v_1$ and $v_2$ are eigenvectors whose eigenvalues are $\lambda_1 \neq \lambda_2$, then $v_1$ and $v_2$ are linearly independent.

The above equation implies that

$$\begin{aligned} Av - \lambda v &= 0 \\ (A - \lambda I)v &= 0 \\ \det(A - \lambda I) &= 0 \quad (\text{or } \ v = 0) \end{aligned}$$

If $A$ is an $N \times N$ matrix, then the last expression is called the characteristic equation of $A$ and the left hand side is a polynomial of degree $N$. Thus, it has potentially $N$ solutions. Note that, even for real matrices, the solution may yield complex eigenvalues, in which case, the corresponding eigenvectors will also be complex.

If there are solutions to the characteristic equation which are multiple roots, then these eigenvalues are said to have a multiplicity greater than 1. These eigenvalues may have multiple corresponding eigenvectors. That is, different values of $v$ (which are not just a multiple of each other) may satisfy the equation $Av = \lambda v$.

The number of independent eigenvectors corresponding to an eigenvalue with multiplicity $> 1$ may be less than that multiplicity[15]. Such eigenvalues are called "defective", and any matrix with defective eigenvalues is likewise called defective.

If 0 is an eigenvalue, then the matrix $A$ is singular. And conversely, singular matrices necessarily have 0 as one of their eigenvalues.

If we define $\Lambda$ to be a diagonal matrix with the values of $\lambda$ along the diagonal, then we have (for non-defective matrices)

$$AV = V\Lambda$$

where the columns of $V$ are the eigenvectors. If $A$ is defective, we can construct a $V$ that satisfies this equation too, but some of the columns will have to be all zeros. There will be one such column for each missing eigenvector, and the other columns will be the eigenvectors.

If $A$ is not defective, then all of the columns of $V$ are linearly independent, which implies that $V$ is not singular (i.e. $V$ is "invertible"). Then,

$$\begin{aligned} A &= V\Lambda V^{-1} \\ \Lambda &= V^{-1}AV \end{aligned}$$

This is known as "diagonalizing" the matrix $A$. The determinant and trace are preserved by this procedure, which implies two more properties of eigenvalues:

$$\begin{aligned} \det(A) &= \prod_{k=1}^{N} \lambda_k \\ \text{tr}(A) &= \sum_{k=1}^{N} \lambda_k \end{aligned}$$

[15] The multiplicity of the eigenvalue is often referred to as its algebraic multiplicity. The number of corresponding eigenvectors is referred to as its geometric multiplicity. So $1 \leq$ geometric multiplicity $\leq$ algebraic multiplicity.

If $A$ is a “normal” matrix – which means that $A$ commutes with its adjoint, $AA^\dagger = A^\dagger A$ – then the matrix $V$ is unitary, and $A$ cannot be defective. The most common example of a normal matrix is a hermitian matrix (where $A^\dagger = A$), which has the additional property that all of the eigenvalues are real[16].

So far, the TMV library can only find the eigenvalues and eigenvectors of hermitian matrices. The routines to do so are

```
void Eigen(const HermMatrix<T>& A, Matrix<T>& V, Vector<RT>& lambda)
void Eigen(const HermBandMatrix<T>& A, Matrix<T>& V, Vector<RT>& lambda)
```

On output, `V.col(i)` is the eigenvector corresponding to each eigenvalue `lambda(i)`, and `A*V` will be equal to `V*DiagMatrixViewOf(lambda)`.

There are also routines which only find the eigenvalues, which are faster, since they do not perform the calculations to determine the eigenvectors:

```
void Eigen(const HermMatrix<T>& A, Vector<RT>& lambda)
void Eigen(const HermBandMatrix<T>& A, Vector<RT>& lambda)
```

The eigenvalues `lambda` will be in ascending order as is usual for eigenvalue applications. Note that this is different from the order returned by `SV_Decompose` (cf. §14). Other than this detail and the fact that `lambda` is packaged as a `Vector` rather than a `DiagMatrix`, the result of these two functions for Hermitian matrices is identical.

[16] Other examples of normal matrices are unitary matrices ($A^\dagger A = AA^\dagger = I$) and skew-hermitian matrices ($A^\dagger = -A$). However, normal matrices do not have to be one of these special types.

# 14 Matrix decompositions

While many matrix decompositions are primarily useful for performing matrix division (or least-squares pseudo-division), one sometimes wants to perform the decompositions for their own sake. It is possible to get at the underlying decomposition with the various divider accessor routines like `m.lud()`, `m.qrd()`, etc. However, this is somewhat roundabout, and at times inefficient. So we provide direct ways to perform all of the various matrix decompositions that are implemented by the TMV code.

In some cases, the input matrix, which we call `A` below, can also serve as at least part of the output as well. For example, the LU decomposition can be done in place, so that on output, $L$ and $U$ are the lower and upper portions of `A`. Whenever this is the case, `A` will be listed as a non-`const` reference, and there will be a line after the function such as `L = A.lowerTri()` to indicate the portion or portions of the decomposition that are output in `A`. In other cases, the input matrix is just used as workspace, and it is junk on output, in which case, there is no such line following the function. If the input matrix is listed as `const`, then it won't be changed on output.

Also, all the matrix parameters that you pass to these functions may be views as well. If the matrix is listed as a `const` reference then you may use the corresponding `Const` view type. If it is listed as a non-`const` reference, the you may use the corresponding non-`Const` view type.

Sometimes, only certain parts of a decomposition are wanted. For example, you might want to know the singular values of a matrix, but not care about the $U$ and $V$ matrices. For cases such as this, there are versions of the decomposition routines which omit certain output parameters. These routines are generally faster than the versions which include all output parameters, since they can omit some of the calculations.

None of the decompositions are valid for `T = int` or `complex<int>`.

## 14.1 LU decomposition (`Matrix`, `BandMatrix`)

$A \rightarrow PLU$ where $L$ is lower triangular, $U$ is upper triangular, and $P$ is a permutation.

```
void LU_Decompose(Matrix<T>& A, Permutation& P);
L = A.unitLowerTri();
U = A.upperTri();

void LU_Decompose(
      const BandMatrix<T>& A, LowerTriMatrix<T>& L, BandMatrix<T>& U,
      Permutation& P);
```

In the second case, `U` must have `U.nhi() = A.nlo()+A.nhi()`, and `L` should be `UnitDiag`.

## 14.2 Cholesky decomposition (`HermMatrix`, `HermBandMatrix`)

$A \rightarrow LL^\dagger$, where $L$ is lower triangular, and $A$ is hermitian.

```
void CH_Decompose(HermMatrix<T>& A);
L = A.lowerTri();

void CH_Decompose(HermBandMatrix<T>& A);
L = A.lowerBand();
```

If `T` is real, then `A` may also be a `SymMatrix` or `SymBandMatrix` respectively, since these are also hermitian.

If $A$ is found to be not positive definite, a `NonPosDef` exception is thrown.

## 14.3 Bunch-Kaufman decomposition (`HermMatrix`, `SymMatrix`)

$A \rightarrow PLDL^\dagger P^T$ if $A$ is hermitian, and $A \rightarrow PLDL^T P^T$ if $A$ is symmetric, where $P$ is a permutation, $L$ is lower triangular, and $D$ is hermitian or symmetric tridiagonal (respectively). In fact, $D$ is even more special than that: it is block diagonal with $1 \times 1$ and $2 \times 2$ blocks, which means that there are no two consecutive non-zero elements along the off-diagonal.

```
void LDL_Decompose(SymMatrix<T>& A, SymBandMatrix<T>& D, Permutation& P);
L = A.unitLowerTri();
```

Note: If you are using LAPACK, rather than the native TMV code, then the `LDL_Decompose` routine throws a `tmv::Singular` exception if the matrix is found to be exactly singular. The LAPACK documentation says that the decomposition is supposed to finish successfully, but I have not found that to always be true. So if LAPACK reports that it has found a singular matrix, TMV will throw an exception. The native code will always successfully decompose the matrix.

## 14.4 Tridiagonal LDL$^\dagger$ decomposition (`HermBandMatrix`, `SymBandMatrix` with `nlo=1`)

$A \rightarrow LDL^\dagger$ or $A \rightarrow LDL^T$. where this time $D$ is a regular diagonal matrix and $L$ is a lower band matrix with a single subdiagonal and all 1's on the diagonal.

It turns out that the Bunch-Kaufman algorithm on banded matrices tends to expand the band structure without limit because of the pivoting involved, so it is not practical. However, with tridiagonal matrices, it is often possible to perform the decomposition without pivoting. There is then no growth of the band structure, but it is not as stable for singular or nearly singular matrices. If an exact zero is found on the diagonal along the way `tmv::NonPosDef` is thrown.[17]

```
void LDL_Decompose(HermBandMatrix<T>& A);
void LDL_Decompose(SymBandMatrix<T>& A);
L = A.lowerBand();
L.diag().setAllTo(T(1));
D = DiagMatrixViewOf(A.diag());
```

## 14.5 QR decomposition (`Matrix`, `BandMatrix`)

$A \rightarrow QR$, where $Q$ is column-unitary (i.e. $Q^\dagger Q = I$), $R$ is upper triangular, and $A$ is either square or has more rows than columns.

```
void QR_Decompose(Matrix<T>& A, UpperTriMatrix<T>& R);
Q = A;

void QR_Decompose(const BandMatrix<T>& A, Matrix<T>& Q, BandMatrix<T>& R);
```

In the second case, `R` must have `R.nhi()` $>=$ `A.nlo()+A.nhi()`.

If you only need $R$, the following versions are faster, since they do not fully calculate $Q$.

```
void QR_Decompose(Matrix<T>& A);
R = A.upperTri();

void QR_Decompose(const BandMatrix<T>& A, BandMatrix<T>& R);
```

[17] Note, however, that if $A$ is complex, symmetric - i.e. not hermitian - then this doesn't actually mean that $A$ is not positive definite (since such a quality is only defined for hermitian matrices). Furthermore, hermitian matrices that are not positive definite will probably be decomposed successfully without throwing, resulting in D having negative values.

Also, the LAPACK implementation throws an exception for matrices that the native code successfully decomposes. It throws for hermitian matrices whenever they are not positive definite, whereas the native code succeeds for many indefinite matrices.

## 14.6 QRP decomposition (`Matrix`)

$A \to QRP$, where $Q$ is column-unitary (i.e. $Q^\dagger Q = I$), $R$ is upper triangular, $P$ is a permutation, and $A$ is either square or has more rows than columns.

```
void QRP_Decompose(
      Matrix<T>& A, UpperTriMatrix<T>& R, Permutation& P,
      bool strict=false);
Q = A;
```

As discussed in §5.8.3, there are two slightly different algorithms for doing a QRP decomposition. If `strict` is `true`, then the diagonal elements of $R$ be strictly decreasing (in absolute value) from upper-left to lower-right[18].

If `strict` is `false` (or omitted) however, then the diagonal elements of $R$ will not be strictly decreasing. Rather, there will be no diagonal element of $R$ below and to the right of one which is more than a factor of $\epsilon^{1/4}$ smaller in absolute value, where $\epsilon$ is the machine precision. This restriction is almost always sufficient to make the decomposition useful for singular or nearly singular matrices, and it is much faster than the strict algorithm for most matrices.

If you only need $R$, the following versions is faster, since it does not fully calculate $Q$.

```
void QRP_Decompose(Matrix<T>& A, bool strict=false);
R = A.upperTri();
```

## 14.7 Singular value decomposition (`Matrix`, `BandMatrix`, `SymMatrix`, `HermMatrix`, `SymBandMatrix`, `HermBandMatrix`)

$A \to USV^\dagger$, where $U$ is column-unitary (i.e. $U^\dagger U = I$), $S$ is real diagonal, $V$ is square unitary, and $A$ is either square or has more rows than columns. Note that the functions below do not deal with $V$ directly, they instead return $V^\dagger$ in the parameter `Vt`. So on output, `U*S*Vt` will equal the original matrix `A`. I find this to usually be more convenient, but if you prefer dealing with $V$ directly, you can pass `V.adjoint()` as the parameter to these functions.

```
void SV_Decompose(Matrix<T>& A, DiagMatrix<RT>& S, Matrix<T>& Vt);
U = A;
V = Vt.adjoint();

void SV_Decompose(
      const SymMatrix<T>& A, Matrix<T>& U, DiagMatrix<RT>& S,
      Matrix<T>& Vt);
V = Vt.adjoint();

void SV_Decompose(
      const HermMatrix<T>& A, Matrix<T>& U, DiagMatrix<RT>& S,
      Matrix<T>& Vt);
V = Vt.adjoint();

void SV_Decompose(
      const BandMatrix<T>& A, Matrix<T>& U, DiagMatrix<RT>& S,
      Matrix<T>& Vt);
V = Vt.adjoint();
```

[18] If you are using a LAPACK library, you might find that the output $R$ diagonal is not always strictly decreasing, although it will usually be close. If strictly monotonic diagonal elements are important for you, you can use the native TMV algorithm instead by compiling with the SCons option `USE_GEQP3=false`. (`geqp3` is the name of the LAPACK function that does the QRP decomposition.)

```
void SV_Decompose(
      const SymBandMatrix<T>& A, Matrix<T>& U, DiagMatrix<RT>& S,
      Matrix<T>& Vt);
V = Vt.adjoint();

void SV_Decompose(
      const HermBandMatrix<T>& A, Matrix<T>& U, DiagMatrix<RT>& S,
      Matrix<T>& Vt);
V = Vt.adjoint();
```

The input $A$ matrix must not have more columns than rows. If you want to calculate the SVD of such a matrix, you should decompose $A^T$ instead:

```
tmv::Matrix<double> A(nrows,ncols); // ncols > nrows
// A = ...
tmv::Matrix<double> Vt = A;
tmv::DiagMatrix<double> S(nrows);
tmv::Matrix<double> U(nrows,nrows);
SV_Decompose(Vt.transpose(),S.view(),U.transpose());
// Now A = U * S * Vt
```

If you only need $S$, or $S$ and $V$, or $S$ and $U$, the following versions are faster, since they do not fully calculate the omitted matrices.

```
void SV_Decompose(
      Matrix<T>& A, DiagMatrix<RT>& S, Matrix<T>& Vt, bool StoreU);
// With StoreU=false, then U != A

void SV_Decompose(Matrix<T>& A, DiagMatrix<RT>& S, bool StoreU);
if (StoreU) U = A;

void SV_Decompose(SymMatrix<T>& A, DiagMatrix<RT>& S);

void SV_Decompose(HermMatrix<T>& A, DiagMatrix<RT>& S);

void SV_Decompose(const SymMatrix<T>& A, DiagMatrix<RT>& S, Matrix<T>& Vt);

void SV_Decompose(
      const HermMatrix<T>& A, DiagMatrix<RT>& S, Matrix<T>& Vt);

void SV_Decompose(const SymMatrix<T>& A, Matrix<T>& U, DiagMatrix<RT>& S);

void SV_Decompose(
      const HermMatrix<T>& A, Matrix<T>& U, DiagMatrix<RT>& S);

void SV_Decompose(const BandMatrix<T>& A, DiagMatrix<RT>& S);

void SV_Decompose(
      const BandMatrix<T>& A, DiagMatrix<RT>& S, Matrix<T>& Vt);

void SV_Decompose(
      const BandMatrix<T>& A, Matrix<T>& U, DiagMatrix<RT>& S);

void SV_Decompose(const SymBandMatrix<T>& A, DiagMatrix<RT>& S);

void SV_Decompose(const HermBandMatrix<T>& A, DiagMatrix<RT>& S);

void SV_Decompose(
      const SymBandMatrix<T>& A, DiagMatrix<RT>& S, Matrix<T>& Vt);

void SV_Decompose(
      const HermBandMatrix<T>& A, DiagMatrix<RT>& S, Matrix<T>& Vt);

void SV_Decompose(
      const SymBandMatrix<T>& A, Matrix<T>& U, DiagMatrix<RT>& S);

void SV_Decompose(
      const HermBandMatrix<T>& A, Matrix<T>& U, DiagMatrix<RT>& S);
```

On output, the singular values `S` will be in decreasing order of absolute value. This is the typical order one would want for principal value decompositions, where only the largest several singular values (aka principal values) would be used. Note that this is different from the order returned by `Eigen` (cf. §13). Other than this detail and the fact that `S` is packaged as a `DiagMatrix` rather than a `Vector`, the result of these two functions for Hermitian matrices is identical.

## 14.8 Polar decomposition (`Matrix`, `BandMatrix`)

$A \rightarrow UP$ where $U$ is unitary and $P$ is positive definite hermitian.

This is similar to polar form of a complex number: $z = re^{i\theta}$. In the matrix version, $P$ acts as $r$, being in some sense the "magnitude" of the matrix. And $U$ acts as $e^{i\theta}$, being a generalized rotation.

```
void Polar_Decompose(Matrix<T>& A, SymMatrix<T>& P);
U = A;

void Polar_Decompose(
      const BandMatrix<T>& A, Matrix<T>& U, SymMatrix<T>& P);
```

If `T` is real, then `P` may also be a `SymMatrix`, since it is also hermitian.

## 14.9 Matrix square root (`HermMatrix`, `HermBandMatrix`)

$A \rightarrow SS$, where $A$ and $S$ are each positive definite hermitian matrices.

```
void SquareRoot(HermMatrix<T>& A);
S = A;

void SquareRoot(const HermBandMatrix<T>& A, SymMatrix<T>& S);
```

If `T` is real, then `A` and `S` may also be a `SymMatrix` or `SymBandMatrix` as appropriate, since these are also hermitian.

If $A$ is found to be not positive definite, a `NonPosDef` exception is thrown.

## 14.10 Update a QR decomposition

One reason that it can be useful to create and deal with the QR decomposition directly, rather than just relying on the division routines is the possibility of updating or "downdating" the resulting $R$ matrix.

If you are doing a least-square fit to a large number of linear equations, you can write the system as a matrix equation: $Ax = b$, where $A$ is a matrix with more rows than columns, and you are seeking, not an exact solution for $x$, but rather the value of $x$ which minimizes $||b - Ax||_2$. See §5.8.2 for a more in-depth discussion of this topic.

It may be the case that you have more rows (i.e. constraints) than would allow the entire matrix to fit in memory. In this case it may be tempting to use the so-called normal equation instead:

$$\begin{aligned} A^\dagger A x &= A^\dagger b \\ x &= (A^\dagger A)^{-1} A^\dagger b \end{aligned}$$

This equation theoretically gives the same solution as using the QR decomposition on the original design matrix. However, it can be shown that the condition of $A^\dagger A$ is the <u>square</u> of the condition of $A$. Since larger condition values lead to larger numerical instabilities and round-off problems, a mildly ill-conditioned matrix is made much worse by this procedure.

When all of $A$ fits in memory, the better solution is to use the QR decomposition, $A = QR$, to calculate $x$.

$$\begin{aligned} QRx &= b \\ x &= R^{-1} Q^\dagger b \end{aligned}$$

In fact, this is the usual behind-the-scenes procedure when you write `x = b/A` in TMV. But if $A$ is too large to fit in memory, then so is $Q$.

A compromise solution, which is not quite as good as doing the full QR decomposition, but is better than using the normal equation, is to just calculate the $R$ of the QR decomposition, and not $Q$. Then:

$$\begin{aligned} A^\dagger A x &= A^\dagger b \\ R^\dagger Q^\dagger Q R x = R^\dagger R x &= A^\dagger b \\ x &= R^{-1} (R^\dagger)^{-1} A^\dagger b \end{aligned}$$

Calculating $R$ directly from $A$ is numerically much more stable than calculating it through, say, a Cholesky decomposition of $A^\dagger A$. So this method produces a more accurate answer for $x$ than the normal equation does.

But how can $R$ be calculated if we cannot fit all of $A$ into memory at once?

First, we point out a characteristic of unitary matrices that the product of two or more of them is also unitary. This implies that if we can calculate something like: $A = Q_0 Q_1 Q_2 ... Q_n R$, then this is the $R$ that we want.

So, consider breaking $A$ into a submatrix, $A_0$, which can fit into memory, plus the remainder, $A_1$, which may or may not.

$$A = \begin{pmatrix} A_0 \\ A_1 \end{pmatrix}$$

First perform a QR decomposition of $A_0 = Q_0 R_0$. Then we have:

$$\begin{aligned} A &= \begin{pmatrix} Q_0 R_0 \\ A_1 \end{pmatrix} \\ &= \begin{pmatrix} Q_0 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} R_0 \\ A_1 \end{pmatrix} \\ &\equiv Q_0' A_1' \end{aligned}$$

Assuming that $A_0$ has more rows than columns, then $A_1'$ has fewer rows than the original matrix $A$. So we can iterate this process until the resulting matrix can fit in memory, and we can perform the final QR update to get the final value of $R$.

For the numerical reasons mentioned above, the fewer such iterations you do, the better. So you should try to include as many rows of the matrix $A$ as possible in each step, given the amount of memory available.

The solution equation, written above, also needs the quantity $A^\dagger b$, which can be accumulated in the same blocks:

$$A^\dagger b = A_0^\dagger b_0 + A_1^\dagger b_1 + ....$$

This, combined with the calculation of $R$, allows us to determine $x$ using the above formula.

The TMV library includes a command which does the update step of the above procedure directly, which is slightly more efficient than explicitly forming the $A'_k$ matrices. The commands is

```
void QR_Update(UpperTriMatrix<T>& R, Matrix<T>& X)
```

which updates the value of $R$ such that $R_{\text{out}}^\dagger R_{\text{out}} = R_{\text{in}}^\dagger R_{\text{in}} + X^\dagger X$. (The input matrix X is destroyed in the process.) This is equivalent to the QR definition of the update described above.

So the entire process might be coded using TMV as:

```
int n_full = nrows_for_full_A_matrix;
int n_mem = nrows_that_fit_in_memory;
assert(n_mem <= n_full);
assert(n_mem > ncols);

tmv::Matrix<double> A(n_mem,ncols);
tmv::Vector<double> b(n_mem);

// Import_Ab sets A to the first n_mem rows of the full matrix,
// and also sets b to the same components of the full rhs vector.
// Maybe it reads from a file, or performs a calculation, etc.
Import_Ab(0,n_mem,A,b);

// x will be the solution to A_full x = b_full when we are done
// But for now, it is accumulating A_full.transpose() * b_full.
tmv::Vector<double> x = A.transpose() * b;

// Do the initial QR decomposition:
QR_Decompose(A.view());
tmv::UpperTriMatrix<double> R = A.upperTri();

// Iterate until we have done all the rows
for(int n1=n_mem, n2=n1+n_mem; n2<n_full; n1=n2, n2+=n_mem) {
    if (n2 > n_full) n2 = n_full;

    // Import the next bit:
    Import_Ab(n1,n2,A,b);

    // (Usually, A1==A, b1==b, but not the last time through the loop.)
    tmv::MatrixView<double> A1 = A.rowRange(0,n2-n1);
    tmv::VectorView<double> b1 = b.subVector(0,n2-n1);

    // Update, x, R:
    x += A1.transpose() * b1;
    QR_Update(R,A1);
}
```

```
// Finish the solution:
x /= R.transpose();
x /= R;
```

If the update only needs to add a single row, you can also use the form

```
void QR_Update(UpperTriMatrix<T>& R, Vector<T>& v)
```

where `v` is the single row that you want to have added to the matrix. Again, `v` is destroyed in the process.

## 14.11 Downdate a QR decomposition

When performing a least-square fit of some data to a model, it is common to do some kind of outlier rejection to remove data that seem not to be applicable to the model - things like spurious measurements and such. For this, we basically want the opposite of a QR update - instead we want to find the QR decomposition that results from removing a few rows from $A$. This is called a QR "downdate", and is performed using the subroutine:

```
void QR_Downdate(UpperTriMatrix<T>& R, Matrix<T>& X)
```

where `X` represents the rows from the original matrix to remove from the QR decomposition. (The input matrix `X` is destroyed in the process.) If you only want to remove a single row, you can also use

```
void QR_Downdate(UpperTriMatrix<T>& R, Vector<T>& v)
```

It is possible for the downdate to fail (and throw an exception) if the matrix $X$ does not represent rows of the matrix that was originally used to create $R$. Furthermore, with round-off errors, the error may still result with actual rows from the original $A$ if $R$ gets too close to singular. In this case, `QR_Downdate` throws a `NonPosDef` exception. This might seem like a strange choice, but the logic is that $R^\dagger R$ is the Cholesky decomposition of $A^\dagger A$, and `QR_Downdate(R,X)` basically updates $R$ to be the Cholesky decomposition of $A^\dagger A - X^\dagger X$. The procedure fails (and throws) when this latter matrix is found not to be positive definite.

It is worth pointing out that the algorithm used in TMV is a new one developed by Mike Jarvis. Most of the texts and online resources that discuss the QR downdate algorithm only explain how to do one row at a time, using a modification of the QR update using Givens rotations. If you are doing many rows, it is common that roundoff errors in such a procedure accumulate sufficiently for the routine to fail. The TMV algorithm instead downdates all of the rows together using a modification of the Householder reflection algorithm for updates. This algorithm seems to be much more stable than ones that use Givens rotations.

The only references to a similar algorithm that I could find in the literature is described in the paper, "Stability Analysis of a Householder-based Algorithm for Downdating the Cholesky Factorization", Bojanczyk and Steinhardt, 1991, Siam J. Sci. Stat. Comput. 12, 6, 1255[19]. This paper describes a similar algorithm to compute the downdated $R$ matrix using Householder matrices. However, the details of the computation are somewhat different from the TMV algorithm. Also, they only consider real matrices, and they do not include the block-householder techniques in their description to employ more so-called "level-3" matrix operations.

Therefore, I will describe the TMV downdate algorithm here. I think it is clearer to begin by describing the update algorithm in §14.11.1, since it is quite similar to the algorithm we use for downdating, but is a bit easier to understand. Then the downdate algorithm is described in §14.11.2.

### 14.11.1 The update algorithm

First lets look at the Householder algorithm for QR update:

[19] It seems that this paper has become a bit forgotten. A recent paper, "Efficient Algorithms for Block Downdating of Least Squares Solutions", Yanev and Kontoghiorghes, 2004, Applied Numerical Mathematics, 49, 3, evaluates five algorithms for doing the downdate. However, all of them are block versions of the Given matrix approach. They do not consider any algorithms that use Householder matrices to do the downdate, and do not reference the above paper by Bojanczyk and Steinhardt. In addition, none of the papers that do cite the Bojanczyk and Steinhardt paper seem to be about the general problem of QR (or Cholesky) downdating.

Given the initial decomposition $A_0 = Q_0 R_0$, we want to find $R$ such that

$$\begin{aligned} A_1 = \begin{pmatrix} A_0 \\ X \end{pmatrix} &= Q_1 R_1 \\ \begin{pmatrix} Q_0 R_0 \\ X \end{pmatrix} &= Q_1 R_1 \\ \begin{pmatrix} Q_0 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} R_0 \\ X \end{pmatrix} &= Q_1 R_1 \end{aligned}$$

So if we perform a QR decomposition:

$$S \equiv \begin{pmatrix} R_0 \\ X \end{pmatrix} = Q_S R_S$$

Then this is the $R$ we want: $R_1 = R_S$, and

$$Q_1 = \begin{pmatrix} Q_0 & 0 \\ 0 & 1 \end{pmatrix} Q_S$$

For the following discussion, let $N$ be the number of rows (and columns) in $R_0$, and $M$ be the number of rows in $X$.

To perform the decomposition, we multiply $S$ by a series of Householder reflections on the left to zero out each column of $X$ one at a time. Householder reflections are unitary, so their product in the reverse order is $Q_S$:

$$\begin{aligned} \begin{pmatrix} R_1 \\ 0 \end{pmatrix} &= H_N H_{N-1} ... H_2 H_1 \begin{pmatrix} R_0 \\ A_1 \end{pmatrix} \\ Q_S &= H_1^\dagger H_2^\dagger ... H_{N-1}^\dagger H_N^\dagger \end{aligned}$$

Householder reflections are defined as $H = I - \beta(x - ye_1)(x - ye_1)^\dagger$ where $x$ is a (column) vector, $y$ is a scalar with $|y| = ||x||_2$, $e_1$ is the basis vector whose only non-zero element is the first: $e_1(1) = 1$, and $\beta = (||x||_2^2 - y^* x(1))^{-1}$. They have the useful properties that $Hx = ye_1$ and they are unitary: $H^\dagger H = HH^\dagger = I$. Furthermore, if $\beta$ is real, they are also hermitian: $H = H^\dagger$.

$H_1$ is defined for the vector $x = (R_0(1,1), 0, 0, ..., 0, 0, X(1,1), X(2,1), ..., X(M,1))$ where the stretch of 0's includes a total of $(N-1)$ 0's. This value of $x$ completely determines the Householder matrix $H_1$ up to an arbitrary sign on either $y$ or $\beta$ (or in general an arbitrary factor $e^{i\theta}$) which is chosen to minimize rounding errors. The optimal choice is to choose $y = -||x||_2\, x(1)/|x(1)|$, which makes $\beta$ real. However, the LAPACK choice is $y = -||x||_2\, sign(real(x(1)))$, which means $\beta$ is complex, and $H$ is not Hermitian[20].

The product $H_1 S$ "reflects" the first column of $X$ into the first diagonal element of $R_0$. Because of all the 0's, most of $R_0$ is unaffected – only the first row of $R_0$ and the rest of $X$ are changed. The subsequent Householder reflections are defined similarly, each zeroing out a column of $X$, and modifying the corresponding row of $R_0$ and the remaining elements of $X$.

At the end of this procedure, the matrix $R_0$ will be changed into the matrix $R_1$. If desired, $Q_S$ (and then $Q_1$) may also be calculated in the process, but the TMV implementation of the QR update does not calculate $Q_1$. If there is a demand for such a routine, it would not be hard to add it, but I think most applications of the update do not use the $Q$ matrix explicitly.

[20] This choice complicates a lot of the calling routines which use Householder matrices, since you need to keep track of conjugation of the $\beta$ values. Since TMV is designed to be able to call LAPACK when possible, it is forced to follow the same convention.

In fact, it could be argued that the LAPACK convention is even "wrong" in the sense that their Householder matrices are not actually "reflections". A reflection is a unitary matrix whose determinant is $-1$. The determinant of a Householder matrix as defined here is $-\beta^2/|\beta|^2$ which is $-1$ for real $\beta$, but not for complex $\beta$. But we are stuck with their choice, so we allow $\beta$ to be complex in this discussion.

### 14.11.2 The downdate algorithm

Given the initial decomposition

$$A_1 = \left( \begin{array}{c} A_0 \\ X \end{array} \right) = Q_1 R_1$$

we want to find $R_0$ such that $A_0 = Q_0 R_0$.

The TMV algorithm to do this essentially performs the same steps as in the update algorithm above, but instead removes the effect of each $H$ from $R_1$. This is easy to do if we can determine what each $H$ is, since $H^{-1} = H^\dagger$, so we just apply $H^\dagger$ to update each row of $R_1$. The $X$ update takes the regular $H$ matrix, since we need to replicate the steps that we would do for an update to keep finding the correct values for the remaining columns of $X$.

All of the values in the vector $x$ needed to define $H_1$ are given, except for the first, $R_0(0,0)$. But this is easy to calculate, since

$$|R_1(0,0)|^2 = |R_0(0,0)|^2 + ||X(1:M,0)||_2^2$$

This determines the $x$ vector, which in turn defines $H_1$ (modulo an arbitrary sign, which again is chosen to minimize rounding errors). Thus, we can calculate $H_1$ and apply it as described above. Each subsequent Householder matrix is created and applied similarly for each column of $X$. When we have finished this process, we are left with $R_0$ in the place of $R_1$.

If at any point in the process, we find the calculated $|R_0(k,k)|^2 < 0$, then the algorithm fails. In the TMV implementation, a `NonPosDef` exception is thrown.

In practice, for both of these algorithms, we actually use a blocked implementation for updating the $R$ and $X$ matrices. We accumulate the effect of the Householder matrices until there are sufficiently many (e.g. 64), at which point we update the appropriate rows of the $R$ matrix and the rest of $X$. Implementing this correctly is mostly a matter of keeping track of which elements have been updated yet, making sure that whenever an element is used, it is already updated, while delaying as much of the calculation as possible in order to make maximum use of the so-called "level-3" matrix functions, which are the most efficient on modern computers. We also make the additional improvement of using a recursive algorithm within each block, which gains some additional level-3 operations, for a bit more efficiency.

## 15 Matrix I/O Styles

### 15.1 The normal I/O format

As mentioned previously, the simplest syntax for writing a matrix to a stream is the usual C++ syntax:

```
os << m;
```

This always uses the same format no matter what kind of matrix is being printed. Namely, the size of the matrix (nrows ncols), and then each row listed as a vector surrounded by parentheses. So for a $4 \times 5$ matrix, the output would be something like:

```
4 5
( 1.2  3.4  0.9  5.3  6.1 )
( 0.9  3.9  8.8  9.0  1.0 )
( 2.4  1.7  5.6  7.6  8.3 )
( 0.1  4.5  6.6  1.9  6.2 )
```

This format can be read back in using the usual C++ syntax:

```
is >> m;
```

If the size of the matrix `m` doesn't match the size being read from the input stream, then the matrix will be resized to match, assuming the matrix is a type that can be resized appropriately. If the matrix type is intrinsically square (e.g. `DiagMatrix` or `SymMatrix`), then the input sizes must be equal. The above $4 \times 5$ matrix could not be read into these types for example.

And if the type of matrix `m` requires zeros in certain places, then the input matrix must have zeros in those places too. If it does not, then a `ReadError` is thrown. Likewise if the matrix type is symmetric or Hermitian, then the input matrix must be such as well.

### 15.2 The compact I/O format

The other standard I/O format that we have already mentioned is what I call the "Compact" format. This format puts everything on a single line and includes all the information to load the values back into memory, but no more than than. It starts with a letter code indicating the kind of matrix being written. Then any size values. Then all the elements in row-major order, omitting any trivial values based on the shape or type of matrix.

To write using this format, you would write:

```
os << tmv::CompactIO() << m;
```

The above $4 \times 5$ matrix would be output as:

```
M 4 5 1.2 3.4 0.9 5.3 6.1 0.9 3.9 8.8 9.0 1.0 2.4 1.7 5.6 7.6 8.3 0.1 4.5
6.6 1.9 6.2
```

all on one line (i.e. without the line break).

This format isn't as pretty of course, but it saves space, especially when writing some of the special matrix varieties that have lots of zeros in them. Also, this format records extra information that might be necessary when reading back. For example, the compact format for banded matrices includes the number of super- and sub-diagonals, so the matrix can be correctly resized on input.

Also, `Permutation` objects store their data in a very different way than what gets output in the normal format, so it can only be read in using the compact format.

### 15.3 The `IOStyle` class

The above function `tmv::CompactIO()` returns a `tmv::IOStyle` object. This is the class TMV uses to mediate everything about the I/O. And if you don't like either of the above two styles, you can design your own.

For example, you might like the normal format, but don't want the size printed. Or you might want to use square brackets rather than parentheses. Or no brackets at all, but still in a rectangular grid. All these things are possible with the `IOStyle` class. So let's get into some details about how you can use it.

The only constructor is a default constructor:

```
IOStyle()
```

which gives you the "normal" I/O format described above. (However, see §15.4 below for how you can change the default IOStyle.) In fact, TMV internally converts a statement like:

```
os << m;
```

into

```
os << IOStyle() << m;
```

The `IOStyle` class has quite a few methods that change various aspects of the I/O behavior:

- `useCode()`
  `noCode()`

  Specify whether or not to use the letter code that indicates what kind of object is being printed. The codes TMV currently uses are:

  - `V = Vector`
  - `M = Matrix`
  - `P = Permutation`
  - `D = DiagMatrix`
  - `U = UpperTriMatrix`
  - `L = LowerTriMatrix`
  - `B = BandMatrix`
  - `S = SymMatrix`
  - `H = HermMatrix`
  - `sB = SymBandMatrix`
  - `hB = HermBandMatrix`

- `simpleSize()`

  Write the size as `nrows ncols` regardless of whether this is redundant (because the matrix type is necessarily square, like a `DiagMatrix`) or insufficient (because the matrix type needs more information like the number of super- and sub-diagonals for a `BandMatrix`).

- `fullSize()`

  Write all the size information necessary for the matrix type in question, and only as much as is necessary. e.g. for a `DiagMatrix`, only write `nrows` (since `ncols` is the same), and for a `BandMatrix`, write `nrows ncols nlo nhi` to describe the full band structure.

- `noSize()`

  Do not write any size values.

- `noPrefix()`

  Turn off all prefix items (code and size)

- `markup(start, lparen, space, rparen, rowend, final)`

  Change the markup text to use. (All the parameters are input as `std::string`, although one would usually give tham as string literals.) `start` is the text to put after the prefix and before the first row of a matrix. `lparen` is the text to put before the first element of a row (or vector). `space` is the text to put between two elements in a row (or vector). `rparen` is the text to put after the last element of a row (or vector). `rowend` is the text to put after each row in a matrix, except for the last one. `final` is the text to put after the last row in a matrix.

  To give a concrete example, the "normal" format described above would be specified as

  ```
  markup("\n","( ","  "," )","\n","\n")
  ```

- `fullMatrix()`
  `compact()`

  Specify whether to write all the elements of a matrix, even if they are trivially zero because of the type of matrix being printed (`fullMatrix`), or to only write those elements that are non-trivial (`compact`).

- `setThresh(double thresh)`

  Specify a threshold, below which (in absolute value) any output value should be written as zero, instead of its value in memory. This is useful when looking at matrices that should be numerically mostly (or all) 0's, so you don't have to wade through many values like `6.4345e-16` looking for elements that are significant.

  Note that for complex values, the real and imaginary components are separately checked, so something like `(1,7.54351e-16)` would be written as `(1,0)` if `thresh` is something reasonable like `1.e-10`.

- `setPrecision(int prec)`
  `useDefaultPrecision()`

  Specify the precision to use on output. The first method basically applies the `std::ostream` method `precision(prec)` to the matrix elements being written. The second method instructs TMV to just use whatever the stream's current precision is without changing it.

All of these methods return the `IOStyle` object back (by reference), so they can be strung together and then passed to the `ostream` or `istream` for a particular matrix.

```
os << IOStyle().noPrefix().setPrecision(8) << m;
```

Note that each TMV object being read or written needs its own `IOStyle` specification preceding it if you want to use something other than the default. So if you write

```
os << IOStyle().noPrefix().setPrecision(8) << m1 << m2;
```

`m1` would get the modified format spectified, but `m2` would get the default style. If you want to use the same modified format for several matrices, you can create and name an `IOStyle` object and use it several times. For example,

```
IOStyle myStyle;
myStyle.noPrefix().setPrecision(8);
os << myStyle << m1 << myStyle << m2;
```

Or if you want to use an alternate style for every I/O statement in your program, you could change the default `IOStyle`...

## 15.4 Changing the default style

If you have a preference for a different I/O style, you can set it to be the default `IOStyle` that is used automatically when using the simple I/O commands like `os << m` or `is >> m`. To set a particular `IOStyle` object to be the default, use the method

```
makeDefault()
```

For example, if you want the standard I/O style to have no prefix and use square brackets rather than parentheses, you could write

```
IOStyle().noPrefix().markup("","[ ","  "," ]","\n","\n").makeDefault();
```

near the beginning of you program. Then every time your program writes a matrix (unless you explicitly specify some other style), it will use that format instead of the original default.

You can also revert the default I/O style back to its original state with the class function

```
tmv::IOStyle::revertDefault();
```

## 15.5 Functions that return an `IOStyle` object

We have several functions that return an `IOStyle` object that may be more convenient than starting with the default `IOStyle()` and specifying modifications:

- `CompactIO()` was already mentioned above. It is equivalent to:

  ```
  IOStyle().useCode().fullSize().compact().markup("",""," ",""," ","")
  ```

- `NormalIO()` is normally unnecessary, since it is the default `IOStyle`. However, it can be useful if the default has been changed. It is equivalent to

  ```
  IOStyle().noCode().simpleSize().fullMatrix().
         markup("\n","( ","  "," )","\n","\n")
  ```

- `ThreshIO(thresh)` is equivalent to

  ```
  IOStyle().setThresh(thresh)
  ```

- `PrecIO(prec)` is equivalent to

  ```
  IOStyle().setPrecision(prec)
  ```

- `EigenIO()` mimics the default I/O style of the Eigen matrix library. It is equivalent to

  ```
  IOStyle().noPrefix().fullMatrix().markup("",""," ","","\n","");
  ```

Notice that all of these functions start with the default `IOStyle`, so they inherit from the current default style anything that they don't explicitly override. For example, most of them do not explicitly do anything to the precision. So whatever is currently set in the default will also be used when you use these functions.

Also, you can start with one of these functions and modify from there, rather than starting with `IOStyle()`, if that is more convenient. For example, if you want to use the "compact format" and also set a threshold of `1.e-6`, you could write

```
os << tmv::CompactIO().setThresh(1.e-6) << m;
```

## 16 Example code

Here are five complete programs that use the TMV library. They are intended to showcase some of the features of the library, but they are certainly not comprehensive. Hopefully, they will be useful for someone who is just getting started with the TMV library as a kind of tutorial of the basics.

The code includes the output as comments that start with `//!`, so you can more easily see what is going on. And each file listed here is also included with the TMV distribution in the `examples` directory, so you can easily compile and run them yourself.

### 16.1 Vector

```
#define TMV_EXTRA_DEBUG // To get the 888's below.

#include "TMV.h"
#include <iostream>

int main() try
{
    // Several ways to create and initialize vectors:

    // Create with uninitialized values
    tmv::Vector<double> v1(6);
    // In debug mode, all are initialized to 888 to make it easier to
    // notice when you fail to initialize correctly.
    std::cout<<"v1 = "<<v1<<std::endl;
    //! v1 = 6  ( 888  888  888  888  888  888 )

    // Initialize with STL-compliant iterator
    srand(51572);
    std::generate(v1.begin(),v1.end(),rand);
    v1 /= double(RAND_MAX);
    std::cout<<"v1 = "<<v1<<std::endl;
    //! v1 = 6  ( 0.403622  0.66681  0.0776762  0.504604  0.871968  0.163157 )

    // Create with all 2's.
    tmv::Vector<double> v2(6,2.);
    std::cout<<"v2 = "<<v2<<std::endl;
    //! v2 = 6  ( 2  2  2  2  2  2 )

    // Create with given elements from C-array
    double vv[6] = {1.1,8.,-15.,2.5,6.3,-12.};
    tmv::Vector<double> v3(6);
    std::copy(vv,vv+6,v3.begin());
    std::cout<<"v3 = "<<v3<<std::endl;
    //! v3 = 6  ( 1.1  8  -15  2.5  6.3  -12 )

    // Initialize with direct access of each element
    tmv::Vector<double> v4(6);
    for(int i=0;i<v4.size();i++) {
        v4(i) = 2*i+10.;  // Could also use v4[i] instead of v4(i)
    }
    std::cout<<"v4 = "<<v4<<std::endl;
    //! v4 = 6  ( 10  12  14  16  18  20 )
```

```
// Initialize with comma-delimited list
v4 << 1.2, 9., 12, 2.5, -7.4, 14;
std::cout<<"v4 = "<<v4<<std::endl;
//! v4 = 6  ( 1.2  9  12  2.5  -7.4  14 )
// If the list is the wrong size, a run time error occurs:
try {
    v4 << 1.2, 9., 12, 2.5;
} catch (tmv::Error& e) {
    std::cout<<"Caught e = "<<e;
    //! Caught e = TMV Error: Reading from list initialization.
    //! Reached end of list, but expecting 2 more elements.
}
try {
    v4 << 1.2, 9., 12, 2.5, -7.4, 14, 99;
} catch (tmv::Error& e) {
    std::cout<<"Caught e = "<<e;
    //! Caught e = TMV Error: Reading from list initialization.
    //! List has more elements than expected.
}

// Norms, etc.
std::cout<<"Norm1(v3) = "<<Norm1(v3);
std::cout<<" = "<<v3.sumAbsElements()<<std::endl;
//! Norm1(v3) = 44.9 = 44.9

std::cout<<"Norm2(v3) = "<<Norm2(v3);
std::cout<<" = "<<v3.norm()<<std::endl;
//! Norm2(v3) = 21.9123 = 21.9123

std::cout<<"NormInf(v3) = "<<NormInf(v3);
std::cout<<" = "<<v3.maxAbsElement()<<std::endl;
//! NormInf(v3) = 15 = 15

std::cout<<"SumElements(v3) = "<<SumElements(v3)<<std::endl;
//! SumElements(v3) = -9.1

// Min/Max elements:
int i1,i2,i3,i4;
double x1 = v3.minAbsElement(&i1);
double x2 = v3.maxAbsElement(&i2);
double x3 = v3.minElement(&i3);
double x4 = v3.maxElement(&i4);
std::cout<<"|v3("<<i1<<")| = "<<x1<<" is the minimum absolute value\n";
//! |v3(0)| = 1.1 is the minimum absolute value
std::cout<<"|v3("<<i2<<")| = "<<x2<<" is the maximum absolute value\n";
//! |v3(2)| = 15 is the maximum absolute value
std::cout<<"v3("<<i3<<") = "<<x3<<" is the minimum value\n";
//! v3(2) = -15 is the minimum value
std::cout<<"v3("<<i4<<") = "<<x4<<" is the maximum value\n";
//! v3(1) = 8 is the maximum value


// Modifications:

v1 << 1, 2, 4, 8, 16, 32;
std::cout<<"v1 = "<<v1<<std::endl;
```

```
//! v1 = 6  ( 1  2  4  8  16  32 )

v1.addToAll(5.);
std::cout<<"v1.addToAll(5.) = "<<v1<<std::endl;
//! v1.addToAll(5.) = 6  ( 6  7  9  13  21  37 )

v1.reverseSelf();
std::cout<<"v1.reverseSelf() = "<<v1<<std::endl;
//! v1.reverseSelf() = 6  ( 37  21  13  9  7  6 )

v1.setZero();
std::cout<<"v1.setZero() = "<<v1<<std::endl;
//! v1.setZero() = 6  ( 0  0  0  0  0  0 )

v1.setAllTo(20.);
std::cout<<"v1.setAllTo(20.) = "<<v1<<std::endl;
//! v1.setAllTo(20.) = 6  ( 20  20  20  20  20  20 )

v1.makeBasis(2);
std::cout<<"v1.makeBasis(2) = "<<v1<<std::endl;
//! v1.makeBasis(2) = 6  ( 0  0  1  0  0  0 )


// Views:

std::cout<<"v3 = "<<v3<<std::endl;
//! v3 = 6  ( 1.1  8  -15  2.5  6.3  -12 )
std::cout<<"v3.subVector(0,3) = "<<v3.subVector(0,3)<<std::endl;
//! v3.subVector(0,3) = 3  ( 1.1  8  -15 )
std::cout<<"v3.subVector(0,6,2) = "<<v3.subVector(0,6,2)<<std::endl;
//! v3.subVector(0,6,2) = 3  ( 1.1  -15  6.3 )
std::cout<<"v3.reverse() = "<<v3.reverse()<<std::endl;
//! v3.reverse() = 6  ( -12  6.3  2.5  -15  8  1.1 )

// Views can be initialized with << too.
v3.reverse() << 1.2, 9., 12, 2.5, -7.4, 14;
std::cout<<"v3 = "<<v3<<std::endl;
//! v3 = 6  ( 14  -7.4  2.5  12  9  1.2 )
v3.subVector(0,6,2) << 18, 22, 33;
std::cout<<"v3 = "<<v3<<std::endl;
//! v3 = 6  ( 18  -7.4  22  12  33  1.2 )

// Can use the views within expressions
v3.reverse() += v4;
std::cout<<"v3.reverse() += v4 => v3 = "<<v3<<std::endl;
//! v3.reverse() += v4 => v3 = 6  ( 32  -14.8  24.5  24  42  2.4 )

v3.subVector(0,3) *= 2.;
std::cout<<"v3.subVector(0,3) *= 2 => v3 = "<<v3<<std::endl;
//! v3.subVector(0,3) *= 2 => v3 = 6  ( 64  -29.6  49  24  42  2.4 )


// Fortran Indexing:

tmv::Vector<double,tmv::FortranStyle> fv3 = v3;
```

```
std::cout<<"fv3 = v3 = "<<fv3<<std::endl;
//! fv3 = v3 = 6  ( 64  -29.6  49  24  42  2.4 )
std::cout<<"fv3(1) = "<<fv3(1)<<std::endl;
//! fv3(1) = 64
std::cout<<"fv3(6) = "<<fv3(6)<<std::endl;
//! fv3(6) = 2.4
std::cout<<"fv3.subVector(1,3) = "<<fv3.subVector(1,3)<<std::endl;
//! fv3.subVector(1,3) = 3  ( 64  -29.6  49 )
std::cout<<"fv3.makeBasis(3) = "<<fv3.makeBasis(3)<<std::endl;
//! fv3.makeBasis(3) = 6  ( 0  0  1  0  0  0 )


// Vector arithmetic:

tmv::Vector<double> v3pv4 = v3 + v4;
std::cout<<"v3 + v4 = "<<v3pv4<<std::endl;
//! v3 + v4 = 6  ( 65.2  -20.6  61  26.5  34.6  16.4 )

// Inner product
double v3v4 = v3 * v4;
std::cout<<"v3 * v4 = "<<v3v4<<std::endl;
//! v3 * v4 = 181.2

v3 *= 2.;
std::cout<<"v3 *= 2 = "<<v3<<std::endl;
//! v3 *= 2 = 6  ( 128  -59.2  98  48  84  4.8 )

v3 += v4;
std::cout<<"v3 += v4 = "<<v3<<std::endl;
//! v3 += v4 = 6  ( 129.2  -50.2  110  50.5  76.6  18.8 )

// Get as complicated as you want:
std::cout<<"(v1*v2) * v3 + ((-v4 + 4.*v1)*v2) * v2/20. =\n"<<
    (v1*v2) * v3 + ((-v4 + 4.*v1)*v2) * v2/20.<<std::endl;
//! (v1*v2) * v3 + ((-v4 + 4.*v1)*v2) * v2/20. =
//! 6  ( 252.94  -105.86  214.54  95.54  147.74  32.14 )

// Automatically checks for aliases:
v3 = v4 - 3.*v3;
std::cout<<"v3 = v4-3.*v3 => v3 = "<<v3<<std::endl;
//! v3 = v4-3.*v3 => v3 = 6  ( -386.4  159.6  -318  -149  -237.2  -42.4 )


// Complex vectors:

tmv::Vector<std::complex<double> > cv4 = v4 * std::complex<double>(1,2);
std::cout<<"cv4 = v4 * (1+2i) =\n"<<cv4<<std::endl;
//! cv4 = v4 * (1+2i) =
//! 6  ( (1.2,2.4)  (9,18)  (12,24)  (2.5,5)  (-7.4,-14.8)  (14,28) )

std::cout<<"cv4.conjugate() =\n"<<cv4.conjugate()<<std::endl;
//! cv4.conjugate() =
//! 6  ( (1.2,-2.4)  (9,-18)  (12,-24)  (2.5,-5)  (-7.4,14.8)  (14,-28) )
std::cout<<"cv4.realPart() = "<<cv4.realPart()<<std::endl;
//! cv4.realPart() = 6  ( 1.2  9  12  2.5  -7.4  14 )
std::cout<<"cv4.imagPart() = "<<cv4.imagPart()<<std::endl;
```

```
//! cv4.imagPart() = 6  ( 2.4  18  24  5  -14.8  28 )
std::cout<<"Norm(cv4) = "<<Norm(cv4)<<std::endl;
//! Norm(cv4) = 49.1655
std::cout<<"sqrt(cv4*cv4.conjugate()) = "<<
     sqrt(cv4*cv4.conjugate())<<std::endl;
//! sqrt(cv4*cv4.conjugate()) = (49.1655,0)
std::cout<<"cv4.maxAbsElement() = "<<cv4.maxAbsElement()<<std::endl;
//! cv4.maxAbsElement() = 31.305

// Can mix real and complex in any combination
std::cout<<"cv4 - v4 = "<<cv4 - v4<<std::endl;
//! cv4 - v4 = 6  ( (0,2.4)  (0,18)  (0,24)  (0,5)  (0,-14.8)  (0,28) )
std::cout<<"cv4 * v4 * (1-2i) = "<<
     cv4 * v4 * std::complex<double>(1,-2)<<std::endl;
//! cv4 * v4 * (1-2i) = (2417.25,0)


// Sorting:

v4 << 2, 5.3, -1.5, -7, 0.5, -2.8;
std::cout<<"v4 = "<<v4<<std::endl;
//! v4 = 6  ( 2  5.3  -1.5  -7  0.5  -2.8 )
tmv::Permutation P;
v4.sort(P);
std::cout<<"Sorted: v4 = "<<v4<<std::endl;
//! Sorted: v4 = 6  ( -7  -2.8  -1.5  0.5  2  5.3 )
v4 = P.inverse() * v4;
std::cout<<"Sort undone: v4 = "<<v4<<std::endl;
//! Sort undone: v4 = 6  ( 2  5.3  -1.5  -7  0.5  -2.8 )
v4.sort(); // Don't necessarily need P.
std::cout<<"Resorted: v4 = "<<v4<<std::endl;
//! Resorted: v4 = 6  ( -7  -2.8  -1.5  0.5  2  5.3 )

// Can sort by other criteria:
std::cout<<"v4.sort(Descend) = "<<v4.sort(tmv::Descend)<<std::endl;
//! v4.sort(Descend) = 6  ( 5.3  2  0.5  -1.5  -2.8  -7 )

cv4.realPart() << -3,  1, -2, -1,  7,  3;
cv4.imagPart() <<  4, -1,  0, -6,  5, -1;
// (I find this complex initialization to be more readable than a list
//  filled with values that look like complex<double>(-3,4), ...)
std::cout<<"cv4 = "<<cv4<<std::endl;
//! cv4 = 6  ( (-3,4)  (1,-1)  (-2,0)  (-1,-6)  (7,5)  (3,-1) )

std::cout<<"cv4.sort(Descend,RealComp) =\n"<<
     cv4.sort(tmv::Descend,tmv::RealComp)<<std::endl;
//! cv4.sort(Descend,RealComp) =
//! 6  ( (7,5)  (3,-1)  (1,-1)  (-1,-6)  (-2,0)  (-3,4) )
std::cout<<"cv4.sort(Ascend,ImagComp) =\n"<<
     cv4.sort(tmv::Ascend,tmv::ImagComp)<<std::endl;
//! cv4.sort(Ascend,ImagComp) =
//! 6  ( (-1,-6)  (3,-1)  (1,-1)  (-2,0)  (-3,4)  (7,5) )
std::cout<<"cv4.sort(Ascend,AbsComp) =\n"<<
     cv4.sort(tmv::Ascend,tmv::AbsComp)<<std::endl;
//! cv4.sort(Ascend,AbsComp) =
//! 6  ( (1,-1)  (-2,0)  (3,-1)  (-3,4)  (-1,-6)  (7,5) )
```

```
    std::cout<<"cv4.sort(Ascend,ArgComp) =\n"<<
        cv4.sort(tmv::Ascend,tmv::ArgComp)<<std::endl;
    //! cv4.sort(Ascend,ArgComp) =
    //! 6 ( (-1,-6) (1,-1) (3,-1) (7,5) (-3,4) (-2,0) )

    // The default component for complex vectors is RealComp:
    std::cout<<"cv4.sort() =\n"<< cv4.sort()<<std::endl;
    //! cv4.sort() =
    //! 6 ( (-3,4) (-2,0) (-1,-6) (1,-1) (3,-1) (7,5) )

    return 0;
} catch (tmv::Error& e) {
    std::cerr<<e<<std::endl;
    return 1;
}
```

## 16.2 Matrix

```
#define TMV_EXTRA_DEBUG // To get the 888's below.

#include "TMV.h"
#include <iostream>

int main() try
{
    // Several ways to create and initialize matrices:

    // Create with uninitialized values
    tmv::Matrix<double> m1(4,3);
    // In debugging mode (the default), these are all initialized to 888
    // to help you more easily notice when you have failed to correctly
    // initialize a matrix.
    std::cout<<"m1 =\n"<<m1;
    //! m1 =
    //! 4  3
    //! (  888  888  888  )
    //! (  888  888  888  )
    //! (  888  888  888  )
    //! (  888  888  888  )

    // Initialize with direct element access:
    for(int i=0;i<m1.nrows();i++)
        for(int j=0;j<m1.ncols();j++)
            m1(i,j) = 2.*i-3.*j+10.;
    std::cout<<"m1 =\n"<<m1;
    //! m1 =
    //! 4  3
    //! (  10  7  4  )
    //! (  12  9  6  )
    //! (  14  11  8  )
    //! (  16  13  10  )

    // Create with all 2's.
    tmv::Matrix<double> m2(4,3,2.);
    std::cout<<"m2 =\n"<<m2;
    //! m2 =
    //! 4  3
    //! (  2  2  2  )
    //! (  2  2  2  )
    //! (  2  2  2  )
    //! (  2  2  2  )

    // Create from given elements in a C array;
    // First in column major order:
    double mm[12] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
    tmv::Matrix<double,tmv::ColMajor> m3(4,3);
    std::copy(mm,mm+12,m3.colmajor_begin());
    std::cout<<"m3 (ColMajor) =\n"<<m3;
    //! m3 (ColMajor) =
    //! 4  3
    //! (  1  5  9  )
    //! ( 2  6  10 )
    //! ( 3  7  11 )
    //! ( 4  8  12 )

    // Can also iterate in row major order, even if the matrix is
    // declared to be ColMajor:
    std::copy(mm,mm+12,m3.rowmajor_begin());
    std::cout<<"m3 (RowMajor) =\n"<<m3;
    //! m3 (RowMajor) =
    //! 4  3
    //! ( 1  2  3 )
    //! ( 4  5  6 )
    //! ( 7  8  9 )
    //! ( 10  11  12 )

    // Initialize with comma-delimited list.
    tmv::Matrix<double,tmv::RowMajor> m4(4,3);
    m4 <<
         2, -4,  1,
         -5,  8,  2,
         1, -7,  4,
         -3,  1,  0;
    std::cout<<"m4 =\n"<<m4;
    //! m4 =
    //! 4  3
    //! ( 2  -4  1 )
    //! ( -5  8  2 )
    //! ( 1  -7  4 )
    //! ( -3  1  0 )

    // The elements are always taken to be listed in row-major order,
    // even if the matrix is declared as ColMajor:
    tmv::Matrix<double,tmv::ColMajor> m5(3,3);
    m5 <<
        0,  1,  2,
        2,  0, -2,
        4, -1, -6;
    std::cout<<"m5 =\n"<<m5;
    //! m5 =
    //! 3  3
    //! ( 0  1  2 )
    //! ( 2  0  -2 )
    //! ( 4  -1  -6 )


    // Norms, etc.

    std::cout<<"Norm1(m1) = "<<Norm1(m1)<<std::endl;
    //! Norm1(m1) = 52
    std::cout<<"Norm2(m1) = "<<Norm2(m1)<<std::endl;
    //! Norm2(m1) = 36.452
    std::cout<<"NormInf(m1) = "<<NormInf(m1)<<std::endl;
    //! NormInf(m1) = 39
    std::cout<<"NormF(m1) = "<<NormF(m1);
    std::cout<<" = "<<Norm(m1)<<std::endl;
    //! NormF(m1) = 36.4966 = 36.4966
```

```
std::cout<<"MaxAbsElement(m1) = "<<MaxAbsElement(m1)<<std::endl;
//! MaxAbsElement(m1) = 16
std::cout<<"Trace(m5) = "<<Trace(m5)<<std::endl;
//! Trace(m5) = -6
std::cout<<"Det(m5) = "<<Det(m5)<<std::endl;
//! Det(m5) = 0


// Views:

std::cout<<"m1 =\n"<<m1;
//! m1 =
//! 4  3
//! ( 10  7  4 )
//! ( 12  9  6 )
//! ( 14  11  8 )
//! ( 16  13  10 )
std::cout<<"m1.row(1) = "<<m1.row(1)<<std::endl;
//! m1.row(1) = 3  ( 12  9  6 )
std::cout<<"m1.col(2) = "<<m1.col(2)<<std::endl;
//! m1.col(2) = 4  ( 4  6  8  10 )
std::cout<<"m1.diag() = "<<m1.diag()<<std::endl;
//! m1.diag() = 3  ( 10  9  8 )
std::cout<<"m1.diag(1) = "<<m1.diag(1)<<std::endl;
//! m1.diag(1) = 2  ( 7  6 )
std::cout<<"m1.diag(-1) = "<<m1.diag(-1)<<std::endl;
//! m1.diag(-1) = 3  ( 12  11  10 )
std::cout<<"m1.subMatrix(2,4,0,2) =\n"<<m1.subMatrix(2,4,0,2);
//! m1.subMatrix(2,4,0,2) =
//! 2  2
//! ( 14  11 )
//! ( 16  13 )
std::cout<<"m1.subMatrix(0,4,1,3,2,1) =\n"<<m1.subMatrix(0,4,1,3,2,1);
//! m1.subMatrix(0,4,1,3,2,1) =
//! 2  2
//! ( 7  4 )
//! ( 11  8 )
std::cout<<"m1.transpose() =\n"<<m1.transpose();
//! m1.transpose() =
//! 3  4
//! ( 10  12  14  16 )
//! ( 7  9  11  13 )
//! ( 4  6  8  10 )
std::cout<<"m1.colRange(1,3) =\n"<<m1.colRange(1,3);
//! m1.colRange(1,3) =
//! 4  2
//! ( 7  4 )
//! ( 9  6 )
//! ( 11  8 )
//! ( 13  10 )
std::cout<<"m1.rowPair(3,0) =\n"<<m1.rowPair(3,0);
//! m1.rowPair(3,0) =
//! 2  3
//! ( 16  13  10 )
//! ( 10  7  4 )
// Can use the views within expressions
```

```
m1.rowRange(0,3) += m5.transpose();
std::cout<<"m1.rowRange(0,3) += m5.transpose() => m1 =\n"<<m1;
//! m1.rowRange(0,3) += m5.transpose() => m1 =
//! 4  3
//! (  10  9  8  )
//! (  13  9  5  )
//! (  16  9  2  )
//! (  16  13  10  )
m1.row(0) *= 2.;
std::cout<<"m1.row(0) *= 2 => m1 =\n"<<m1;
//! m1.row(0) *= 2 => m1 =
//! 4  3
//! (  20  18  16  )
//! (  13  9  5  )
//! (  16  9  2  )
//! (  16  13  10  )


// Fortran Indexing:

tmv::Matrix<double,tmv::FortranStyle> fm1 = m1;
std::cout<<"fm1 = m1 =\n"<<fm1;
//! fm1 = m1 =
//! 4  3
//! (  20  18  16  )
//! (  13  9  5  )
//! (  16  9  2  )
//! (  16  13  10  )
std::cout<<"fm1(1,1) = "<<fm1(1,1)<<std::endl;
//! fm1(1,1) = 20
std::cout<<"fm1(4,3) = "<<fm1(4,3)<<std::endl;
//! fm1(4,3) = 10
std::cout<<"fm1.row(1) = "<<fm1.row(1)<<std::endl;
//! fm1.row(1) = 3  (  20  18  16  )
std::cout<<"fm1.col(3) = "<<fm1.col(3)<<std::endl;
//! fm1.col(3) = 4  (  16  5  2  10  )
std::cout<<"fm1.subMatrix(2,3,1,2) =\n"<<fm1.subMatrix(2,3,1,2);
//! fm1.subMatrix(2,3,1,2) =
//! 2  2
//! (  13  9  )
//! (  16  9  )
std::cout<<"fm1.subMatrix(1,3,2,3,2,1) =\n"<<fm1.subMatrix(1,3,2,3,2,1);
//! fm1.subMatrix(1,3,2,3,2,1) =
//! 2  2
//! (  18  16  )
//! (  9  2  )
std::cout<<"fm1.colRange(1,2) =\n"<<fm1.colRange(1,2);
//! fm1.colRange(1,2) =
//! 4  2
//! (  20  18  )
//! (  13  9  )
//! (  16  9  )
//! (  16  13  )
std::cout<<"fm1.rowPair(4,1) =\n"<<fm1.rowPair(4,1);
//! fm1.rowPair(4,1) =
//! 2  3
//! ( 16  13  10 )
//! ( 20  18  16 )


// Matrix arithmetic:

tmv::Matrix<double> m1pm3 = m1 + m3;
std::cout<<"m1 + m3 =\n"<<m1pm3;
//! m1 + m3 =
//! 4  3
//! ( 21  20  19 )
//! ( 17  14  11 )
//! ( 23  17  11 )
//! ( 26  24  22 )
// Works correctly even if matrices are stored in different order:
tmv::Matrix<double> m3pm4 = m3 + m4;
std::cout<<"m3 + m4 =\n"<<m3pm4;
//! m3 + m4 =
//! 4  3
//! ( 3  -2  4 )
//! ( -1  13  8 )
//! ( 8  1  13 )
//! ( 7  12  12 )

m1 *= 2.;
std::cout<<"m1 *= 2 =\n"<<m1;
//! m1 *= 2 =
//! 4  3
//! ( 40  36  32 )
//! ( 26  18  10 )
//! ( 32  18  4 )
//! ( 32  26  20 )

m1 += m4;
std::cout<<"m1 += m4 =\n"<<m1;
//! m1 += m4 =
//! 4  3
//! ( 42  32  33 )
//! ( 21  26  12 )
//! ( 33  11  8 )
//! ( 29  27  20 )

// Vector outer product
tmv::Vector<double> v1 = m4.col(0);
tmv::Vector<double> v2 = m4.row(1);
tmv::Matrix<double> v1v2 = v1^v2;
std::cout<<"v1 = "<<v1<<std::endl;
//! v1 = 4  ( 2  -5  1  -3 )
std::cout<<"v2 = "<<v2<<std::endl;
//! v2 = 3  ( -5  8  2 )
std::cout<<"v1^v2 =\n"<<v1v2;
//! v1^v2 =
//! 4  3
//! ( -10  16  4 )
//! ( 25  -40  -10 )
//! ( -5  8  2 )
//! ( 15 −24 −6 )
std::cout<<"ColVectorViewOf(v1)*RowVectorViewOf(v2) =\n"<<
    ColVectorViewOf(v1)*RowVectorViewOf(v2);
//! ColVectorViewOf(v1)*RowVectorViewOf(v2) =
//! 4 3
//! ( −10 16 4 )
//! ( 25 −40 −10 )
//! ( −5 8 2 )
//! ( 15 −24 −6 )

// Matrix * Vector product
std::cout<<"m1 * v2 = "<<m1*v2<<std::endl;
//! m1 * v2 = 4 ( 112 127 −61 111 )
std::cout<<"v1 * m1 = "<<v1*m1<<std::endl;
//! v1 * m1 = 3 ( −75 −136 −46 )

// Matrix * Matrix product
tmv::Matrix<double> m1m5 = m1 * m5;
std::cout<<"m1 * m5 =\n"<<m1m5;
//! m1 * m5 =
//! 4 3
//! ( 196 9 −178 )
//! ( 100 9 −82 )
//! ( 54 25 −4 )
//! ( 134 9 −116 )
std::cout<<"m1.row(0) * m5.col(2) = "<<m1.row(0)*m5.col(2)<<std::endl;
//! m1.row(0) * m5.col(2) = −178
std::cout<<"(m1 * m5)(0,2) = "<<m1m5(0,2)<<std::endl;
//! (m1 * m5)(0,2) = −178

// Can handle aliases:
// No alias problem here:
std::cout<<"m1 + 3*m1-m2 =\n"<<m1+3.*m1-m2;
//! m1 + 3*m1-m2 =
//! 4 3
//! ( 166 126 130 )
//! ( 82 102 46 )
//! ( 130 42 30 )
//! ( 114 106 78 )
// But this would be a problem for a naive implementation:
m1 += 3.*m1-m2;
std::cout<<"m1 += 3*m1-m2 =\n"<<m1;
//! m1 += 3*m1-m2 =
//! 4 3
//! ( 166 126 130 )
//! ( 82 102 46 )
//! ( 130 42 30 )
//! ( 114 106 78 )

// Again: here is the correct answer:
std::cout<<"m5 * m5 =\n"<<m5*m5;
//! m5 * m5 =
//! 3 3
//! ( 10 −2 −14 )
//! ( −8 4 16 )
//! ( −26 10 46 )
```

```
// And here, the alias is dealt with correctly:
m5 *= m5;
std::cout<<"m5 *= m5 =\n"<<m5;
//! m5 *= m5 =
//! 3  3
//! ( 10  -2  -14 )
//! ( -8  4  16 )
//! ( -26  10  46 )

// Scalars can be treated as a multiple of identity matrix
m5 += 32.;
std::cout<<"m5 += 32 =\n"<<m5;
//! m5 += 32 =
//! 3  3
//! ( 42  -2  -14 )
//! ( -8  36  16 )
//! ( -26  10  78 )


// Complex matrices:

tmv::Matrix<std::complex<double> > cm4 = m4 * std::complex<double>(1,2);
std::cout<<"cm4 = m4 * (1+2i) =\n"<<cm4;
//! cm4 = m4 * (1+2i) =
//! 4  3
//! ( (2,4)  (-4,-8)  (1,2) )
//! ( (-5,-10)  (8,16)  (2,4) )
//! ( (1,2)  (-7,-14)  (4,8) )
//! ( (-3,-6)  (1,2)  (0,0) )
std::cout<<"cm4.conjugate() =\n"<<cm4.conjugate();
//! cm4.conjugate() =
//! 4  3
//! ( (2,-4)  (-4,8)  (1,-2) )
//! ( (-5,10)  (8,-16)  (2,-4) )
//! ( (1,-2)  (-7,14)  (4,-8) )
//! ( (-3,6)  (1,-2)  (0,-0) )
std::cout<<"cm4.transpose() =\n"<<cm4.transpose();
//! cm4.transpose() =
//! 3  4
//! ( (2,4)  (-5,-10)  (1,2)  (-3,-6) )
//! ( (-4,-8)  (8,16)  (-7,-14)  (1,2) )
//! ( (1,2)  (2,4)  (4,8)  (0,0) )
std::cout<<"cm4.adjoint() =\n"<<cm4.adjoint();
//! cm4.adjoint() =
//! 3  4
//! ( (2,-4)  (-5,10)  (1,-2)  (-3,6) )
//! ( (-4,8)  (8,-16)  (-7,14)  (1,-2) )
//! ( (1,-2)  (2,-4)  (4,-8)  (0,-0) )
std::cout<<"cm4.realPart() =\n"<<cm4.realPart();
//! cm4.realPart() =
//! 4  3
//! ( 2  -4  1 )
//! ( -5  8  2 )
//! ( 1  -7  4 )
//! ( -3  1  0 )
std::cout<<"cm4.imagPart() =\n"<<cm4.imagPart();
```

```
    //! cm4.imagPart() =
    //! 4  3
    //! ( 4  -8  2 )
    //! ( -10  16  4 )
    //! ( 2  -14  8 )
    //! ( -6  2  0 )
    std::cout<<"Norm(cm4) = "<<Norm(cm4)<<std::endl;
    //! Norm(cm4) = 30.8221
    std::cout<<"cm4*cm4.adjoint() =\n"<<cm4*cm4.adjoint();
    //! cm4*cm4.adjoint() =
    //! 4  4
    //! ( (105,0)  (-200,0)  (170,0)  (-50,0) )
    //! ( (-200,0)  (465,0)  (-265,0)  (115,0) )
    //! ( (170,0)  (-265,0)  (330,0)  (-50,0) )
    //! ( (-50,0)  (115,0)  (-50,0)  (50,0) )

    // Can mix real and complex in any combination
    std::cout<<"cm4 - m4 =\n"<<cm4 - m4;
    //! cm4 - m4 =
    //! 4  3
    //! ( (0,4)  (0,-8)  (0,2) )
    //! ( (0,-10)  (0,16)  (0,4) )
    //! ( (0,2)  (0,-14)  (0,8) )
    //! ( (0,-6)  (0,2)  (0,0) )
    std::cout<<"cm4 * m4.transpose() * (1-2i) =\n"<<
        cm4 * m4.transpose() * std::complex<double>(1,-2);
    //! cm4 * m4.transpose() * (1-2i) =
    //! 4  4
    //! ( (105,0)  (-200,0)  (170,0)  (-50,0) )
    //! ( (-200,0)  (465,0)  (-265,0)  (115,0) )
    //! ( (170,0)  (-265,0)  (330,0)  (-50,0) )
    //! ( (-50,0)  (115,0)  (-50,0)  (50,0) )

    return 0;
} catch (tmv::Error& e) {
    std::cerr<<e<<std::endl;
    return 1;
}
```

## 16.3 Division

```
#define TMV_EXTRA_DEBUG

#include "TMV.h"
#include "TMV_Diag.h"
#include <iostream>

int main() try
{
    tmv::Matrix<double> A(4,4);
    for(int i=0;i<A.nrows();i++)
        for(int j=0;j<A.ncols();j++)
            A(i,j) = 6.*i-2*j*j+2.;
    A.diag().addToAll(5.);

    tmv::Vector<double> b(4);
    for(int i=0;i<b.size();i++)
        b(i) = 3.+2.*i;

    // Basic Ax=b solution:
    std::cout<<"A =\n"<<A;
    //! A =
    //! 4  4
    //! ( 7   0   -6   -16 )
    //! ( 8   11   0   -10 )
    //! ( 14   12   11   -4 )
    //! ( 20   18   12   7 )
    std::cout<<"b = "<<b<<std::endl;
    //! b = 4  ( 3  5  7  9 )

    // Solve: Ax = b
    tmv::Vector<double> x = b/A; // Default: use LU decomposition
    std::cout<<"x = b/A = "<<x<<std::endl;
    //! x = b/A = 4  ( 0.294545  0.170909  0.0472727  -0.0763636 )
    std::cout<<"Check: A*x = "<<A*x<<std::endl;
    //! Check: A*x = 4  ( 3  5  7  9 )

    // Can update A and then re-solve:
    A(0,0) = 20.;
    x = b/A;
    std::cout<<"Now x = b/A = "<<x<<std::endl;
    //! Now x = b/A = 4  ( 0.133458  0.29877  0.117091  -0.064587 )
    std::cout<<"A*x = "<<A*x<<std::endl;
    //! A*x = 4  ( 3  5  7  9 )

    // If the matrix won't change, but you want to solve with multiple
    // vectors, then it's faster to let TMV know this with saveDiv().
    tmv::Matrix<double> A2 = A;
    A2.saveDiv();
    x = b/A2;
    std::cout<<"x1 = "<<x<<std::endl;
    //! x1 = 4  ( 0.133458  0.29877  0.117091  -0.064587 )
    std::cout<<"A*x = "<<A2*x<<std::endl;
    //! A*x = 4  ( 3  5  7  9 )
```

```
x = b.reverse()/A2; // Fast, since doesn't recalculate LU decomposition.
std::cout<<"x2 = "<<x<<std::endl;
//! x2 = 4  ( 0.136753  0.207381  -0.0775483  -0.362478 )
std::cout<<"A*x2 = "<<A2*x<<std::endl;
//! A*x2 = 4  ( 9  7  5  3 )
A2.row(0) *= 2.;
x = b/A2;  // Wrong, since doesn't recalculate LU decomposition.
std::cout<<"Wrong x = "<<x<<std::endl;
//! Wrong x = 4  ( 0.133458  0.29877  0.117091  -0.064587 )
std::cout<<"A*x = "<<A2*x<<std::endl;
//! A*x = 4  ( 6  5  7  9 )
// If the matrix does change when saveDiv() is set,
// you can manually recalculate the decomposition:
A2.resetDiv();
x = b/A2;  // Now it is correct.
std::cout<<"x = "<<x<<std::endl;
//! x = 4  ( 0.0703537  0.348858  0.144442  -0.0599736 )
std::cout<<"A*x = "<<A2*x<<std::endl;
//! A*x = 4  ( 3  5  7  9 )

// Matrix inverse:
tmv::Matrix<double> A2inv = A2.inverse();
std::cout<<"Ainv =\n"<<A2inv;
//! Ainv =
//! 4  4
//! ( 0.0210347  -0.0395431  -0.012522  0.0325132 )
//! ( -0.016696  0.0966608  -0.0527241  0.0316344 )
//! ( -0.00911687  -0.0643234  0.139631  -0.0537786 )
//! ( -0.00153779  -0.0253076  -0.0680141  0.0608084 )
std::cout<<"Ainv*A =\n"<<A2inv*A2;
//! Ainv*A =
//! 4  4
//! ( 1  0  5.55112e-17  1.66533e-16 )
//! ( 1.11022e-16  1  -1.66533e-16  -4.996e-16 )
//! ( 0  2.22045e-16  1  2.22045e-16 )
//! ( 0  -2.22045e-16  -1.11022e-16  1 )

// This is a case where it can be useful to see the
// matrix elements that are larger than some threshold value:
std::cout<<"Ainv*A =\n" << tmv::ThreshIO(1.e-8) << (A2inv*A2);
//! Ainv*A =
//! 4  4
//! ( 1  0  0  0 )
//! ( 0  1  0  0 )
//! ( 0  0  1  0 )
//! ( 0  0  0  1 )

// 1/x notation is treated as arithmetic:
// (But the 1 has to be the same type as the elements of the matrix.
/// In this case, double.  With a float matrix, use 1.F instead.)
std::cout<<"1./A =\n"<<1./A2;
//! 1./A =
//! 4  4
//! ( 0.0210347  -0.0395431  -0.012522  0.0325132 )
//! ( -0.016696  0.0966608  -0.0527241  0.0316344 )
//! ( -0.00911687  -0.0643234  0.139631  -0.0537786 )
//! ( −0.00153779   −0.0253076   −0.0680141   0.0608084 )
std::cout<<"5./A =\n"<<5./A2;
//! 5./A =
//! 4  4
//! ( 0.105174   −0.197715   −0.0626098   0.162566 )
//! ( −0.0834798   0.483304   −0.26362   0.158172 )
//! ( −0.0455844   −0.321617   0.698155   −0.268893 )
//! ( −0.00768893   −0.126538   −0.34007   0.304042 )

// Can also use inverse() notation instead of /
// x = b/A2 inlines to exactly the same thing as x = A2.inverse() * b;
// ie. accurate back−substitution methods are used rather than
// actually computing the inverse and then multiplying:
x =  A2.inverse() * b;
std::cout<<"x = A.inverse() * b = "<<x<<std::endl;
//! x = A.inverse() * b = 4  ( 0.0703537   0.348858   0.144442   −0.0599736 )

// Division from the other side can either be done with this
// inverse() notation or with the % operator:
// This is the solution to the equation x A = b, rather than A x = b.
x = b * A2.inverse();
std::cout<<"x = b * A.inverse() = "<<x<<std::endl;
//! x = b * A.inverse() = 4  ( −0.0980338   −0.313357   0.0641037   0.426538 )
x = b % A2;
std::cout<<"x = b % A = "<<x<<std::endl;
//! x = b % A = 4  ( −0.0980338   −0.313357   0.0641037   0.426538 )
std::cout<<"Check: x*A = "<<x*A2<<std::endl;
//! Check: x*A = 4  ( 3  5  7  9 )

tmv::Matrix<double> B(4,3);
for(int i=0;i<B.nrows();i++)
    for(int j=0;j<B.ncols();j++)
        B(i,j) = 1.+2.*i+j*j;
// Multiple right hand sides may be calculated at once if
// B is a matrix, rather than a vector:
std::cout<<"B =\n"<<B;
//! B =
//! 4  3
//! ( 1  2  5 )
//! ( 3  4  7 )
//! ( 5  6  9 )
//! ( 7  8  11 )
tmv::Matrix<double> X = B/A2;
std::cout<<"X = B/A =\n"<<X;
//! X = B/A =
//! 4  3
//! ( 0.067388   0.0688708   0.0733194 )
//! ( 0.231107   0.289982   0.466608 )
//! ( 0.119618   0.13203   0.169266 )
//! ( 0.0081283   −0.0259227   −0.128076 )
std::cout<<"AX = "<<A2*X;
//! AX = 4  3
//! ( 1  2  5 )
//! ( 3  4  7 )
//! ( 5  6  9 )
//! ( 7  8  11 )
```

```
// And as always, you can mix complex and real objects:
tmv::Vector<std::complex<double> > cb = b * std::complex<double>(3,-2);
cb(1) = std::complex<double>(-1,8);
cb(2) = std::complex<double>(1,1);
std::cout<<"cb = "<<cb<<std::endl;
//! cb = 4  ( (9,-6)  (-1,8)  (1,1)  (27,-18) )
tmv::Vector<std::complex<double> > cx = cb/A;
std::cout<<"cx = cb/A =\n"<<cx<<std::endl;
//! cx = cb/A =
//! 4  ( (1.2835,-1.16652)  (0.404218,0.351494)  (-1.41217,0.70246)
    (1.57144,-1.34657) )
std::cout<<"A*cx = "<<A*cx<<std::endl;
//! A*cx = 4  ( (9,-6)  (-1,8)  (1,1)  (27,-18) )
tmv::Matrix<std::complex<double> > CA = A * std::complex<double>(5,-2);
CA(1,1) = std::complex<double>(1,6);
CA(2,3) = std::complex<double>(4,-1);
std::cout<<"CA = "<<CA;
//! CA = 4  4
//! ( (100,-40)  (0,0)  (-30,12)  (-80,32) )
//! ( (40,-16)  (1,6)  (0,0)  (-50,20) )
//! ( (70,-28)  (60,-24)  (55,-22)  (4,-1) )
//! ( (100,-40)  (90,-36)  (60,-24)  (35,-14) )
std::cout<<"cx = b/CA =\n"<<(cx=b/CA)<<std::endl;
//! cx = b/CA =
//! 4  ( (-0.0858093,0.219988)  (0.270489,-0.423628)  (-0.0665359,0.214597)
    (-0.114638,0.18158) )
std::cout<<"CA*cx = "<<tmv::ThreshIO(1.e-8)<<CA*cx<<std::endl;
//! CA*cx = 4  ( (3,0)  (5,0)  (7,0)  (9,0) )
std::cout<<"cb/CA =\n"<<(cx=cb/CA)<<std::endl;
//! cb/CA =
//! 4  ( (0.0374442,0.698211)  (0.792852,-1.67098)  (-0.916414,0.915387)
    (0.267616,0.555355) )
std::cout<<"CA*cx = "<<CA*cx<<std::endl;
//! CA*cx = 4  ( (9,-6)  (-1,8)  (1,1)  (27,-18) )


// Least-squares solutions:
// If A in the matrix equation A x = b has more rows than columns,
// then there is, in general, no solution to the equation.
// Instead, one is generally looking for the x that comes closest
// to satisfying the equation, in a least-squares sense.
// Specifically, the x for which Norm2(b-Ax) is minimized.
// This is the solution produced by TMV for such matrices.
//
// Here I model a theoretical system for which each observation
// is 5 + 6i - 3i^2 in the absense of measurement errors.
// We observe the actual values which are not quite equal to
// that because of noise.  And the goal is to determine the
// coefficients of 1,i,i^2 (5,6,-3) from noisy observations:
tmv::Vector<double> b3(10);
tmv::Matrix<double> A3(10,3);
double errors[10] = {0.01,-0.02,0.02,-0.02,0.00,-0.03,0.01,-0.02,0.03,0.02};
for(int i=0;i<10;i++) {
    b3(i) = 5. + 6.*i - 3.*i*i + errors[i]; // Model of measurements
    A3(i,0) = 1.;  // Parameterization of the model...
    A3(i,1) = i;
    A3(i,2) = i*i;
}
double sigma = 0.02; // sigma = estimate of rms errors
A3 /= sigma;
b3 /= sigma;

tmv::Vector<double> x3 = b3/A3;  // Uses QR decomposition by default
std::cout<<"x = "<<x3<<std::endl;
//! x = 3  ( 5.00773  5.98989  -2.99867 )
std::cout<<"A*x => \n"<<A3*x3<<std::endl;
//! A*x =>
//! 10  ( 250.386  399.947  249.64  -200.534  -950.576  -2000.48  -3350.26
    -4999.91  -6949.42  -9198.8 )
std::cout<<"chisq = NormSq(A*x-b) = "<<NormSq(A3*x3-b3)<<std::endl;
//! chisq = NormSq(A*x-b) = 6.99811
// The expected value for this is 10 observations minus 3 degrees
// of freedom = 7.

// The covariance matrix for x is (A.Transpose() * A)^-1
// This combination is easy to calculate from the QR decomposition
// that TMV has used to do the division.  Therefore, we provide
// it as an explicit function:
tmv::Matrix<double> cov(3,3);
A3.makeInverseATA(cov);
std::cout<<"Cov(x) =\n"<<cov;
//! Cov(x) =
//! 3  3
//! ( 0.000247273  -0.000103636  9.09091e-06 )
//! ( -0.000103636  6.62121e-05  -6.81818e-06 )
//! ( 9.09091e-06  -6.81818e-06  7.57576e-07 )

// The singular value decomposition can detect ill-conditioned matrices
// and correct for them.
// For example, if you model the above observations with the
// 1,i,i^2 components as before, but add as well 6*i-5 as a component,
// then that would be degenerate with 1 and i.
// SVD is able to detect this defect and deal with it appropriately:
tmv::Matrix<double> A4(10,4);
A4.colRange(0,3) = A3*sigma;
for(int i=0;i<10;i++) A4(i,3) = 6.*i-5.;
std::cout<<"Now A*sigma =\n"<<A4;
//! Now A*sigma =
//! 10  4
//! ( 1  0  0  -5 )
//! ( 1  1  1  1 )
//! ( 1  2  4  7 )
//! ( 1  3  9  13 )
//! ( 1  4  16  19 )
//! ( 1  5  25  25 )
//! ( 1  6  36  31 )
//! ( 1  7  49  37 )
//! ( 1  8  64  43 )
//! ( 1  9  81  49 )
A4 /= sigma;
try {
        // This may or may not succeed, but if it does, the results will be
        // unusable, typically with values around 1.e13 and such.
        tmv::Vector<double> x4 = b3/A4;
        std::cout<<"Unstable x = b/A =\n"<<x4<<std::endl;
        //! Unstable x = b/A =
        //! 4  ( -1.31399e+13  1.57679e+13  -2.99808  -2.62798e+12 )
        std::cout<<"A*x =\n"<<A4*x4<<std::endl;
        //! A*x =
        //! 10  ( 251.125  400.203  249.5  -201.25  -951.5  -2001  -3351.5  -5000
            -6950  -9199 )
        std::cout<<"chisq = "<<NormSq(A4*x4-b3);
        std::cout<<"       Norm(x) = "<<Norm(x4)<<std::endl;
        //! chisq = 13.9006       Norm(x) = 2.06927e+13
    } catch (tmv::Error& e) {
        std::cout<<"Tried x = b/A, but caught error: \n"<<e<<std::endl;
    }

    // So instead, tell TMV to use SVD for division rather than QR.
    A4.divideUsing(tmv::SV);
    std::cout<<"Singular values for A are "<<A4.svd().getS().diag()<<std::endl;
    //! Singular values for A are 4  ( 7618.67  733.074  116.312  5.15828e-14 )
    std::cout<<"Using only the first "<<A4.svd().getKMax()<<" components\n";
    //! Using only the first 3 components
    tmv::Vector<double> x4 = b3/A4;
    std::cout<<"SVD division yields: x = "<<x4<<std::endl;
    //! SVD division yields: x = 4  ( 5.88681  4.93498  -2.99867  0.175817 )
    std::cout<<"chisq = "<<NormSq(A4*x4-b3);
    std::cout<<"       Norm(x) = "<<Norm(x4)<<std::endl;
    //! chisq = 6.99811       Norm(x) = 8.24813

    // QRP can also give useful results, but isn't quite as flexible
    // as SVD:
    A4.divideUsing(tmv::QRP);
    x4 = b3/A4;
    std::cout<<"QRP division yields: x = "<<x4<<std::endl;
    std::cout<<"chisq = "<<NormSq(A4*x4-b3);
    std::cout<<"       Norm(x) = "<<Norm(x4)<<std::endl;
    //! QRP division yields: x = 4  ( 5.00773  5.98989  -2.99867  0 )
    //! chisq = 6.99811       Norm(x) = 8.3635

    // Note that both methods give answers with an equal chisq, so
    // Ax is equally close to b for each but they have different
    // specific choices for the degeneracy.
    // SVD will give the solution within this degeneracy freedom that
    // has the minimum Norm(x), but QRP will be faster --
    // significantly so for large matrices.

    return 0;
} catch (tmv::Error& e) {
    std::cerr<<e<<std::endl;
    return 1;
}
```

## 16.4 BandMatrix

```
#define TMV_EXTRA_DEBUG

#include "TMV.h"
// Note: extra include file for BandMatrix
#include "TMV_Band.h"
// Also need to link with -ltmv_symband
#include <iostream>

int main() try
{
    // Several ways to create and initialize band matrices:

    // Create with uninitialized values
    tmv::BandMatrix<double> m1(6,6,1,2);
    for(int i=0;i<m1.nrows();i++)
        for(int j=0;j<m1.ncols();j++)
            if (i<=j+m1.nlo() && j<=i+m1.nhi())
                m1(i,j) = 3.*i-j*j+7.;
    std::cout<<"m1 =\n"<<m1;
    //! m1 =
    //! 6  6
    //! ( 7  6  3  0  0  0 )
    //! ( 10  9  6  1  0  0 )
    //! ( 0  12  9  4  -3  0 )
    //! ( 0  0  12  7  0  -9 )
    //! ( 0  0  0  10  3  -6 )
    //! ( 0  0  0  0  6  -3 )

    // Create with all 2's.
    tmv::BandMatrix<double> m2(6,6,1,3,2.);
    std::cout<<"m2 =\n"<<m2;
    //! m2 =
    //! 6  6
    //! ( 2  2  2  2  0  0 )
    //! ( 2  2  2  2  2  0 )
    //! ( 0  2  2  2  2  2 )
    //! ( 0  0  2  2  2  2 )
    //! ( 0  0  0  2  2  2 )
    //! ( 0  0  0  0  2  2 )

    // A BandMatrix can be non-square:
    tmv::BandMatrix<double> m3(6,8,1,3,2.);
    std::cout<<"m3 =\n"<<m3;
    //! m3 =
    //! 6  8
    //! ( 2  2  2  2  0  0  0  0 )
    //! ( 2  2  2  2  2  0  0  0 )
    //! ( 0  2  2  2  2  2  0  0 )
    //! ( 0  0  2  2  2  2  2  0 )
    //! ( 0  0  0  2  2  2  2  2 )
    //! ( 0  0  0  0  2  2  2  2 )

    // Create from given elements:
```

```
double mm[20] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
tmv::BandMatrix<double,tmv::ColMajor> m4(6,6,2,1);
std::copy(mm,mm+20,m4.colmajor_begin());
std::cout<<"m4 (ColMajor) =\n"<<m4;
//! m4 (ColMajor) =
//! 6  6
//! ( 1  4  0  0  0  0 )
//! ( 2  5  8  0  0  0 )
//! ( 3  6  9  12  0  0 )
//! ( 0  7  10  13  16  0 )
//! ( 0  0  11  14  17  19 )
//! ( 0  0  0  15  18  20 )
tmv::BandMatrix<double,tmv::RowMajor> m5(6,6,2,1);
std::copy(mm,mm+20,m5.rowmajor_begin());
std::cout<<"m5 (RowMajor) =\n"<<m5;
//! m5 (RowMajor) =
//! 6  6
//! ( 1  2  0  0  0  0 )
//! ( 3  4  5  0  0  0 )
//! ( 6  7  8  9  0  0 )
//! ( 0  10  11  12  13  0 )
//! ( 0  0  14  15  16  17 )
//! ( 0  0  0  18  19  20 )
tmv::BandMatrix<double,tmv::DiagMajor> m6(6,6,2,1);
std::copy(mm,mm+20,m6.diagmajor_begin());
std::cout<<"m6 (DiagMajor) =\n"<<m6;
//! m6 (DiagMajor) =
//! 6  6
//! ( 10  16  0  0  0  0 )
//! ( 5  11  17  0  0  0 )
//! ( 1  6  12  18  0  0 )
//! ( 0  2  7  13  19  0 )
//! ( 0  0  3  8  14  20 )
//! ( 0  0  0  4  9  15 )

// Can make from the banded portion of a regular Matrix:
tmv::Matrix<double> xm(6,6);
for(int i=0;i<xm.nrows();i++)
    for(int j=0;j<xm.ncols();j++)
        xm(i,j) = 5.*i-j*j+3.;
tmv::BandMatrix<double> m7(xm,3,2);
std::cout<<"m7 =\n"<<m7;
//! m7 =
//! 6  6
//! ( 3  2  -1  0  0  0 )
//! ( 8  7  4  -1  0  0 )
//! ( 13  12  9  4  -3  0 )
//! ( 18  17  14  9  2  -7 )
//! ( 0  22  19  14  7  -2 )
//! ( 0  0  24  19  12  3 )
// Or from a wider BandMatrix:
tmv::BandMatrix<double> m8(m7,3,0);
std::cout<<"m8 =\n"<<m8;
//! m8 =
//! 6  6
//! ( 3  0  0  0  0  0 )
//! ( 8  7  0  0  0  0 )
//! ( 13  12  9  0  0  0 )
//! ( 18  17  14  9  0  0 )
//! ( 0  22  19  14  7  0 )
//! ( 0  0  24  19  12  3 )

// Shortcuts to Bi- and Tri-diagonal matrices:
tmv::Vector<double> v1(5,1.);
tmv::Vector<double> v2(6,2.);
tmv::Vector<double> v3(5,3.);
tmv::BandMatrix<double> m9 = LowerBiDiagMatrix(v1,v2);
tmv::BandMatrix<double> m10 = UpperBiDiagMatrix(v2,v3);
tmv::BandMatrix<double> m11 = TriDiagMatrix(v1,v2,v3);
std::cout<<"LowerBiDiagMatrix(v1,v2) =\n"<<m9;
//! LowerBiDiagMatrix(v1,v2) =
//! 6  6
//! ( 2  0  0  0  0  0 )
//! ( 1  2  0  0  0  0 )
//! ( 0  1  2  0  0  0 )
//! ( 0  0  1  2  0  0 )
//! ( 0  0  0  1  2  0 )
//! ( 0  0  0  0  1  2 )
std::cout<<"UpperBiDiagMatrix(v2,v3) =\n"<<m10;
//! UpperBiDiagMatrix(v2,v3) =
//! 6  6
//! ( 2  3  0  0  0  0 )
//! ( 0  2  3  0  0  0 )
//! ( 0  0  2  3  0  0 )
//! ( 0  0  0  2  3  0 )
//! ( 0  0  0  0  2  3 )
//! ( 0  0  0  0  0  2 )
std::cout<<"TriDiagMatrix(v1,v2,v3) =\n"<<m11;
//! TriDiagMatrix(v1,v2,v3) =
//! 6  6
//! ( 2  3  0  0  0  0 )
//! ( 1  2  3  0  0  0 )
//! ( 0  1  2  3  0  0 )
//! ( 0  0  1  2  3  0 )
//! ( 0  0  0  1  2  3 )
//! ( 0  0  0  0  1  2 )


// Norms, etc.

std::cout<<"Norm1(m1) = "<<Norm1(m1)<<std::endl;
//! Norm1(m1) = 30
std::cout<<"Norm2(m1) = "<<Norm2(m1)<<std::endl;
//! Norm2(m1) = 24.0314
std::cout<<"NormInf(m1) = "<<NormInf(m1)<<std::endl;
//! NormInf(m1) = 28
std::cout<<"NormF(m1) = "<<NormF(m1)<<" = "<<Norm(m1)<<std::endl;
//! NormF(m1) = 32.0312 = 32.0312
std::cout<<"MaxAbsElement(m1) = "<<MaxAbsElement(m1)<<std::endl;
//! MaxAbsElement(m1) = 12
std::cout<<"Trace(m1) = "<<Trace(m1)<<std::endl;
//! Trace(m1) = 32
```

```
std::cout<<"Det(m1) = "<<Det(m1)<<std::endl;
//! Det(m1) = 67635


// Views:

std::cout<<"m1 =\n"<<m1;
//! m1 =
//! 6 6
//! ( 7 6 3 0 0 0 )
//! ( 10 9 6 1 0 0 )
//! ( 0 12 9 4 -3 0 )
//! ( 0 0 12 7 0 -9 )
//! ( 0 0 0 10 3 -6 )
//! ( 0 0 0 0 6 -3 )
std::cout<<"m1.diag() = "<<m1.diag()<<std::endl;
//! m1.diag() = 6 ( 7 9 9 7 3 -3 )
std::cout<<"m1.diag(1) = "<<m1.diag(1)<<std::endl;
//! m1.diag(1) = 5 ( 6 6 4 0 -6 )
std::cout<<"m1.diag(-1) = "<<m1.diag(-1)<<std::endl;
//! m1.diag(-1) = 5 ( 10 12 12 10 6 )
std::cout<<"m1.subBandMatrix(0,3,0,3,1,1) =\n"<<
    m1.subBandMatrix(0,3,0,3,1,1);
//! m1.subBandMatrix(0,3,0,3,1,1) =
//! 3 3
//! ( 7 6 0 )
//! ( 10 9 6 )
//! ( 0 12 9 )
std::cout<<"m1.transpose() =\n"<<m1.transpose();
//! m1.transpose() =
//! 6 6
//! ( 7 10 0 0 0 0 )
//! ( 6 9 12 0 0 0 )
//! ( 3 6 9 12 0 0 )
//! ( 0 1 4 7 10 0 )
//! ( 0 0 -3 0 3 6 )
//! ( 0 0 0 -9 -6 -3 )

// rowRange, colRange shrink both dimensions of the matrix to include only
// the portions that are in those rows or columns:
std::cout<<"m1.rowRange(0,4) =\n"<<m1.rowRange(0,4);
//! m1.rowRange(0,4) =
//! 4 6
//! ( 7 6 3 0 0 0 )
//! ( 10 9 6 1 0 0 )
//! ( 0 12 9 4 -3 0 )
//! ( 0 0 12 7 0 -9 )
std::cout<<"m1.colRange(1,4) =\n"<<m1.colRange(1,4);
//! m1.colRange(1,4) =
//! 5 3
//! ( 6 3 0 )
//! ( 9 6 1 )
//! ( 12 9 4 )
//! ( 0 12 7 )
//! ( 0 0 10 )
std::cout<<"m1.diagRange(0,2) =\n"<<m1.diagRange(0,2);
```

```
//! m1.diagRange(0,2) =
//! 6  6
//! ( 7  6  0  0  0  0 )
//! ( 0  9  6  0  0  0 )
//! ( 0  0  9  4  0  0 )
//! ( 0  0  0  7  0  0 )
//! ( 0  0  0  0  3  -6 )
//! ( 0  0  0  0  0  -3 )
std::cout<<"m1.diagRange(-1,1) =\n"<<m1.diagRange(-1,1);
//! m1.diagRange(-1,1) =
//! 6  6
//! ( 7  0  0  0  0  0 )
//! ( 10  9  0  0  0  0 )
//! ( 0  12  9  0  0  0 )
//! ( 0  0  12  7  0  0 )
//! ( 0  0  0  10  3  0 )
//! ( 0  0  0  0  6  -3 )


// Fortran Indexing:

tmv::BandMatrix<double,tmv::FortranStyle> fm1 = m1;
std::cout<<"fm1 = m1 =\n"<<fm1;
//! fm1 = m1 =
//! 6  6
//! ( 7  6  3  0  0  0 )
//! ( 10  9  6  1  0  0 )
//! ( 0  12  9  4  -3  0 )
//! ( 0  0  12  7  0  -9 )
//! ( 0  0  0  10  3  -6 )
//! ( 0  0  0  0  6  -3 )
std::cout<<"fm1(1,1) = "<<fm1(1,1)<<std::endl;
//! fm1(1,1) = 7
std::cout<<"fm1(4,3) = "<<fm1(4,3)<<std::endl;
//! fm1(4,3) = 12
std::cout<<"fm1.subBandMatrix(1,3,1,3,1,1) =\n"<<
    fm1.subBandMatrix(1,3,1,3,1,1);
//! fm1.subBandMatrix(1,3,1,3,1,1) =
//! 3  3
//! ( 7  6  0 )
//! ( 10  9  6 )
//! ( 0  12  9 )
std::cout<<"fm1.rowRange(1,4) =\n"<<fm1.rowRange(1,4);
//! fm1.rowRange(1,4) =
//! 4  6
//! ( 7  6  3  0  0  0 )
//! ( 10  9  6  1  0  0 )
//! ( 0  12  9  4  -3  0 )
//! ( 0  0  12  7  0  -9 )
std::cout<<"fm1.colRange(2,4) =\n"<<fm1.colRange(2,4);
//! fm1.colRange(2,4) =
//! 5  3
//! ( 6  3  0 )
//! ( 9  6  1 )
//! ( 12  9  4 )
//! ( 0  12  7 )
//! ( 0 0 10 )
std::cout<<"fm1.diagRange(0,1) =\n"<<fm1.diagRange(0,1);
//! fm1.diagRange(0,1) =
//! 6 6
//! ( 7 6 0 0 0 0 )
//! ( 0 9 6 0 0 0 )
//! ( 0 0 9 4 0 0 )
//! ( 0 0 0 7 0 0 )
//! ( 0 0 0 0 3 -6 )
//! ( 0 0 0 0 0 -3 )
std::cout<<"fm1.diagRange(-1,0) =\n"<<fm1.diagRange(-1,0);
//! fm1.diagRange(-1,0) =
//! 6 6
//! ( 7 0 0 0 0 0 )
//! ( 10 9 0 0 0 0 )
//! ( 0 12 9 0 0 0 )
//! ( 0 0 12 7 0 0 )
//! ( 0 0 0 10 3 0 )
//! ( 0 0 0 0 6 -3 )


// Matrix arithmetic:

tmv::BandMatrix<double> m1pm2 = m1 + m2;
std::cout<<"m1 + m2 =\n"<<m1pm2;
//! m1 + m2 =
//! 6 6
//! ( 9 8 5 2 0 0 )
//! ( 12 11 8 3 2 0 )
//! ( 0 14 11 6 -1 2 )
//! ( 0 0 14 9 2 -7 )
//! ( 0 0 0 12 5 -4 )
//! ( 0 0 0 0 8 -1 )
// Works correctly even if matrices are stored in different order:
tmv::BandMatrix<double> m5pm6 = m5 + m6;
std::cout<<"m5 + m6 =\n"<<m5pm6;
//! m5 + m6 =
//! 6 6
//! ( 11 18 0 0 0 0 )
//! ( 8 15 22 0 0 0 )
//! ( 7 13 20 27 0 0 )
//! ( 0 12 18 25 32 0 )
//! ( 0 0 17 23 30 37 )
//! ( 0 0 0 22 28 35 )
// Also expands the number of off-diagonals appropriately as needed:
tmv::BandMatrix<double> m2pm4 = m2 + m4;
std::cout<<"m2 + m4 =\n"<<m2pm4;
//! m2 + m4 =
//! 6 6
//! ( 3 6 2 2 0 0 )
//! ( 4 7 10 2 2 0 )
//! ( 3 8 11 14 2 2 )
//! ( 0 7 12 15 18 2 )
//! ( 0 0 11 16 19 21 )
//! ( 0 0 0 15 20 22 )
```

```
m1 *= 2.;
std::cout<<"m1 *= 2 =\n"<<m1;
//! m1 *= 2 =
//! 6  6
//! ( 14  12  6  0  0  0 )
//! ( 20  18  12  2  0  0 )
//! ( 0  24  18  8  -6  0 )
//! ( 0  0  24  14  0  -18 )
//! ( 0  0  0  20  6  -12 )
//! ( 0  0  0  0  12  -6 )

m2 += m1;
std::cout<<"m2 += m1 =\n"<<m2;
//! m2 += m1 =
//! 6  6
//! ( 16  14  8  2  0  0 )
//! ( 22  20  14  4  2  0 )
//! ( 0  26  20  10  -4  2 )
//! ( 0  0  26  16  2  -16 )
//! ( 0  0  0  22  8  -10 )
//! ( 0  0  0  0  14  -4 )

tmv::Vector<double> v = xm.col(0);
std::cout<<"v = "<<v<<std::endl;
//! v = 6  ( 3  8  13  18  23  28 )
std::cout<<"m1 * v = "<<m1*v<<std::endl;
//! m1 * v = 6  ( 216  396  432  60  162  108 )
std::cout<<"v * m1 = "<<v*m1<<std::endl;
//! v * m1 = 6  ( 202  492  780  832  396  -768 )

// Matrix * matrix product also expands bands appropriately:
tmv::BandMatrix<double> m1m2 = m1 * m2;
std::cout<<"m1 * m2 =\n"<<m1m2;
//! m1 * m2 =
//! 6  6
//! ( 488  592  400  136  0  12 )
//! ( 716  952  704  264  -8  -8 )
//! ( 528  948  904  272  -56  -32 )
//! ( 0  624  844  464  -320  -104 )
//! ( 0  0  520  452  -80  -332 )
//! ( 0  0  0  264  12  -96 )

// Can mix BandMatrix with other kinds of matrices:
std::cout<<"xm * m1 =\n"<<xm*m1;
//! xm * m1 =
//! 6  6
//! ( 82  48  -120  -348  -336  396 )
//! ( 252  318  180  -128  -276  216 )
//! ( 422  588  480  92  -216  36 )
//! ( 592  858  780  312  -156  -144 )
//! ( 762  1128  1080  532  -96  -324 )
//! ( 932  1398  1380  752  -36  -504 )
tmv::UpperTriMatrix<double> um(xm);
std::cout<<"um + m1 =\n"<<um+m1;
//! um + m1 =
//! 6  6
```

```
        //! ( 17  14  5  −6  −13  −22 )
        //! ( 20  25  16  1  −8  −17 )
        //! ( 0  24  27  12  −9  −12 )
        //! ( 0  0  24  23  2  −25 )
        //! ( 0  0  0  20  13  −14 )
        //! ( 0  0  0  0  12  −3 )
        tmv::LowerTriMatrix<double> lm(xm);
        lm *= m8;
        std::cout<<"lm *= m8 =\n"<<lm;
        //! lm *= m8 =
        //! 6  6
        //! ( 9  0  0  0  0  0 )
        //! ( 80  49  0  0  0  0 )
        //! ( 252  192  81  0  0  0 )
        //! ( 534  440  252  81  0  0 )
        //! ( 744  774  500  224  49  0 )
        //! ( 954  1064  782  396  120  9 )
        tmv::DiagMatrix<double> dm(xm);
        m1 *= dm;
        std::cout<<"m1 *= dm =\n"<<m1;
        //! m1 *= dm =
        //! 6  6
        //! ( 42  84  54  0  0  0 )
        //! ( 60  126  108  18  0  0 )
        //! ( 0  168  162  72  −42  0 )
        //! ( 0  0  216  126  0  −54 )
        //! ( 0  0  0  180  42  −36 )
        //! ( 0  0  0  0  84  −18 )
        return 0;
    } catch (tmv::Error& e) {
        std::cerr<<e<<std::endl;
        return 1;
    }
```

## 16.5 SymMatrix

```
#define TMV_EXTRA_DEBUG // To get the 888's below.

#include "TMV.h"
// Note: extra include file for SymMatrix
#include "TMV_Sym.h"
// Also need to link with -ltmv_symband
#include <iostream>

int main() try
{
    // Several ways to create and initialize matrices:

    // Create with uninitialized values
    tmv::SymMatrix<double> m1(5);
    // In debug mode, the elements are all set to 888 to make it easier
    // to find errors related to not correctly initializing the matrix.
    std::cout<<"m1 =\n"<<m1;
    //! m1 =
    //! 5  5
    //! ( 888  888  888  888  888 )
    //! ( 888  888  888  888  888 )
    //! ( 888  888  888  888  888 )
    //! ( 888  888  888  888  888 )
    //! ( 888  888  888  888  888 )

    // Initialize with element access:
    for(int i=0;i<m1.nrows();i++)
        for(int j=0;j<m1.ncols();j++)
            if (i>=j) m1(i,j) = 2.*i-j*j+2.;
    std::cout<<"m1 =\n"<<m1;
    //! m1 =
    //! 5  5
    //! ( 2  4  6  8  10 )
    //! ( 4  3  5  7  9 )
    //! ( 6  5  2  4  6 )
    //! ( 8  7  4  -1  1 )
    //! ( 10  9  6  1  -6 )

    tmv::SymMatrix<double> m2(5,2.); // Create with all 2's.
    std::cout<<"m2 =\n"<<m2;
    //! m2 =
    //! 5  5
    //! ( 2  2  2  2  2 )
    //! ( 2  2  2  2  2 )
    //! ( 2  2  2  2  2 )
    //! ( 2  2  2  2  2 )
    //! ( 2  2  2  2  2 )

    // Initialize from given elements:
    double mm[15] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15};
    tmv::SymMatrix<double,tmv::Upper|tmv::ColMajor> m3(5);
    // Use colmajor iterator of the upperTri portion:
    std::copy(mm,mm+15,m3.upperTri().colmajor_begin());
```

```
std::cout<<"m3 (Upper, ColMajor) =\n"<<m3;
//! m3 (Upper, ColMajor) =
//! 5  5
//! ( 1  2  4  7  11 )
//! ( 2  3  5  8  12 )
//! ( 4  5  6  9  13 )
//! ( 7  8  9  10  14 )
//! ( 11  12  13  14  15 )

// The iteration does not have to match the internal storage of
// the SymMatrix.  It will just be slightly less efficient.
tmv::SymMatrix<double,tmv::Lower|tmv::RowMajor> m4(5);
std::copy(mm,mm+15,m4.upperTri().rowmajor_begin());
std::cout<<"m4 (Lower,RowMajor) initialized with (Upper,RowMajor) =\n"<<m4;
//! m4 (Lower,RowMajor) initialized with (Upper,RowMajor) =
//! 5  5
//! ( 1  2  3  4  5 )
//! ( 2  6  7  8  9 )
//! ( 3  7  10  11  12 )
//! ( 4  8  11  13  14 )
//! ( 5  9  12  14  15 )

// Initialiion from comma separated list is not possible,
// but you can initialize either triangle:
tmv::SymMatrix<double> m5(5); // Default is Lower, ColMajor
m5.lowerTri() <<
    4,
    8, 10,
    1, -4, -7,
    9,  3,  3, 12,
    5,  0,  1,  8, 9;
std::cout<<"m5 =\n"<<m5;
//! m5 =
//! 5  5
//! ( 4  8  1  9  5 )
//! ( 8  10  -4  3  0 )
//! ( 1  -4  -7  3  1 )
//! ( 9  3  3  12  8 )
//! ( 5  0  1  8  9 )

tmv::SymMatrix<double> m6(5);
m6.upperTri() <<
    4,  8,  1,  9,  5,
       10, -4,  3,  0,
           -7,  3,  1,
               12,  8,
                    9;

std::cout<<"m6 =\n"<<m6;
//! m6 =
//! 5  5
//! ( 4  8  1  9  5 )
//! ( 8  10  -4  3  0 )
//! ( 1  -4  -7  3  1 )
//! ( 9  3  3  12  8 )
//! ( 5  0  1  8  9 )
```

```
// Make from the corresponding portion of a regular Matrix
// Note - it copies from the correct triangle regardless of
// the storage order of the two matrices.
tmv::Matrix<double,tmv::RowMajor> xm(5,5);
xm <<
     2, 5, 1, 9, 8,
     1, 5, 7, 2, 0,
     3, 9, 6, 8, 4,
     4, 2, 1, 9, 0,
     9, 8, 3, 7, 5;
tmv::SymMatrix<double,tmv::Lower|tmv::ColMajor> m7(xm);
std::cout<<"m7 =\n"<<m7;
//! m7 =
//! 5  5
//! ( 2  1  3  4  9 )
//! ( 1  5  9  2  8 )
//! ( 3  9  6  1  3 )
//! ( 4  2  1  9  7 )
//! ( 9  8  3  7  5 )


// Norms, etc.

std::cout<<"Norm1(m1) = "<<Norm1(m1)<<std::endl;
//! Norm1(m1) = 32
std::cout<<"Norm2(m1) = "<<Norm2(m1)<<std::endl;
//! Norm2(m1) = 24.4968
std::cout<<"NormInf(m1) = "<<NormInf(m1)<<std::endl;
//! NormInf(m1) = 32
std::cout<<"NormF(m1) = "<<NormF(m1);
std::cout<<" = "<<Norm(m1)<<std::endl;
//! NormF(m1) = 30.0333 = 30.0333
std::cout<<"MaxAbsElement(m1) = "<<MaxAbsElement(m1)<<std::endl;
//! MaxAbsElement(m1) = 10
std::cout<<"Trace(m1) = "<<Trace(m1)<<std::endl;
//! Trace(m1) = 0
std::cout<<"Det(m1) = "<<Det(m1)<<std::endl;
//! Det(m1) = 4866


// Views:

std::cout<<"m1 =\n"<<m1;
//! m1 =
//! 5  5
//! ( 2  4  6  8  10 )
//! ( 4  3  5  7  9 )
//! ( 6  5  2  4  6 )
//! ( 8  7  4  -1  1 )
//! ( 10  9  6  1  -6 )
std::cout<<"m1.diag() = "<<m1.diag()<<std::endl;
//! m1.diag() = 5  ( 2  3  2  -1  -6 )
std::cout<<"m1.diag(1) = "<<m1.diag(1)<<std::endl;
//! m1.diag(1) = 4  ( 4  5  4  1 )
std::cout<<"m1.diag(-1) = "<<m1.diag(-1)<<std::endl;
```

```
//! m1.diag(-1) = 4  ( 4  5  4  1 )
std::cout<<"m1.subSymMatrix(0,3) =\n"<<m1.subSymMatrix(0,3);
//! m1.subSymMatrix(0,3) =
//! 3  3
//! ( 2  4  6 )
//! ( 4  3  5 )
//! ( 6  5  2 )
std::cout<<"m1.upperTri() =\n"<<m1.upperTri();
//! m1.upperTri() =
//! 5  5
//! ( 2  4  6  8  10 )
//! ( 0  3  5  7  9 )
//! ( 0  0  2  4  6 )
//! ( 0  0  0  -1  1 )
//! ( 0  0  0  0  -6 )
std::cout<<"m1.lowerTri(tmv::UnitDiag) =\n"<<m1.lowerTri(tmv::UnitDiag);
//! m1.lowerTri(tmv::UnitDiag) =
//! 5  5
//! ( 1  0  0  0  0 )
//! ( 4  1  0  0  0 )
//! ( 6  5  1  0  0 )
//! ( 8  7  4  1  0 )
//! ( 10  9  6  1  1 )


// Fortran-style indexing:

tmv::SymMatrix<double,tmv::FortranStyle> fm1 = m1;
std::cout<<"fm1 = m1 =\n"<<fm1;
//! fm1 = m1 =
//! 5  5
//! ( 2  4  6  8  10 )
//! ( 4  3  5  7  9 )
//! ( 6  5  2  4  6 )
//! ( 8  7  4  -1  1 )
//! ( 10  9  6  1  -6 )
std::cout<<"fm1(1,1) = "<<fm1(1,1)<<std::endl;
//! fm1(1,1) = 2
std::cout<<"fm1(4,3) = "<<fm1(4,3)<<std::endl;
//! fm1(4,3) = 4
std::cout<<"fm1.subSymMatrix(1,3) =\n"<<fm1.subSymMatrix(1,3);
//! fm1.subSymMatrix(1,3) =
//! 3  3
//! ( 2  4  6 )
//! ( 4  3  5 )
//! ( 6  5  2 )


// Matrix arithmetic:

tmv::SymMatrix<double> m1pm3 = m1 + m3;
std::cout<<"m1 + m3 =\n"<<m1pm3;
//! m1 + m3 =
//! 5  5
//! ( 3  6  10  15  21 )
//! ( 6  6  10  15  21 )
//! ( 10  10  8  13  19 )
//! ( 15  15  13  9  15 )
//! ( 21  21  19  15  9 )

// Works correctly even if matrices are stored in different order:
tmv::SymMatrix<double> m3pm4 = m3 + m4;
std::cout<<"m3 + m4 =\n"<<m3pm4;
//! m3 + m4 =
//! 5  5
//! ( 2  4  7  11  16 )
//! ( 4  9  12  16  21 )
//! ( 7  12  16  20  25 )
//! ( 11  16  20  23  28 )
//! ( 16  21  25  28  30 )

tmv::SymMatrix<double> m4pm5 = m4 + m5;
std::cout<<"m4 + m5 =\n"<<m4pm5;
//! m4 + m5 =
//! 5  5
//! ( 5  10  4  13  10 )
//! ( 10  16  3  11  9 )
//! ( 4  3  3  14  13 )
//! ( 13  11  14  25  22 )
//! ( 10  9  13  22  24 )

m1 *= 2.;
std::cout<<"m1 *= 2 =\n"<<m1;
//! m1 *= 2 =
//! 5  5
//! ( 4  8  12  16  20 )
//! ( 8  6  10  14  18 )
//! ( 12  10  4  8  12 )
//! ( 16  14  8  -2  2 )
//! ( 20  18  12  2  -12 )

m1 += m4;
std::cout<<"m1 += m4 =\n"<<m1;
//! m1 += m4 =
//! 5  5
//! ( 5  10  15  20  25 )
//! ( 10  12  17  22  27 )
//! ( 15  17  14  19  24 )
//! ( 20  22  19  11  16 )
//! ( 25  27  24  16  3 )

// Vector outer product
tmv::Vector<double> v = xm.col(0);
tmv::SymMatrix<double> vv = v^v;
std::cout<<"v = "<<v<<std::endl;
//! v = 5  ( 2  1  3  4  9 )
std::cout<<"v^v =\n"<<vv;
//! v^v =
//! 5  5
//! ( 4  2  6  8  18 )
//! ( 2  1  3  4  9 )
//! ( 6  3  9  12  27 )
//! ( 8  4  12  16  36 )
//! ( 18  9  27  36  81 )

// SymMatrix * Vector product
std::cout<<"m1 * v = "<<m1*v<<std::endl;
//! m1 * v = 5  ( 370  414  381  307  240 )
std::cout<<"v * m1 = "<<v*m1<<std::endl;
// Note: This is only the same answer because m1 is symmetric.
//! v * m1 = 5  ( 370  414  381  307  240 )

// SymMatrix * Matrix product
tmv::Matrix<double> m1xm = m1 * xm;
std::cout<<"m1 * xm =\n"<<m1xm;
//! m1 * xm =
//! 5  5
//! ( 370  450  260  540  225 )
//! ( 414  523  299  637  283 )
//! ( 381  516  309  620  296 )
//! ( 307  531  347  587  316 )
//! ( 240  532  383  636  311 )

// Note: the product of two symmetrix matrices is not symmetric!:
tmv::Matrix<double> m1m5 = m1 * m5;
std::cout<<"m1 * m5 =\n"<<m1m5;
//! m1 * m5 =
//! 5  5
//! ( 420  140  -55  560  425 )
//! ( 486  198  -64  657  486 )
//! ( 501  291  -70  648  457 )
//! ( 454  337  -152  563  351 )
//! ( 499  422  -200  594  304 )


// Complex matrices:

tmv::SymMatrix<std::complex<double> > cm1 = m1 * std::complex<double>(1,2);
std::cout<<"cm1 = m1 * (1+2i) =\n"<<cm1;
//! cm1 = m1 * (1+2i) =
//! 5  5
//! ( (5,10)  (10,20)  (15,30)  (20,40)  (25,50) )
//! ( (10,20)  (12,24)  (17,34)  (22,44)  (27,54) )
//! ( (15,30)  (17,34)  (14,28)  (19,38)  (24,48) )
//! ( (20,40)  (22,44)  (19,38)  (11,22)  (16,32) )
//! ( (25,50)  (27,54)  (24,48)  (16,32)  (3,6) )
std::cout<<"cm1.conjugate() =\n"<<cm1.conjugate();
//! cm1.conjugate() =
//! 5  5
//! ( (5,-10)  (10,-20)  (15,-30)  (20,-40)  (25,-50) )
//! ( (10,-20)  (12,-24)  (17,-34)  (22,-44)  (27,-54) )
//! ( (15,-30)  (17,-34)  (14,-28)  (19,-38)  (24,-48) )
//! ( (20,-40)  (22,-44)  (19,-38)  (11,-22)  (16,-32) )
//! ( (25,-50)  (27,-54)  (24,-48)  (16,-32)  (3,-6) )
std::cout<<"cm1.transpose() =\n"<<cm1.transpose();
//! cm1.transpose() =
//! 5  5
//! ( (5,10)  (10,20)  (15,30)  (20,40)  (25,50) )
//! ( (10,20)  (12,24)  (17,34)  (22,44)  (27,54) )
//! ( (15,30)  (17,34)  (14,28)  (19,38)  (24,48) )
//! ( (20,40)  (22,44)  (19,38)  (11,22)  (16,32) )
//! ( (25,50)  (27,54)  (24,48)  (16,32)  (3,6) )
std::cout<<"cm1.adjoint() =\n"<<cm1.adjoint();
//! cm1.adjoint() =
//! 5  5
//! ( (5,-10)  (10,-20)  (15,-30)  (20,-40)  (25,-50) )
//! ( (10,-20)  (12,-24)  (17,-34)  (22,-44)  (27,-54) )
//! ( (15,-30)  (17,-34)  (14,-28)  (19,-38)  (24,-48) )
//! ( (20,-40)  (22,-44)  (19,-38)  (11,-22)  (16,-32) )
//! ( (25,-50)  (27,-54)  (24,-48)  (16,-32)  (3,-6) )
std::cout<<"cm1.realPart() =\n"<<cm1.realPart();
//! cm1.realPart() =
//! 5  5
//! ( 5  10  15  20  25 )
//! ( 10  12  17  22  27 )
//! ( 15  17  14  19  24 )
//! ( 20  22  19  11  16 )
//! ( 25  27  24  16  3 )
std::cout<<"cm1.imagPart() =\n"<<cm1.imagPart();
//! cm1.imagPart() =
//! 5  5
//! ( 10  20  30  40  50 )
//! ( 20  24  34  44  54 )
//! ( 30  34  28  38  48 )
//! ( 40  44  38  22  32 )
//! ( 50  54  48  32  6 )
std::cout<<"Norm(cm1) = "<<Norm(cm1)<<std::endl;
//! Norm(cm1) = 207.183

tmv::Vector<std::complex<double> > cv = v * std::complex<double>(-2,1);
cm1 += cv^cv;
std::cout<<"cm1 += cv^cv =\n"<<cm1;
//! cm1 += cv^cv =
//! 5  5
//! ( (17,-6)  (16,12)  (33,6)  (44,8)  (79,-22) )
//! ( (16,12)  (15,20)  (26,22)  (34,28)  (54,18) )
//! ( (33,6)  (26,22)  (41,-8)  (55,-10)  (105,-60) )
//! ( (44,8)  (34,28)  (55,-10)  (59,-42)  (124,-112) )
//! ( (79,-22)  (54,18)  (105,-60)  (124,-112)  (246,-318) )
cm1 += v^v;
std::cout<<"cm1 += v^v =\n"<<cm1;
//! cm1 += v^v =
//! 5  5
//! ( (21,-6)  (18,12)  (39,6)  (52,8)  (97,-22) )
//! ( (18,12)  (16,20)  (29,22)  (38,28)  (63,18) )
//! ( (39,6)  (29,22)  (50,-8)  (67,-10)  (132,-60) )
//! ( (52,8)  (38,28)  (67,-10)  (75,-42)  (160,-112) )
//! ( (97,-22)  (63,18)  (132,-60)  (160,-112)  (327,-318) )

tmv::Matrix<std::complex<double> > cx(5,2);
cx.col(0) = v;
cx.col(1) = cv;
cm1 -= cx*cx.transpose();
std::cout<<"cm1 -= cx*cx.transpose() =\n"<<cm1;
```

```
//! cm1 -= cx*cx.transpose() =
//! 5  5
//! ( (5,10)  (10,20)  (15,30)  (20,40)  (25,50) )
//! ( (10,20)  (12,24)  (17,34)  (22,44)  (27,54) )
//! ( (15,30)  (17,34)  (14,28)  (19,38)  (24,48) )
//! ( (20,40)  (22,44)  (19,38)  (11,22)  (16,32) )
//! ( (25,50)  (27,54)  (24,48)  (16,32)  (3,6) )

tmv::HermMatrix<std::complex<double> > cm2 = m1;
cm2.upperTri().offDiag() *= std::complex<double>(1,2);
std::cout<<"cm2 =\n"<<cm2;
//! cm2 =
//! 5  5
//! ( (5,-0)  (10,20)  (15,30)  (20,40)  (25,50) )
//! ( (10,-20)  (12,-0)  (17,34)  (22,44)  (27,54) )
//! ( (15,-30)  (17,-34)  (14,-0)  (19,38)  (24,48) )
//! ( (20,-40)  (22,-44)  (19,-38)  (11,-0)  (16,32) )
//! ( (25,-50)  (27,-54)  (24,-48)  (16,-32)  (3,-0) )
std::cout<<"cm2.conjugate() =\n"<<cm2.conjugate();
//! cm2.conjugate() =
//! 5  5
//! ( (5,0)  (10,-20)  (15,-30)  (20,-40)  (25,-50) )
//! ( (10,20)  (12,0)  (17,-34)  (22,-44)  (27,-54) )
//! ( (15,30)  (17,34)  (14,0)  (19,-38)  (24,-48) )
//! ( (20,40)  (22,44)  (19,38)  (11,0)  (16,-32) )
//! ( (25,50)  (27,54)  (24,48)  (16,32)  (3,0) )
std::cout<<"cm2.transpose() =\n"<<cm2.transpose();
//! cm2.transpose() =
//! 5  5
//! ( (5,-0)  (10,-20)  (15,-30)  (20,-40)  (25,-50) )
//! ( (10,20)  (12,-0)  (17,-34)  (22,-44)  (27,-54) )
//! ( (15,30)  (17,34)  (14,-0)  (19,-38)  (24,-48) )
//! ( (20,40)  (22,44)  (19,38)  (11,-0)  (16,-32) )
//! ( (25,50)  (27,54)  (24,48)  (16,32)  (3,-0) )
std::cout<<"cm2.adjoint() =\n"<<cm2.adjoint();
//! cm2.adjoint() =
//! 5  5
//! ( (5,0)  (10,20)  (15,30)  (20,40)  (25,50) )
//! ( (10,-20)  (12,0)  (17,34)  (22,44)  (27,54) )
//! ( (15,-30)  (17,-34)  (14,0)  (19,38)  (24,48) )
//! ( (20,-40)  (22,-44)  (19,-38)  (11,0)  (16,32) )
//! ( (25,-50)  (27,-54)  (24,-48)  (16,-32)  (3,0) )
std::cout<<"cm2.realPart() =\n"<<cm2.realPart();
//! cm2.realPart() =
//! 5  5
//! ( 5  10  15  20  25 )
//! ( 10  12  17  22  27 )
//! ( 15  17  14  19  24 )
//! ( 20  22  19  11  16 )
//! ( 25  27  24  16  3 )
// Note: imagPart is invalid for hermitian matrix, since the result
// would be anti-symmetric, which we don't have as a matrix type.
std::cout<<"Norm(cm2) = "<<Norm(cm2)<<std::endl;
//! Norm(cm2) = 202.349

cm2 += cv^cv.conjugate();
```

```
    std::cout<<"cm2 += cv^cv.conjugate() =\n"<<cm2;
    //! cm2 += cv^cv.conjugate() =
    //! 5  5
    //! ( (25,0)  (20,20)  (45,30)  (60,40)  (115,50) )
    //! ( (20,-20)  (17,0)  (32,34)  (42,44)  (72,54) )
    //! ( (45,-30)  (32,-34)  (59,0)  (79,38)  (159,48) )
    //! ( (60,-40)  (42,-44)  (79,-38)  (91,0)  (196,32) )
    //! ( (115,-50)  (72,-54)  (159,-48)  (196,-32)  (408,0) )
    cm2 += v^v;
    std::cout<<"cm2 += v^v =\n"<<cm2;
    //! cm2 += v^v =
    //! 5  5
    //! ( (29,0)  (22,20)  (51,30)  (68,40)  (133,50) )
    //! ( (22,-20)  (18,0)  (35,34)  (46,44)  (81,54) )
    //! ( (51,-30)  (35,-34)  (68,0)  (91,38)  (186,48) )
    //! ( (68,-40)  (46,-44)  (91,-38)  (107,0)  (232,32) )
    //! ( (133,-50)  (81,-54)  (186,-48)  (232,-32)  (489,0) )
    cm2 -= cx*cx.adjoint();
    std::cout<<"cm2 -= cx*cx.adjoint() =\n"<<cm2;
    //! cm2 -= cx*cx.adjoint() =
    //! 5  5
    //! ( (5,0)  (10,20)  (15,30)  (20,40)  (25,50) )
    //! ( (10,-20)  (12,0)  (17,34)  (22,44)  (27,54) )
    //! ( (15,-30)  (17,-34)  (14,0)  (19,38)  (24,48) )
    //! ( (20,-40)  (22,-44)  (19,-38)  (11,0)  (16,32) )
    //! ( (25,-50)  (27,-54)  (24,-48)  (16,-32)  (3,0) )


    // Can mix SymMatrix with other kinds of matrices:
    tmv::UpperTriMatrix<double> um(xm);
    std::cout<<"um + m1 =\n"<<um+m1;
    //! um + m1 =
    //! 5  5
    //! ( 7  15  16  29  33 )
    //! ( 10  17  24  24  27 )
    //! ( 15  17  20  27  28 )
    //! ( 20  22  19  20  16 )
    //! ( 25  27  24  16  8 )
    tmv::DiagMatrix<double> dm(xm);
    std::cout<<"dm * m1 =\n"<<dm*m1;
    //! dm * m1 =
    //! 5  5
    //! ( 10  20  30  40  50 )
    //! ( 50  60  85  110  135 )
    //! ( 90  102  84  114  144 )
    //! ( 180  198  171  99  144 )
    //! ( 125  135  120  80  15 )

    return 0;
} catch (tmv::Error& e) {
    std::cerr<<e<<std::endl;
    return 1;
}
```

# 17 Known bugs and deficiencies

## 17.1 Known compiler issues

I have tested the code using the following compilers:

GNU's g++ – versions 3.4.6, 4.1.2, 4.3.2, 4.4.7, 4.5.3, 4.6.2, 4.8.2
Apple's g++ – version 4.0.1, 4.2.1
Intel's icpc – versions 10.1, 11.1, 12.0.4, 14.0.1
Portland's pgCC – version 6.1
LLVM's clang++ – version 3.1
Apple's clang++ (erroneously named g++) – versions 425, 500
Microsoft's cl – Visual C++ 2008 Express Edition

It should work with any ansi-compliant compiler, but no guarantees if you use one other than these[21]. So if you do try to compile on a different compiler, I would appreciate it if you could let me know whether you were successful. Please email the TMV discussion group at `tmv-discuss@googlegroups.com` with your experience (good or bad) using compilers other than these.

There are a few issues that I have discovered when compiling with various versions of compilers, and I have usually come up with a work-around for the problems. So if you have a problem, check this list to see if a solution is given for you.

- **g++ versions 4.4 and 4.5 on Snow Leopard:** g++ 4.4 and 4.5 (at least 4.4.2 through 4.5.0) have a pretty egregious bug in their exception handling that can cause the code to abort rather than allow a thrown value to be caught. This is a reported bug (42159), and according to the bug report it seems to only occur in conjunction with MacOS 10.6 (Snow Leopard), however it is certainly possible that it may show up in other systems too. (I don't have Lion yet, so I haven't tested it with that.)

  However, I have discovered a workaround that seems to fix the problem. Link with `-lpthread` even if you are not using OpenMP or have any other reason to use that library. Something about linking with that library fixes the exception handling problems. Anyway, this is mostly to explain why TMV adds `-lpthread` to the recommended linking instructions for these systems even if you disable OpenMP support.

- **g++ -O2, versions 4.1 and 4.2:** It seems that there is some problem with the -O2 optimization of g++ versions 4.1.2 and 4.2.2 when used with TMV debugging turned on. Everything compiles fine when I use `g++ -O` or `g++ -O2 -DNDEBUG`. But when I compile with `g++ -O2` (or `-O3`) without `-DNDEBUG`, then the test suite fails, getting weird results for some arithmetic operations that look like uninitialized memory was used.

  I distilled the code down to a small code snippet that still failed and sent it to Gnu as a bug report. They confirmed the bug and suggested using the flag `-fno-strict-aliasing`, which did fix the problems.

  Another option, which might be a good idea anyway is to just use `-O1` when you want a version that includes the TMV assert statements, and make sure to use `-DNDEBUG` when you want a more optimized version.

- **Apple g++:** Older versions of Apple's version of g++ that they shipped with the Tiger OS did not work for compilation of the TMV library. It was called version 4.0, but I do not remember the build number. They seem to have fixed the problem with later XCode update, but if you have an older Mac and want to compile TMV on it and the native g++ is giving you trouble, you should either upgrade to a newer Xcode distribution or download the real GNU gcc instead; I recommend using Fink (`http://fink.sourceforge.net/`).

[21] It does seem to be the case that every time I try the code on a new compiler, there is some issue that needs to be addressed. Either because the compiler fails to support some aspect of the C++ standard, or they enforce an aspect that I have failed to strictly conform to. Or sometimes even because of a bug in the compiler.

- **pgCC:** I only have access to pgCC version 6.1, so these notes only refer to that version.
  - Apparently pgCC does not by default support exceptions when compiled with openmp turned on. So if you want to use the parallel versions of the algorithms, you need to compile with the flags `-mp --exceptions`. This is a documented feature, but it's not very obvious, so I figured I'd point it out here.
  - There was a bug in pgCC version 6.1 that was apparently fixed in version 7.0 where `long double` variables were not correctly written with `std::ostream`. The values were written as either `0` or `-`. So I have written a workaround in the code for pgCC versions before version 7.0[22], where the `long double` values are copied to `double` before writing. This works, but only for values that are so convertible. If you have values that are outside the range representable by a `double`, then you may experience overflow or underflow on output.

- **Borland's C++ Builder:** I tried to compile the library with Borland's C++ Builder for Microsoft Windows Version 10.0.2288.42451 Update 2, but it failed at fairly foundational aspects of the code, so I do not think it is possible to get the code to work. However, if somebody wants to try to get the code running with this compiler or some other Borland product, I welcome the effort and would love to hear about a successful compilation (at `tmv-discuss@googlegroups.com`).

- **Solaris Studio:** I also tried to compile the library with Sun's Solaris Studio 12.2, which includes version 5.11 of CC, its C++ compiler. I managed to get the main library to compile, but the test suite wouldn't compile. The compiler crashed with a "Signal 11" error. (Actually tmvtest1 and tmvtest3a, 3d and 3e worked. But not the others.) I messed around with it for a while, but couldn't figure out a workaround. However, it may be the case that there is something wrong with my installation of CC, since it's not something I really use, and I installed it on a non-Sun machine, so who knows how reliable that is. Since I couldn't get it working completely, I'm not willing to include this on my above list of "supported" compilers, but if you use this compiler regularly and want to vet the TMV code for me, I would appreciate hearing about your efforts at `tmv-discuss@googlegroups.com`.

- **Memory requirements:** The library is pretty big, so it can take quite a lot of memory to compile. For most compilers, it seems that a minimum of around 512K is required. For compiling the test suite with the `XTEST` flag set to something other than 0 (especially something that combines a lot of multiple extra tests, e.g. XTEST=127), at least around 4 GB of memory is recommended.

- **Linker choking:** Some linkers (e.g. the one on my old Mac G5) have trouble with the size of some of the test suite's executables, especially when compiled with `XTEST` set to include lots of extra tests. If you encounter this problem, you can instead compile the smaller test suites.

  The tests in `tmvtest1` are split into `tmvtest1a`, `tmvtest1b` and `tmvtest1c`. Likewise `tmvtest2` has `a`, `b` and `c` versions, and `tmvtest3` has `a`, `b`, `c` and `d` versions. These are compiled by typing `scons tests SMALL_TESTS=true`. Or you can make them one at a time by typing `scons test1a`, `scons test1b`, etc.

  You might also try testing only one type at a time: First compile with `INST_FLOAT=false` and then `INST_FLOAT=true INST_DOUBLE=false`. This cuts the size of the executables in half, which can also help if the above trick doesn't work. (I had to do this for test2c on one of my test systems that does not have much memory.)

- **Non-portable IsNaN():** The LAPACK function `?stegr` sometimes produces `nan` values on output. Apparently there is a bug in at least some distributions of this function. Anyway, TMV checks for this and calls

[22] Thanks to Dan Bonachea for making available his file `portable_platform.h`, which makes it particularly easy to test for particular compiler versions.

the alternative (but slower) function `?stedc` instead whenever a `nan` is found. The problem is that there is no C++ standard way to check for a `nan` value.

The usual way is to use a macro `isnan`, which is usually defined in the file `<math.h>`. However, this is technically a C99 extension, not standard C++. So if this macro is not defined, then TMV tries two other tests that usually detect `nan` correctly. But if this doesn't work correctly for you, then you may need to edit the file `src/TMV_IsNaN.cpp` to work with your system[23].

Note that this can only ever be an issue if you specifically request that TMV use the `stegr` algorithm with the SCons option `USE_STEGR=true`. If you do not do this, then TMV doesn't use the LAPACK `?stegr` functions at all, so it's not a problem.

## 17.2 Known problems with BLAS or LAPACK libraries:

There are a number of possible errors that are related to particular BLAS or LAPACK distributions, or combinations thereof:

- **Strict QRP decomposition fails:** Some versions of the LAPACK function `dgeqp3` do not produce the correct result for $R$. The diagonal elements of $R$ are supposed to be monotonically decreasing along the diagonal, but sometimes this is not strictly true. This probably varies among implementations, so your version might always succeed.

  This would only happen if you install TMV with the SCons option `USE_GEQP3=true`, to use the LAPACK function (which is called `?geqp3` rather than the native TMV function. So the solution is to use `USE_GEQP3=false` instead.

- **Info $<$ 0 errors:** If you get an error that looks something like:
  ```
  TMV Error:  info < 0 returned by LAPACK function dormqr
  ```
  then this probably means that your LAPACK distribution does not support workspace size queries. The solution is to use the flag `-DNOWORKQUERY`. I think TMV usually knows about these cases now and checks for them, but you might have a LAPACK distribution that I'm not aware of. So you can add this flag by hand with the SCons option `EXTRA_FLAGS=-DNOWORKQUERY`. If that doesn't fix your problem, please file a bug report at `https://github.com/rmjarvis/tmv/issues`, since that means it's probably something else.

- **Unable to link LAPACK on Mac:** I get a linking error on my Mac when combining the XCode LAPACK library (either libclapack.dylib or liblapack.dylib) with a non-XCode BLAS library. Specifically, it can't find the `?getri` functions. Basically, the reason is that the XCode LAPACK libraries are designed to be used with one of the BLAS libraries, libcblas.dylib or libblas.dylib, in the same directory, and that BLAS library has the `getri` functions, even though they are properly LAPACK functions. Anyway, it's basically a bug in the Apple XCode distribution of these files, and the result is that if you use a different BLAS library, the linking fails.

  The solution is either to use the XCode BLAS library or to install your own CLAPACK library. If you do the latter, you will probably want to rename (or delete) these Mac library files, since they are in the /usr/lib directory, and `-L` flags usually can't take precedence over /usr/lib.

- **Errors in `Matrix<float>` calculations using BLAS on Mac:** I have found that some XCode BLAS libraries seem to have errors in the calculations for large `<float>` matrices. I think it is an error in the `sgemm` function. Since very many of the other algorithms use this function, the error propagates to lots of other routines as well. The errors seem to be only for large matrices with at least one dimension $N > 100$ or so. (I'm not sure of the exact crossover point from working to non-working code.)

[23] However, the provided code did work successfully on all the compilers I tested it on, so technically this is not a "known" compiler issue, just a potential issue.

Apparently, this is a known bug (ID 7437011), but I don't know if it is fixed in the latest XCode 4 for the Lion OS (I don't yet have a computer with Lion on it). But the bug is at least present in XCode versions 3.2.2 through 3.2.6. The best thing to do to see if your XCode installation has this problem is to install the test suite and make sure that it runs without errors. If it makes it through without errors, then you don't have to worry about it.

If you encounter this problem, which manifests as `tmvtest1` failing in the `<float>` section, then I'm afraid the possible solutions are to forego using `Matrix<float>`, switch to another BLAS library, or compile TMV without BLAS (with SCons option `USE_BLAS=false`).

- **Overflow and underflow when using LAPACK:** The normal distribution of LAPACK algorithms are not as careful as the TMV code when it comes to guarding against overflow and underflow. As a result, you may find matrix decompositions resulting in values with `nan` or `inf` when using TMV with LAPACK support enabled. The solution if this is a problem for you is to compile TMV without the LAPACK library (using the SCons option `WITH_LAPACK=false`). This does not generally result in slower code, since the native TMV code is almost always as fast (usually using the same algorithm) as the LAPACK distribution, but has better guards against overflow and underflow.
- **Segmentation fault or other errors when using LAPACK in multi-threaded program:** I have found that many LAPACK libraries (especially older installations) are not thread-safe. So you might get segmentation faults or other strange non-repeatable errors at random times when using LAPACK within a multi-threaded program. The solution is to compile TMV without LAPACK (with SCons option `WITH_LAPACK=false`).
- **LAPACK fails for extremely large matrices:** For matrices that use more than 2 GBytes of memory, the integers that index the memory addresses need to be more that $2^{31}$. This number exceeds the capacity of a 32 bit (signed) integer. So the TMV library uses `ptrdiff_t` for all integers that might be used in calculating such an index to make sure it is always safe. However, LAPACK distributions generally use `int`. If this happens to be 32 bits on your machine, it can overflow. This is particularly a problem for the LAPACK SVD routine, since it needs workspace of size a bit more than $4n^2$. So if $4n^2$ is more than $2^{31}$, it will fail. Again, the solution is to compile TMV without LAPACK (with SCons option `WITH_LAPACK=false`).

## 17.3 To-do list

Here is a list of various deficiencies with the current version of the TMV library. These are mostly features that are not yet included, rather than bugs per se.

If you find something to add to this list, or if you want me to bump something to the top of the list, let me know. Not that the list is currently in any kind of priority order, but you know what I mean. Please post a feature request at `https://github.com/rmjarvis/tmv/issues`, or email the discussion group about what you need at `tmv-discuss@googlegroups.com`.

1. **Symmetric arithmetic**

   When writing complicated equations involving complex symmetric or hermitian matrices, you may find that an equation that seems perfectly ok does not compile. The reason for this problem is explained in §9.4 in some detail, so you should read about it there. But basically, the workaround is usually to break your equation up into smaller steps that do not require the code to explicitly instantiate any matrices. For example: (this is the example from §9.4)

   ```
   s3 += x*s1 + s2;
   ```

   will not compile if `s1`, `s2`, and `s3` are all complex symmetric, even though it is valid, mathematically. Rewriting this as:

   ```
   s3 += x*s1;
   s3 += s2;
   ```

will compile and work correctly. This bug will be fixed in version 0.90, which is currently in development.

2. **Eigenvalues and eigenvectors**

   The code only finds eigenvalues and eigenvectors for hermitian matrices. I need to add the non-hermitian routines.

3. **More special matrix varieties**

   Block-diagonal, generic sparse (probably both row-based and column-based), block sparse, symmetric and hermitian block diagonal, etc. Maybe skew-symmetric and skew-hermitian. Are these worth adding? Let me know.

4. **Packed storage**

   Triangular and symmetric matrices. can be stored in (approximately) half the memory as a full $N \times N$ matrix using what is known as packed storage. There are BLAS routines for dealing with these packed storage matrices, but I don't yet have the ability to create/use such matrices.

5. **Hermitian eigenvector algorithm**

   There is a faster algorithm for calculating eigenvectors of a hermitian matrix given the eigenvalues, which uses a technique know as a "Relatively Robust Representation". The native TMV code does not use this, so it is slower than a compilation which calls the LAPACK routine (which is only enabled if you use both `WITH_LAPACK=true` and `USE_STEGR=true`). I think this is the only routine for which the LAPACK version is still significantly faster than the native TMV code. Although, I do find that many LAPACK distributions have a fairly buggy version of the algorithm that does not deal very well with overflow and underflow, which is why it must be specifically enabled. Hopefully when I implement this in TMV, I can do a better job in this regard.

6. **Row-major Bunch-Kaufman**

   The Bunch-Kaufman decomposition for row-major symmetric/hermitian matrices is currently $LDL^\dagger$, rather than $L^\dagger DL$. The latter should be somewhat (30%?) faster. The current $LDL^\dagger$ algorithm is the faster algorithm for column-major matrices.[24]

7. **Conditions**

   Currently, SVD is the only decomposition that calculates the condition of a matrix (specifically, the 2-condition). LAPACK has routines to calculate the 1- and infinity-condition from an LU decomposition (and others). I should add a similar capability.

8. **Division error estimates**

   LAPACK provides these. It would be nice to add something along the same lines.

9. **Equilibrate matrices**

   LAPACK can equilibrate matrices before division. Again, I should include this feature too. Probably as an option (since most matrices don't need it) as something like `m.Equilibrate()` before calling a division routine.

10. **OpenMP in the SVD algorithm**

    TMV's implementation of the divide-and-conquer SVD algorithm only uses half of the potential for parallelization at the moment. I need to reorganize the algorithm a bit to make it more amenable to being further parallelized, but it is certainly doable.

[24] These comments hold when the storage of the symmetric matrix is in the lower triangle - it is the opposite for storage in the upper triangle.

11. **Check for memory throws**

    Many algorithms are able to increase their speed by allocating extra workspace. Usually this workspace is significantly smaller than the matrix being worked on, so we assume there is enough space for these allocations. However, I should add try-catch blocks to catch any out-of-memory throws and use a less memory-intesive algorithm when necessary.

    Version 0.90 will include these checks.

12. **Integer determinants for special matrices**

    The determinant of `<int>` matrices is only written for a regular `Matrix`, and of course for the trivial `DiagMatrix` and `TriMatrix` types. But for `BandMatrix`, `SymMatrix`, and `SymBandMatrix`, I just copy to a regular `Matrix` and then calculate the determinant of that. I can speed up the calculation for these special matrix types by taking advantage of their special structure, even using the same Bareiss algorithm as I currently use.

## 17.4 Reporting bugs

If you find a bug in the TMV library that is not listed above, please let me know. The preferred way to report a bug is to enter a ticket at `https://github.com/rmjarvis/tmv/issues`. You will need to have a Google account, but they are free, so I hope that isn't too much of a hardship. Click "New Issue". If you aren't logged into your Google account, you'll be asked to log in. This will bring up a form for you to fill out.

There are two templates for you to choose from: "Defect report from user" and "Feature request". In both cases, you should first enter a summary. Then there is a text box where you can provide more information.

For a defect report, there are three questions that you should answer to help me understand the problem you are having. They are pretty self-explanatory, so just do your best to answer them as completely as possible. Err on the side of providing too much information rather than too little to help me figure out how to reproduce the error. And if appropriate, please also attach the code you are using, preferably distilled down to as small a program as possible, that produces the error.

A feature request is more open-ended. Just describe what you would like to see in a future version of TMV in as much detail as you feel is appropriate.

Another option, especially if you aren't sure if what you are seeing is really a bug or not and you want some more informal feedback, is to email the TMV discussion email list at `tmv-discuss@googlegroups.com`. Either I or someone else on the list may be able to help you figure out the problem.

# 18 History

Here is a list of the changes from version to version. Whenever a change is not backward compatible, meaning that code using the previous version might be broken, I mark the item with a × bullet rather than the usual • to indicate this. Also, the bulleted lists are not comprehensive. In most cases, new versions fix minor bugs that I find in the old version. I only list the more significant changes.

**Version 0.1** The first matrix/vector library I wrote. It wasn't very good, really. It had a lot of the functionality I needed, like mixing complex/real, SV decomposition, LU decomposition, etc. But it wasn't at all fast for large matrices. It didn't call BLAS or LAPACK, nor were the native routines very well optimized. Also, while it had vector views for rows and columns, it didn't have matrix views for things like transpose. Nor did it have any delayed arithmetic evaluation. And there were no special matrices.

I didn't actually name this one 0.1 until I had what I called version 0.3.

**Version 0.2** This was also not named version 0.2 until after the fact. It had most of the current interface for regular Matrix and Vector operations. I added Upper/Lower TriMatrix and DiagMatrix. It also had matrix views and matrix composites to delay arithmetic evaluation. The main problem was that it was still slow. I hadn't included any BLAS calls yet. And while the internal routines at least used algorithms that used unit strides whenever possible, they didn't do any blocking or recursion, which are key for large matrices.

**Version 0.3** Finally, I actually named this one 0.3 at the time. The big addition here was BLAS and LAPACK calls, which helped me to realize how slow my internal code really was (although I hadn't updated them to block or recursive algorithms yet). I also added BandMatrix.

**Version 0.4** The new version number here was because I needed some added functionality for a project I was working on. It retrospect, it really only deserves a 0.01 increment, since the changes weren't that big. But, oh well.

- Added QR_Downdate. (This was the main new functionality I needed.)
- Improved the numerical accuracy of the QRP decomposition.
- Added the possibility of not storing U,V for the SVD.
- Greatly improved the test suite, and consequently found and corrected a few bugs.
- Added some arithmetic functionality that had been missing (like `m += L*U`).

**Version 0.5** The new symmetric matrix classes precipitated a major version number update. I also sped up a lot of the algorithms:

- Added SymMatrix, HermMatrix, and all associated functionality.
- Added blocked versions of most of the algorithms, so the non-LAPACK code runs a lot faster.
- Allowed for loose QRP decomposition.
- Added divideInPlace().

**Version 0.51** Some minor improvements:

- Sped up some functions like matrix addition and assignment by adding the LinearView method.
- Added QR_Update, and improved the QR_Downdate algorithm.
- Blocked some more algorithms like TriMatrix multiplication/division, so non-BLAS code runs significantly faster (but still much slower than BLAS).

**Version 0.52** The first "public" release! And correspondingly, the first with documentation and a web site. A few other people had used previous versions, but since the only documentation was my comments in the .h files, it wasn't all that user-friendly.

- Added saveDiv() and related methods for division control. Also changed the default behavior from saving the decomposition to not saving it.
- Added in-place versions of the algorithms for $S = L^\dagger L$ and $S = LL^\dagger$.

**Version 0.53** By popular demand (well, a polite request by Fritz Stabenau, at least):

- Added the Fortran-style indexing.

**Version 0.54** Inspired by my to-do list, which I wrote for Version 0.52, I tackled a few of the items on the list and addressed some issues that people had been having with compiling:

× Changed from a rudimentary exception scheme (with just one class - `tmv_exception`) to the current more full-featured exception hierarchy. Also added `auto_ptr` usage instead of bald pointers to make the exception throws memory-leak safe.

- Sped up SymLUDiv and SymSVDiv inverses.
- Added the possibility of compiling a small version of the library and test suite.

× Consolidated SymLUDiv and HermLUDiv classes into just SymLUDiv, which now checks whether the matrix is hermitian automatically.

- Reduced the number of operations that make temporary matrices when multiple objects use the same storage.
- Specialized Tridiagonal × Matrix calculation.
- Added ElementProd and AddElementProd functions for matrices.
- Added CLAPACK and ACML as known versions of LAPACK.

**Version 0.60** This revision merits a first-decimal-place increment, since I added a few big features. I also registered it with SourceForge, which is a pretty big milestone as well.

- Added `SmallVector` and `SmallMatrix` with all accompanying algorithms.
- Added `SymBandMatrix` and `HermBandMatrix` with all accompanying algorithms.
- Made arithmetic between any two special matrices compatible, so long as the operation is allowed given their respective shapes.

× Changed `QR_Downdate()` to throw an exception rather than return false when it fails.

- Added the GPL License to the end of this documentation, and a copyright and GPL snippet into each file.

× Changed the -D compiler options for changing which types get instantiated.

- Split up the library into `libtmv.a` and `libtmv_symband.a`.

**Version 0.61** A number of updates mostly precipitated by feature requests by me in my own use of the library, as well as some from a few other users. I also did a complete systematic edit of the documentation which precipitated some more changes to make the UI a bit more intuitive.

× Changed the default storage for the `Matrix` class to `ColMajor` rather than `RowMajor`.

× Changed a lot of `size_t` parameters to `int`.

× Removed `U.MakeUnitDiag`.

- Sped up matrix multiplication for non-blas implementations, including openmp pragmas to allow for multiple threads on machines that support them.
- Changed a few things which prevented Microsoft Visual C++ from compiling successfully. Thanks to Andy Molloy for spearheading this effort and doing the lion's share of the work to make the code compatible with the VC++ compiler.

× Removed the optional index parameter to the non-method versions of MaxElement, etc.

- Added an optional `scale` parameter to `m.NormSq(scale)`.

× Added the explicit decomposition routines. I also got rid of the `SVU`, `SVV` and `SVS` options for `m.divideUsing(...)`, since the point of these was to do the decomposition without calculation $U$ and/or $V$. This is now done more intuitively with the explicit decomposition routines. I also added the (hermitian) eigenvalue/eigenvector routines which used to require using the division accessors in non-intuitive ways to calculate.

- Fixed a couple of places where underflow and overflow could cause problems.
- Updated the native TMV code for the singular value decomposition and hermitian eigenvalue calculation to use the divide-and-conquer algorithm.
- Added `m.logDet()` method.

× Changed `m.svd().getS()` to return a `DiagMatrix` rather than a `Vector`.

**Version 0.62** This release contains a number of bug fixes and corrections of mistakes in the documentation. I also significantly revamped the `SmallMatrix` class to make it faster at the expense of a few fewer features.

- Corrected an error with `m.divIsSet()`.
- Corrected some errors in the documentation – Thanks to Jake VanderPlas for pointing these out.
- Improved the behavior of the division accessors, also at the suggestion of Jake VanderPlas. (Thanks again!) They used to require that the division object had already been set or they would give an error. Now the accessors (like `m.svd()`) set the division object appropriately if it is not already set.
- Added the `ListInit` method for initializing the values of a `Vector` or `Matrix`. Thanks to Paul Sarli for this suggestion.

× Significantly changed the `SmallMatrix` class. See §5.10, on the `SmallMatrix` class in the documentation for the full details of how it works now. But in brief, here are the main changes:

  – `SmallMatrix` and `SmallVector` no longer inherit from `GenMatrix` and `GenVector`.
  – This allowed us to remove all virtual functions and the corresponding `vtable`.
  – Improved the arithmetic so more of the routines correctly do the calculation inline to allow the compiler more opportunity to optimize the calculation.
  – Added inlined division and determinants.
  – Got rid of the "Small" views.

- Consolidated some of the old header files.

× Removed some of the `ViewOf` commands that were not very clear and which have other, clearer ways of doing the same thing:

  – `d = DiagMatrixViewOf(m)` should now be written
    `d = DiagMatrixViewOf(m.diag())`.
  – `U = UpperTriMatrixViewOf(m)` should now be written
    `U = m.upperTri()`.

    - `U = UpperTriMatrixViewOf(m,UnitDiag)` should now be written `U = m.upperTri(UnitDiag)`.
    - `U = UpperTriMatrixViewOf(U,UnitDiag)` should now be written `U = U.viewAsUnitDiag()`.

- Tracked down the problems I had been having with the LAPACK `dstegr` and `sstegr` functions. TMV now checks for some known problems with the LAPACK implementation of `stegr` sometimes failing, and it calls `stedc` if there was a problem.
- Tested the code for memory bugs with Paul Nettle's mmgr.h code. There were only a couple of minor memory leaks, which were fixed.
- Fixed a problem with the OpenMP version of the code giving segmentation faults with pgCC.
- Added the SCons installation method to automatically discover what BLAS and LAPACK libraries are installed on your computer.
- Added compatibility for generic Fortran versions of BLAS and LAPACK.
- Added the CMake installation method. Thanks to Andy Molloy for providing this.
- Added hyperlinks to the PDF documentation. Also, I added an index, complete with hyperlinks as well, that should make it easier to find a particular topic that you might need help with.

**Version 0.63** The biggest thing in this version is the new lowercase syntax for the methods, which was in response to feedback from a number of TMV users, who didn't like the old syntax. Hopefully, I'm not responding to the minority here, but I do agree that the new syntax conforms better to common C++ standards. There are also a couple of bug fixes, and I've started using Google Code for my main website.

× Changed the names of the methods to start with a lowercase. The free functions all still start with a capital letter. Both use camelCase, with underscores after 2 or 3 letter abbreviations such as LU or SV to help them stand out (since the camel case doesn't do it). This seems to be a more common standard in the C++ community and is more self-consistent than the style I had been using.

× Changed the syntax for list initialization to not use `tmv::ListInit`.

- Improved the speed of the native (i.e. non-BLAS) matrix multiplication algorithm.
- Fixed a couple of bugs involving `SmallMatrix`.
- Disabled the cases where `BlasRowMajor` was used in CBlas implementations, since I had problems with it on a particular BLAS version, and I'm not sure if the error is in my code or in that BLAS implementation. So, I removed these pathways for now until I can figure out what might be going on with it. Most calls were with `BlasColMajor` anyway, so this doesn't affect very many calls.
- Fixed the return type of some methods that erroneously returned `CStyle` views when the source object was `FortranStyle`.
- Added a way to access the version of TMV being used. You can do this either in C++ code with the function `TMV_Version()`. This returns a string that gives the version being used within your code. There is also an executable, `tmv-version`, that can be used to access the TMV version that is installed at the moment.
- Added support for `ups`, which is probably completely irrelevant to the majority of users. But if you use `ups` for version control, you can use the command `scons install WITH_UPS=true` to configure tmv in `ups`.

**Version 0.64** This update mostly addresses problems involving underflow and overflow. This had been item 13 on the to-do list (§17), and it was bumped to top priority from a bug report where underflow problems were leading to an infinite loop in the singular value decomposition code. So I added more matrices to the test suite – specifically a matrix with an extremely large dynamic range of singular values (which reproduced the behavior of the bug report), a matrix that is nearly zero, and a matrix that is very close to overflow. These tests uncovered quite a few bugs in the code. Some from overflow and underflow issues of course, but also some imprecision issues, and some just plain bugs that I hadn't uncovered before. So I think this version should be much more stable for a wider range of input matrices.

There are also a bunch of miscellaneous feature additions as well – most notably the `Permutation` class.

- Added a some very difficult matrices to the test suite, and found and fixed quite a few bugs as a result involving underflow, overflow, and loss of precision.
- Fixed a bug that `U(i,i)` was invalid if `U` is non-const and is `UnitDiag` even if the resulting value is not used in a mutable context.
- Added a new `Permutation` class to encapsulate permutation operations.

× The `getP()` methods from the LU and QRP divider objects now return a `Permutation` rather than a `const int*`.

- Added `m.sumElements()` and `m.sumAbsElements()`.
- Added `v.minAbs2Element()`, `v.maxAbs2Element()` and `m.maxAbs2Element()`.
- Added `m.unitUpperTri()` and `m.unitLowerTri()`.
- Added `b.subBandMatrix(i1,i2,j1,j2)` to `BandMatrix` and `SymBandMatrix`.
- Made `Swap(m1,m2)` and `Swap(v1,v2)` efficient when the arguments are both complete matrices or vectors, rather than views.
- Added `cView()` and `fView()` methods to switch indexing styles of a matrix or vector.
- Made arithmetic with `SmallVector` and `SmallMatrix` somewhat more flexible in that the composite objects now (again actually) derive from the normal `GenVector` and `GenMatrix` classes.
- Added a version of `MatrixViewOf` with arbitrary steps.
- Added the three C preprocessor definitions: `TMV_MAJOR_VERSION`, `TMV_MINOR_VERSION`, and `TMV_VERSION_AT_LEAST(major,minor)` to help users customize their code depending on the version of TMV that is installed on a particular computer.
- Fixed a bug in BLAS versions of U*M and L*M when U or L is real and M is complex.
- Fixed a bug in `m.det()` when `m` is a `SmallMatrix<T,1,1>`.
- Fixed a bug in an (apparently) uncommon pathway of `Rank2Update`.
- Researched my `QRDowndate` algorithm to see if it is in the literature already. The only similar algorithm I could find was in Bojanczyk and Steinhardt (1991), a paper which seems to have been overlooked by the matrix algorithms community. It's not quite the same as mine in detail, but it is based on the same basic idea. I added this reference to the discussion in the documentation about my algorithm.

**Version 0.65** This update primarily fixed some problems for integer matrices. In addition to some compiler errors for some combinations of SCons parameters, I also added the ability to calculate the determinant of integer matrices. It had been documented (albeit somewhat poorly) that this would not work for integer matrices. However, user Ronny Bergmann convinced me that it would make sense to include the ability to produce accurate determinants for integer matrices.

- Made `m.det()` produce accurate results if `m` is a `Matrix<int>` or `Matrix<complex<int> >`. Or in general if the underlying type is an integer type.
- Added `m.sumAbs2Elements()`.

- Added `m.addToAll(x)`.
- Added `m.unitUpperTri()` and `m.unitLowerTri()` for symmetric/hermitian matrices.
- Added the `resize` methods to resize an existing vector or matrix, at the request of users Zhaoming Ma and Gary Bernstein.
- Added SCons option `SHARED=true` to compile as a shared library rather than a static library.
- Added the ability to turn off TMV debug statements with `-DTMVNDEBUG`.
- Skip compiling functions that are invalid for integer values, like the various decompositions and such.
- Fixed some compiling errors with `INST_INT=true`, including some linking errors for several combinations of compile options.
- Fixed a couple bugs in the allocation code.
- Increased the maximum number of iterations within the divide-and-conquer SVD algorithm from 20 to 100, since I found a case that required 27 iterations to converge.
- Fixed some places where nan's in the input values could lead to infinite loops in certain algorithms.
- Fixed a place in the SV decomposition which could (rarely) go into an infinite loop.
- Made code more robust to broken BLAS distributions that incorrectly multiply the output matrix or vector by the `beta` variable even if it is zero. This is a problem if there are `nan`'s present in the memory, since multiplies by zero doesn't set the value to zero in that case. Now we always check for `beta == 0` and zero out the memory if appropriate.

**Version 0.70** The impetus for this release was mostly just the first change listed below about the behavior of comma initialization for matrices. But since I knew this change would not be backwards compatible, and might require users to edit their code, I decided to try to put all the non-backward-compatible changes that I've been thinking about introducing in the near future into this release to try to do everything at once.

× Changed the behavior of the `m << 1, 2, 3...` initialization. Now the items in the initialization list should be in row-major order regardless of the storage order of the matrix `m`.

× Removed the constructors from either a C-style array or a `std::vector`. To take the place of these constructors' functionality, I added iterators that iterate over the matrix in either row-major or column-major order, so the assignment can be done with the standard library's `std::copy` function. In addition, `BandMatrix` can use `m.diagmajor_begin()` to iterate in diagonal-major order.

- Added the ability to do comma initialization for some special matrix types. These are also always listed in row-major order regardless of the actual storage order of the matrix in memory. Also, only the values in the corresponding parts of the matrices should be listed in the initialization.

  I chose not to add similar initializers for symmetric or hermitian matrix types, because I think initialization of them is clearer in terms of only the upper or lower triangle part that is stored in memory.
- Turned off error checking for the number of elements for a list initialization when TMV is in non-debugging mode (i.e. either `NDEBUG` or `TMV_NDEBUG` is defined).

× Changed all (non-`Small`) matrix and vector types to only use 2 template parameters. The second parameter is now an integer that includes all the information that was in the sometimes several parameters. The values should be combined using the bitwise or operator (`|`).

Now you can leave out *any* parameters that you don't want to specify, and the default value would be used. And for the values you do specify, you can list them in any order now, so you don't have to try to remember which parameter is supposed to be first.

× Removed the functions `read`, `write` and `writeCompact`, and replaced them with a more streamlined I/O interface. See §15 for more information.

× Removed the `auto_ptr` read functions.

• Added default constructors for the various `Vector` and `Matrix` classes. These construct a zero-sized vector or matrix, so normally you would want to resize it at some point.

× Removed the `ElementProd` and `AddElementProd` functions. The equivalent calculation is now done with a new operator called `ElemProd(a,b)` which returns a composite object representing the element-wise product of two vectors or matrices.

× Removed the routines that returned pointers to a `BaseMatrix` created with `new`.

× Changed the second template parameter of `VIt` and `CVIt` from either `Step` or `Unit` to the actual step size between elements, if known. If the step size is not known at compile time, you can use the special value `tmv::Unknown`, and the value will be determined at run time with a variable. This allows the user more flexibility for optimizing code.

• Added a few more "`ViewOf`" functions to allow direct views of memory as different kinds of matrix types directly specifying the step in each direction. These are more flexible than the versions that merely specify a `StorageType` to use. For non-contiguous data, those weren't sufficient. So now, you can view arbitrary data in memory as an `UpperTriMatrixView`, `LowerTriMatrixView`, `DiagMatrixView`, `BandMatrixView`, `SymMatrixView`, or `SymBandMatrixView` specifying the memory address of $m(0, 0)$ and the step size in each direction.

× Changed the type of the matrix and vector size values from `size_t` to `ptrdiff_t`.

× Changed all of the variables indexing offsets within the matrix (e.g. the arguments `i` and `j` of `m(i,j)`) from `int` to `ptrdiff_t` to avoid 32-bit `int` overflows with matrices that are larger than 2 GBytes.

× Changed the `svd().getV()` method to `svd().getVt()` to conform to the usual definition of the SVD: $A = USV^\dagger$. The method `getVt()` now refers to the matrix $V^\dagger$.

• Added the ability of `SymMatrix` SVD division to be done in place. In addition to the memory used for the `SymMatrix` values, it also uses the other half of the matrix that is normally not used for anything.

× Changed the mechanism for switching from the "Loose QRP" to the "Strict QRP" algorithm (and back). Now you should set this with the function
`tmv::UseStrictQRP();`
to start using the Strict QRP algorithm and
`tmv::UseStrictQRP(false);`
to stop using it. You can access the current state of the variable with
`bool tmv::QRP_IsStrict()`

× Changed what "`const`" in front of a `VectorView`, `MatrixView`, etc. means for the mutability of the view. Now a `const VectorView` works the same as a `ConstVectorView`. That is, it is no longer able to modify the underlying data.

• Changed the definitions of the various decomposition functions to accept either views (as before) or regular `Matrix`, `Vector`, etc. objects.

× Removed the `det` argument from the constructor for `Permutation`. Also, now the determinant is only calculated if you ask for it, rather than every time you create a `Permutation`.

• Removed warnings related to the `norm2` and `makeInverseATA` methods.

• Added a warning for when the matrix multiplication routine catches a `bad_alloc` exception and reverts to using a function that doesn't allocate memory. Also added documentation for the warning that had been output when the divide-and-conquer SVD algorithm has trouble converging.

× Changed the default TMV warning behavior to be no output. To turn on the warnings, the user needs to explicitly specify an `ostream` with the `tmv::WriteWarningsTo` function.

× Added a new compiler flag `-DTMV_EXTRA_DEBUG` and moved all the debugging statements that are more than $O(1)$ time to use this guard, including initializations of matrices and vectors with `888`.

× Changed the compiler flag `-DTMVNDEBUG` to `-DTMV_NDEBUG`.

- Fixed an error in the `TMV_VERSION_AT_LEAST` marco. The old version was actually backwards. It mistakenly returned whether the library's version was equal to or *earlier* than the provided values, rather than equal or later. The workaround if you need to check for versions before 0.70 is to use

  ```
  #if TMV_MINOR_VERSION >= 70
  ```

  to correctly exclude v0.65 and earlier versions, rather than

  ```
  #if TMV_VERSION_AT_LEAST(0,70)
  ```

  which will incorrectly allow version 0.65 and earlier through. This workaround will work until I release a version 1.0 when the `TMV_MAJOR_VERSION` value will change to 1, at which point hopefully nobody will need to be checking for versions earlier than v0.70 anymore.
- Fixed a bug where the symmetric SVD algorithm could go into an infinite loop (very rarely – it required particularly strange properties in the input matrix).
- Made the Givens rotations a bit more robust to overflow and underflow.
- Fixed a bug that `P*v` and `P*m` and similar operators didn't work if `v` or `m` is `Small`.
- Implemented a workaround for a bug in `icpc` version 12.0. It was getting an internal error when compiling `TMV_SymCHDecompose.cpp`.
- Added support for the `clang++` compiler.
- Changed the default for the `USE_STEGR` SCons flag to `false`.
- Changed the default for the `USE_GEQP3` SCons flag to `false`.
- Removed the `make` and `cmake` installation methods.
- Switched the default value for the SCons option `IMPORT_ENV` to `true`.

× Removed the functions and methods that had been deprecated in version 0.65.

**Version 0.71** This release was the first one to be included in fink. So on a Mac, if you use fink, you can now just type:

```
fink install tmv
```

to install TMV in `/sw/lib` and `/sw/include`. However, the process of getting this to happen required a few changes to the SCons files. I also learned a few SCons tricks that I've incorporated as well.

- Added a new SCons parameter `FINAL_PREFIX`.
- Moved the installed `tmv-link` file from `PREFIX/share/tmv-link` to the fink-recommended location `PREFIX/share/tmv/tmv-link`.
- Added documentation files to the installation process. The files `README`, `CHANGELOG`, `LICENSE`, and `TMV_Documentation` are now installed in `PREFIX/share/doc/tmv/`.
- Added a SCons flag `IMPORT_PREFIX` which specifies whether to include `PREFIX/include` and `PREFIX/lib` in the search paths.
- Added a new feature that scons will now automatically try to determine how many cpus your system has and use that for the number of build jobs to run at once. You can turn this off with either `N_BUILD_THREADS` or `scons -jN`.
- Changed the default value of `SHARED` to true, and fixed up some features of the shared library to conform to common standard practices.

**Version 0.72** This release is mostly just some minor bug fixes and some changes in the installation defaults. Probably the most significant change is that I have switched the license from GPL to BSD.

Here is a list of the changes from version 0.72 to 0.73. (See §18 for changes from previous versions to 0.72.) This release is completely backwards compatible as far as the header files and library are concerned. There are a couple (minor) new features, but the library should be link-compatible with previous versions.

- Changed the license from GPL to BSD. This seems to be where the open source community is moving, so I think it makes sense for TMV to have a more permissive license going forward.
- Made Scons correctly detect when g++ is really clang++. Apple has been lying about its C++ compiler in recent MacOS systems (10.7-10.9). They use clang++, but they call the program g++. This used to mess up TMV, since clang++ does not support OpenMP, so it would wrongly try to use OpenMP and end up with linking problems. TMV now detects when clang++ is masquerading as g++ and handles this correctly.
- Made the code compliant with the latest version of clang++, which suddenly got extra picky about how friend functions are declared, giving errors for code that doesn't comply with an obscure line in the standard that it used to be able to compile just fine (as does every other compiler as well). (Issue 9)
- Fixed some problems with the `install_name` in the shared libraries. (Issues 3 and 7)
- Fixed some warnings emitted by clang++. These weren't bugs – just some things that clang++ warns about that other compilers hadn't cared about.
- Changed the default value of `INST_INT` to `True`, so the `Matrix<int>` templates are instantiated now unless you specifically disable them. Enough people had expected them to be there and asked about why they were getting linking errors when they used `Matrix<int>`, so I decided to build them by default now. They don't add much to the library size.
- Added a default conversion from `VarConjIter` to `VIt`. It was an oversight that this wasn't possible before. (Issue 5)
- Fixed a bug in the test suite about `long double` I/O. The standard library does not read in `long double` variables at full precision. They are only accurate to double precision. So I changed the I/O tests to only test `long double` I/O to double precision. (Issue 8)
- Updated the linking checks for MKL to work with the latest version (11.1).

**Version 0.73** See §2.

# 19 License

This software is licensed under the FreeBSD License (also referred to as the Simplified BSD License or the 2-clause BSD License). Essentially, you can use this software however you want provided that you include the TMV_LICENSE file in any distribution that uses it.

Copyright (c) 1998-2016, Mike Jarvis All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

This software is based in part upon work supported by the National Aeronautics and Space Administration (NASA) under Grant No NAG5-10924, a Keck Foundation grant, and the University of Pennsylvania. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of NASA, the Keck Foundation, or the University of Pennsylvania. This software is made available to you on an "as is" basis with no representations or warranties, express or implied, including but not limited to any warranty of performance, merchantability, fitness for a particular purpose, commercial utility, non-infringement or title. Neither the author nor the organizations providing the support under which the work was developed will be liable to you or any third party with respect to any claim arising from your further development of the software or any products related to or derived from the software, or for lost profits, business interruption, or indirect special or consequential damages of any kind.