Template Matrix/Vector Library for C++
User Manual
Version 0.61

Mike Jarvis

July 8, 2008

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 §9 of subsequent releases.

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 += m.Transpose() * v1;
   m *= 3.;
   v2 -= m1*v1 + 3*v2 + 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 += 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,1.,mp,ldm,v1p,s1,1.,v2p,s2);
   ```

In fact, when installed with a BLAS library, the first version with TMV just calls those exact BLAS functions given by the second version. So speed is not sacrificed for the sake of code legibility.

However, a more complicated equation like

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

does not have a specialized routine, so it will require a couple temporary `Vectors`. 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.Tranpose()` 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 things like `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. 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), the code does its best to order the steps of the calculation, including using blocking techniques, to be reasonably efficient, but it is definitely not as fast as BLAS for most machines.

    This feature can be turned off at compile time if desired with the compilation flag -DNOBLAS, although this is not generally recommended if BLAS is available on your machine, and speed is important for your application. For some operations, the BLAS routines can be a factor of 10 or more faster than the native TMV code. (I'm working on reducing this gap, but I don't really expect to ever do better than a factor of 2 slower than an optimized BLAS library.)

11. **LAPACK**

    When possible, the code can call LAPACK routines, which may be faster than the native TMV code. For types that don't have LAPACK routines, the code uses blocked and/or recursive algorithms, which are similarly fast. Again, this feature can be turned off at compile time, this time with -DNOLAP (-DNOBLAS necessarily turns off the LAPACK calls as well.)

    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 turn off the LAPACK calls.

    The only real exception to this statement is the eigenvector calculation for a hermitian matrix. I have not yet implemented the new RRR (Relatively Robust Representation) algorithm by Dhillon. So if the code is spending a significant time calculating eigenvectors, it may be worth having it call the LAPACK routines.

12. **Comments**

    The code is highly commented, especially for the more complicated algorithms. I have this egotistical quirk that I need to understand all of the code I write. So none of the algorithms are just taken verbatim from LAPACK or anything like that. I force myself to understand the algorithm (usually from Golub and van Loan, but sometimes from a journal article or something I find on the net), write a long comment explaining how it works, and then finally write the code from scratch. Then I usually compare my code to the LAPACK version, at which point I have occasionally found ways to either speed it up or make it more accurate (which also get commented, of course).

    But the point is - if you want to understand how, say, the blocked Bunch-Kaufman algorithm works, you'll probably do better looking at the comment in the TMV code than trying to decipher the LAPACK routines. Plus, the code itself is usually a lot more readable, since the arithmetic is written with the operator overloads and such rather than the cryptic BLAS function names.

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

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

This file includes all the other files for the basic TMV routines. The special matrices described below are not included in this. 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 10×10 `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.

Also, throughout most of the documentation, I will write `T` for the underlying type. Wherever you see `T`, you should put `double` or `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.

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 a couple places where `CT` indicates “the complex type associated with `T`”.

It may be worth noting that `Matrix<int>` is possible as well. 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 (or more accurately, not turned off via the compile flag -DNDEBUG), then trying to do anything that requires `sqrt` or `epsilon` for `int`s will result in a runtime error.

There are three fonts used in this document. First, times is used for the main text. 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 syntax in this documentation is not very rigorous, aiming to maximize readability of the code. I tend to freely mix the syntax with which a function is used with how it is declared in order to try to provide all of the information you will need in one line. For example, the constructor listed as:

```
tmv::Vector<T,index> v(size_t n, const T* vv)
```

is actually declared in the code as:

```
tmv::Vector<T,index>::Vector(size_t n, const T* vv);
```

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

```
size_t 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 Vectors

The `Vector` class is our mathematical vector. Not to be confused 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.

`Vector` inherits from its base class `GenVector` (i.e. "generic vector"). Most operations that do not modify the data are actually defined in `GenVector` rather than `Vector`, although some routines are overridden in `Vector` for speed reasons.

The other classes that inherit from `GenVector` are `VectorView`, `ConstVectorView` (both described in more detail below - see §2.3), and `VectorComposite`, which is the base class for all arithmetic operations that return a (logical) `Vector`. This means that any of these other objects can be used any time a `Vector` can be used in a non-assignable capacity. For example, `Norm(v1+m*v2)` is completely valid, and will automatically create the necessary temporary `Vector` to store the result of the mathematical operation in the parentheses.

There is another template argument for a `Vector` in addition to `T` (which represents the data type of the elements). The second template argument may be either `tmv::CStyle` or `tmv::FortranStyle`. Or it may be omitted, in which case `CStyle` is assumed. This argument governs how the element access is performed.

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)`. And ranges indicate 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`.

The only thing to watch out for about the indexing is that `GenVector` and `VectorComposite` do not have the extra `index` template argument and are always indexed using the C-style convention. Therefore, if you want to index a `GenVector` using the Fortran-style convention, you would need to recast it as an object of type `ConstVectorView<T,FortranStyle>`. A `VectorComposite` would not generally be indexed, but if you did want to do so using the Fortran-style conventions, you would need to explicitly instantiate it as a `Vector<T,FortranStyle>`.

2.1 Constructors

Here, `T` is used to represent the data type of the elements of the `Vector` (e.g. `double`, `complex<double>`, `int`, etc.) and `index` is either `tmv::CStyle` or `tmv::FortranStyle`. In all of the constructors the `index` template argument may be omitted, in which case `CStyle` is assumed.

- `tmv::Vector<T,index> v(size_t n)`

  Makes a `Vector` of size `n` with <u>uninitialized</u> values. If debugging is turned on (this is actually the default - turn off debugging by compiling with -DNDEBUG), 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,index> v(size_t n, T x)`

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

- `tmv::Vector<T,index> v(size_t n, const T* vv)`
  `tmv::Vector<T,index> v(const std::vector<T>& vv)`

Make a `Vector` which copies the elements of `vv`. For the first one, `n` specifies the length. The second gets the length from `vv`.

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

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

-
  ```
  tmv::Vector<T,index> v1(const tmv::GenVector<T2>& 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.

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

  Makes a `VectorView` (see §2.3 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.

2.2 Access

- `size_t v.size() const`

  Returns the size (length) of `v`.

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

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

  If `v` is a `const Vector`, a `ConstVectorView`, or a `GenVector`, 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 `typename 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.

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

These provide iterator-style access into a `Vector`, which works just like the standard template library's iterators. If `v` is a `VectorView`, the iterator types are slightly different from the `Vector` iterators, so you should declare them as `typename tmv::VectorView<T>::iterator`, etc. instead.

2.3 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.

One slightly non-intuitive thing about `VectorViews` is that a `const VectorView` is still mutable. The `const` in this case means that one cannot change the components to which the view refers. But a `VectorView` is inherently an object that can be used to modify the underlying data, regardless of any `const` in front of it.

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`, a `ConstVectorView`, or any other `GenVector`, 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. `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).Zero();
  ```

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

  If you are not going to be modifying `v` in the function, you only need to write one function, and you should use the base class `GenVector` for the argument type:

  ```
  double foo(const tmv::GenVector<double>& v)
  { ... [doesn't modify v] ... }
  ```

  The arguments could then be a `const Vector`, a `ConstVectorView`, or even a `VectorComposite`.

- `v.Real()`
  `v.Imag()`

  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.Real();
  tmv::VectorView<double> vi = v.Imag();
  ```

2.4 Functions of a vector

Functions that do not modify the `Vector` are defined in `GenVector`, and so can be used for any type derived from `GenVector`: `Vector, ConstVectorView, VectorView`, or `VectorComposite`.

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

2.4.1 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:

```
RT v.Norm() const
RT Norm(v)
```

are equivalent. In each case, `v` can be any `GenVector`. Just to remind you, `RT` refers to the real type associated with `T`. So for `T` = `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`: $||v||_1 = \sum_i |v(i)|$.

- `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`.

- `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`: $||v||_\infty = \max_i |v(i)|$.

- `RT v.MaxAbsElement(int* i=0) const`
  `RT MaxAbsElement(v)`
  `RT v.MinAbsElement(int* i=0) const`
  `RT MinAbsElement(v)`
  `T v.MaxElement(int* i=0) const`
  `T MaxElement(v)`
  `T v.MinElement(int* i=0) const`
  `T MinElement(v)`

  The maximum/minimum element either by absolute value (the first four cases) or actual value (the last four cases). 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.

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

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

2.4.2 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.Zero();`

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

- `v.SetAllTo(T x);`

  Set each element to the value `x`.

- `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.

- `v.AddToAll(T x)`

  Add the value `x` to each element.

- `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.Permute(const int* p)
  v.Permute(const int* p, int i1, int i2)
  v.ReversePermute(const int* p)
  v.ReversePermute(const int* p, int i1, int i2)
  ```

  The first one performs a series of swaps: (`v(0)`,`v(p[0])`), (`v(1)`,`v(p[1])`), ... The second starts at `i1` and ends at `i2-1` rather than doing the whole range from 0 to `n-1`. The last two work the same way, but do the swaps in the opposite order.

  Note: The indices listed in a permutation array (`p`) always use the C-style convention, even if `v` uses Fortran-style indexing.

-
  ```
  v.Sort(int* p=0, tmv::ADType ad=tmv::ASCEND,
        tmv::COMPType comp=tmv::REAL_COMP)
  ```

  Sorts the `Vector v`, returning the swaps required in the array `p`. If you do not care about the swaps, you can set `p = 0`. Note: the returned permutation array, `p`, uses the C-style indexing convention even if `v` uses Fortran-style indexing.

The second parameter, `ad`, determines whether the sorted `Vector` will have its elements in ascending or descending order. The possible values are `ASCEND` and `DESCEND`.

The third 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 `REAL_COMP`, `ABS_COMP`, `IMAG_COMP`, and `ARG_COMP`. Only the first two make sense for non-complex `Vectors`.

- `Swap(v1,v2)`

  Swap the corresponding elements of `v1` and `v2`. Note that this does physically swap the data elements, not just some pointers to the data. That's because this function is mostly called on views into larger data fields: for example, swapping two rows of a `Matrix`. In any case, it always takes $O(N)$ time, not $O(1)$.

2.5 Arithmetic

2.5.1 Operators

All the usual operators work the way you would expect for `Vectors`. 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 `Vectors` 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 may be any `GenVector`.

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.

```
v2 = -v1;
v2 = x * v1;
v2 = v1 [*/] x;
v3 = v1 [+-] v2;
v [*/]= x;
v2 [+-]= v1;
v1 == v2
v1 != v2
x = v1 * v2;
```

The last one, `v1 * v2`, returns 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 here, 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.

2.5.2 Subroutines

Each of the above equations use deferred calculation 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);
```

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. While you shouldn't ever need to use these directly, it may help you understand when the code will require temporary `Vectors`. If you do use these, note that the `v` parameters are `VectorViews`, rather than `Vectors`. So you would need to call them with `v.View()` if `v` is a `Vector`.

- `MultXV(T x, const VectorView<T>& v)`
  Performs the calculation `v *= x`.
- `MultXV(T x, const GenVector<T1>& v1, const VectorView<T>& v2)`
  Performs the calculation `v2 = x*v1`.
- `AddVV(T x, const GenVector<T1>& v1, const VectorView<T>& v2)`
  Performs the calculation `v2 += x*v1`.

-
  ```
  AddVV(T x1, const GenVector<T1>& v1, T x2, const GenVector<T2>& v2,
        const VectorView<T>& v3)
  ```
  Performs the calculation `v3 = x1*v1 + x2*v2`.
- `T MultVV(const GenVector<T>& v1, const GenVector<T2>& v2)`
  Performs the calculation `v1*v2`.

More complicated arithmetic equations such as

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

will require one or more temporary vectors, and so may be less efficient than you might like, but the code should return the correct result, no matter how complicated the equation is.

2.6 Input/Output

The simplest output 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 in one of two ways:

```
tmv::Vector<T> v(n);
is >> v;
std::auto_ptr<tmv::Vector<T> > pv;
is >> pv;
```

For the first version, the `Vector` must already be declared, which requires knowing how big it needs to be. If the input `Vector` does not match in size, an exception of type `tmv::ReadError` is thrown. The second version allows you to automatically get the size from the input. The `Vector` pointed to by `v2` will be created according to whatever size the input `Vector` is.

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

```
v.Write(std::ostream& os, RT thresh)
```

which is equivalent to

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

but without requiring the temporary `Vector`.

3 Dense Rectangular Matrices

The `Matrix` class is our dense matrix class. It inherits from `GenMatrix`, which has the definitions of all the methods that do not modify the `Matrix`.

The other classes that inherit from `GenMatrix` are `ConstMatrixView`, `MatrixView` (see §3.3 below), and `MatrixComposite`, which is the base class for the various arithmetic operations that return a (logical) `Matrix`.

`GenMatrix` in turn inherits from `BaseMatrix`. All of the various special `Matrix` classes also inherit from `BaseMatrix`. `BaseMatrix` has virtual declarations for the functions that can be performed on any kind of `Matrix` regardless of its structure or sparsity.

In addition to the data type template parameter (indicated here by `T` as usual), there is also a storage template parameter, which may be either `RowMajor` or `ColMajor`.

Finally, there is also a template argument indicating which indexing convention you want the matrix to use, which may be either `CStyle` or `FortranStyle`.

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 3×3 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 3×3 submatrix as given above.

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`.) However, as for `GenVector`, you should be aware that `GenMatrix` and `MatrixComposite` do not have the extra template argument and are always indexed using the C-style convention. So if you want to index one of these using the Fortran-style convention, you need to (respectively) cast the `GenMatrix` as a `ConstMatrixView<T,FortranStyle>` or instantiate the `MatrixComposite` as a `Matrix<T,FortranStyle>`.

You may omit the indexing template argument, in which case `CStyle` is assumed. And if so, you may also then omit the storage argument, in which case `ColMajor` is assumed. If you want to specify `FortranStyle` indexing, you need to include the storage argument.

3.1 Constructors

We use `stor` to indicate the storage template argument. This argument must be either `tmv::RowMajor` or `tmv::ColMajor`. And `index` indicates either `tmv::CStyle` or `tmv::FortranStyle`. The default values for these are `tmv::ColMajor` and `tmv::CStyle` if the arguments are omitted.

- `tmv::Matrix<T,stor,index> m(size_t nrows, size_t ncols)`

  Makes a `Matrix` with `nrows` rows and `ncols` columns with <u>uninitialized</u> values. If debugging is turned on (this is actually the default - turn off debugging by compiling with -DNDEBUG), 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,stor,index> m(size_t nrows, size_t ncols, T x)`

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

-
  ```
  tmv::Matrix<T,stor,index> m(size_t nrows, size_t ncols, const T* vv)
  tmv::Matrix<T,stor,index> m(size_t nrows, size_t ncols,
  ```

```
      const std::vector<T>& vv)
```

Makes a `Matrix` with `nrows` rows and `ncols` columns, which copies the elements of `vv`.

If `stor` is `RowMajor`, then the elements of `vv` are taken to be in row-major order: first the `ncols` elements of the first row, then the `ncols` elements of the second row, and so on for the `nrows` rows. Likewise if `stor` is `ColMajor`, then the elements of `vv` are taken to be in column-major order - the elements of the first column, then the second, and so on.

-
  ```
  tmv::Matrix<T,stor,index> m(const std::vector<std::vector<T> >& vv)
  ```
  Makes a `Matrix` with elements `m(i,j) = vv[i][j]`. The size of the `Matrix` is taken from the sizes of the `vectors`. (If `index` is `FortranStyle`, then `m(i,j) = vv[i-1][j-1]`.)

-
  ```
  tmv::Matrix<T,index> m1(const GenMatrix<T2>& 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,index> m =
        tmv::MatrixViewOf(T* vv, size_t nrows, size_t ncols,
        StorageType stor)
  tmv::ConstMatrixView<T,index> m =
        tmv::MatrixViewOf(const T* vv,  size_t nrows, size_t ncols,
        StorageType stor)
  ```
  Makes a `MatrixView` (see §3.3 below) that refers to the exact elements of `vv`, not copying them to new storage.

-
  ```
  tmv::MatrixView<T,index> m =
        RowVectorViewOf(VectorView<T,index> v)
  tmv::ConstMatrixView<T,index> m =
        RowVectorViewOf(ConstVectorView<T,index> v)
  ```
  Makes a view of the `Vector`, which treats it as a $1 \times n$ `Matrix` (i.e. a single row).

-
  ```
  tmv::MatrixView<T,index> m =
        ColVectorViewOf(VectorView<T,index> v)
  tmv::ConstMatrixView<T,index> m =
        ColVectorViewOf(ConstVectorView<T,index> v)
  ```
  Makes a view of the `Vector`, which treats it as an $n \times 1$ `Matrix` (i.e. a single column).

3.2 Access

-
  ```
  size_t m.nrows() const
  size_t m.ncols() const
  size_t m.colsize() const
  size_t 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[int i][int j] const
  typename tmv::Matrix<T>::reference m(int i, int j)
  typename tmv::Matrix<T>::reference m[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`, a `ConstMatrixView`, or a `GenMatrix`, 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 object, which is an lvalue (i.e. it is assignable), but which may not be `T&`. It has the type `typename MatrixView<T>::reference`, (which is the same as `typename 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.

-
  ```
  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.

3.3 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.

It is worth pointing out again that a `const MatrixView` is still mutable, just like a `const VectorView`. The `const` just means that you cannot change which elements the view references.

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`, `ConstMatrixView`, or any other `GenMatrix`, 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::ColMajor,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.Rows(int i1, int i2)`
  `m.Cols(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 are 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)
```

respectively. The equivalent in Fortran-style indexing would be:

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

- `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.Real()`
  `m.Imag()`

  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.Real();
tmv::MatrixView<double> mi = m.Imag();
```

- `m.UpperTri(DiagType dt = NonUnitDiag)`
  `m.LowerTri(DiagType dt = NonUnitDiag)`

  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 both 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 §4.2 for more details about this parameter and triangular matrices in general.

3.4 Functions of a matrix

Functions that do not modify the `Matrix` are defined in `GenMatrix`, and so can be used for any type derived from `GenMatrix`: `Matrix`, `ConstMatrixView`, `MatrixView`, or `MatrixComposite`. Functions that modify the `Matrix` are only defined for `Matrix` and `MatrixView`.

3.4.1 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. In each case, `m` can be any `GenMatrix<T>`. 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`: $||m||_1 = \max_j(\sum_i |m(i,j)|)$.

- `RT m.Norm2() const`
  `RT Norm2(m)`
  `RT m.DoNorm2() const`

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

  This function can be fairly expensive if you have not already performed an SV decomposition of `m`, so the first two versions output a warning to `std::cout` if there is no SV decomposition already set. If you understand that you are asking TMV to perform an SV decomposition to calculate $||m||_2$, and you are ok with it, you can either compile with the `-DNWARN` flag (which suppresses all warnings), or compile with the `-DNDEBUG` flag (which suppresses all warnings and most other error checking), or you can just call `DoNorm2()` instead, which will do the necessary SV decomposition without any warnings.

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

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

- `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`.

- `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) instead, which can help avoid underflow or overflow problems.

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

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

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

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

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

  The determinant of `m`, $\det(m)$. For speed issues regarding this function, see §3.6 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 3.6.6 for more details.

- `bool m.Singular() const`

  Return whether `m` is singular, i.e. $\det(m) = 0$. (Singular matrices are discussed in more detail in §3.6.8.)

- `tmv::Matrix<T> minv = m.Inverse()`
  `tmv::Matrix<T> minv = Inverse(m)`
  `void m.Inverse(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 §3.6.4 and §3.6.8 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 an object whose type is derived from `MatrixComposite`. The calculation of the inverse is then delayed until there is a place to store the result.

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

  Set `cov` to be $(m^\dagger m)^{-1}$.

  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 §3.6.3). 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.

3.4.2 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.Zero()`

  Set all elements to 0.

- `m.SetAllTo(T x)`

  Set all elements to the value `x`.

- `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.

- `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.Zero().diag().SetAllTo(x)`.

- `m.ConjugateSelf()`

  Conjugate each element. Note the difference between this and `m.Conjugate()`, which returns a view 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 view 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.

- `m.PermuteRows(const int* p)`
  `m.PermuteRows(const int* p, int i1, int i2)`
  `m.ReversePermuteRows(const int* p)`
  `m.ReversePermuteRows(const int* p, int i1, int i2)`

  These are equivalent to the corresponding routines for `Vectors` (`Permute` and `ReversePermute`), performing a series of `SwapRows` commands.

- `m.PermuteCols(const int* p)`
  `m.PermuteCols(const int* p, int j1, int j2)`
  `m.ReversePermuteCols(const int* p)`
  `m.ReversePermuteCols(const int* p, int j1, int j2)`

  Same as above, but performing the a series of `SwapCols` commands.

- `Swap(m1,m2)`

  Swap the corresponding elements of `m1` and `m2`. Note that this does physically swap the data elements, not just some pointers to the data, so it takes $O(N^2)$ time.

3.5 Arithmetic

3.5.1 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 `[*/]`.

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

```
m2 [+-]= m1
v2 = m * v1
v2 = v1 * m
v *= m
m3 = m1 * m2
m2 *= m1
m1 == m2
m1 != m2
```

Note that the orientation of a vector is inferred from context. `m*v` involves a column vector, and `v*m` involves a row vector.

It is sometimes convenient to be able to treat scalars as the scalar times an identity matrix. So the following operations are allowed and use that convention:

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

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

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

3.5.2 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. 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!]
```

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)
```

3.5.3 Subroutines

As with `Vectors`, we try to defer calculations until we have a place to store them. So `m*v` returns an object that can be assigned to a `Vector`, but hasn't performed the calculation yet.

The limit to how complicated an expression can be without resorting to a temporary object is set by the functions that the code eventually calls to perform the calculation. While you shouldn't ever need to use these directly, it may help you understand when the code will create a temporary `Matrix`[1].

- `MultXM(T x, const MatrixView<T>& m)`

  Performs the calculation `m *= x`.

[1]If you do use these, note that the last parameters are `MatrixViews`, rather than `Matrixes`. So you would need to call them with `m.View()` if `m` is a `Matrix`. (For `MultMV`, it would be `v.View()`.)

- `MultMV<bool add>(T x, const GenMatrix<Tm>& m,`
  `      const GenVector<Tv1>& v1, const VectorView<T>& v2)`

  Performs the calculation `v2 (+=) x*m*v1` where "`(+=)`" means "+=" if `add` is true and "=" if `add` is false.

- `AddMM(T x, const GenMatrix<T1>& m1, const MatrixView<T>& m2)`

  Performs the calculation `m2 += x*m1`.

- `AddMM(T x1, const GenMatrix<T1>& m1, T x2, const GenMatrix<T2>& m2,`
  `      const MatrixView<T>& m3)`

  Performs the calculation `m3 = x1*m1 + x2*m2`.

- `MultMM<bool add>(T x, const GenMatrix<T1>& m1,`
  `      const GenMatrix<T2>& m2, const MatrixView<T>& m3)`

  Performs the calculation `m3 (+=) x*m1*m2` where "`(+=)`" means "+=" if `add` is true and "=" if `add` is false.

- `Rank1Update<bool add>(T x, const GenVector<T1>& v1,`
  `      const GenVector<T2>& v2, const MatrixView<T>& m)`

  Performs the calculation `m (+=) x*(v1^v2)` where "`(+=)`" means "+=" if `add` is true and "=" if `add` is false.

3.6 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.

3.6.1 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 §3.6.3) is used to calculate `v2` and `v3`.

3.6.2 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 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: $(\ 1/s_i \quad x_i/s_i \quad x_i^2/s_i\ )$. The corresponding element of the vector b shoudl be $(\ y_i/s_i\ )$. It is easily verified that $||b - Ax||_2^2$ is the normal χ^2 expression. The solution returned by `x = b/A` would then be the least-square fit solution: $(\ c \quad d \quad e\ )$, which would be the solution which minimizes χ^2.

The underdetermined case is not so common, but can still be defined reasonably. As mentioned above, there are 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 $(A^\dagger A)^{-1}$. It is not very efficient to calculate this matrix explicitly and then invert it. But once you have calculated the decomposition needed for the division, it is quite easy. So we provide the routine

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

to perform the calculation efficiently. (Make sure you save the decomposition with `A.SaveDiv()` - see §3.6.5 for more about this.)

3.6.3 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 §3.6.8.)

   There are two slightly different algorithms for doing a QRP decomposition, controlled by a global `bool` variable: `tmv::QRPDiv<T>::StrictQRP`. If this is set to true, then the decomposition will make the diagonal elements of R be strictly decreasing (in absolute value) from upper-left to lower-right.

   If it is false, however (the default), then there will be no diagonal element of R below and to the right of one which is much smaller in absolute value, where "much" means the ratio will be at most $\epsilon^{1/4}$, where ϵ 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 algorithm.

4. **Singular Value Decomposition**: (`dt = tmv::SV`) $A = USV$ where U is unitary (or column-unitary if $M > N$), S is diagonal and real, and V is unitary (or row-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 §3.6.8 on singular matrices below.

3.6.4 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. §5), QRP will be close to correct (again failing only the last equation) and will not throw an exception, and SV will be correct.

3.6.5 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`[2]:

```
m.DivideInPlace()
```

3.6.6 Determinants

The calculation of the determinant of a matrix requires similar calculations as 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 is simply the product of the values along the diagonal.

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. 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, defined to be `(det/abs(det))`.

3.6.7 Accessing the decompositions

Sometimes, you 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:

  ```
  ConstLowerTriMatrixView<T> m.LUD().GetL()
  ConstUpperTriMatrixView<T> m.LUD().GetU()
  int* m.LUD().GetP()
  bool m.LUD().IsTrans()
  ```

  `GetL()` and `GetU()` return L and U. `GetP()` returns the permutation array, P, in a form that can be used as an argument to the routines like `m.PermuteRows(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().GetL() * m.LUD().GetU();
  m2.ReversePermuteRows(m.LUD().GetP());
  if (m.LUD().IsTrans()) m2.TransposeSelf();
  ```

[2]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.

- For the QR decomposition, we have:

  ```
  tmv::Matrix<T> m.QRD().GetQ();
  tmv::ConstUpperTriMatrix<T>& m.QRD().GetR();
  bool m.QRD().IsTrans();
  ```

  `GetQ()` and `GetR()` return Q and R. `GetQ()` needs to make a new `Matrix`, since the `QRD` class actually stores Q in a compact form rather than as the actual matrix. This routine converts it into a regular matrix. (No such explicit conversion is required when the class is used to perform divisions.) `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.QRD().IsTrans() ? m2.Transpose() : m2.View();
  m2v = m.QRD().GetQ() * m.QRD().GetR();
  ```

- For the QRP decomposition, we have:

  ```
  tmv::Matrix<T> m.QRPD().GetQ()
  tmv::ConstUpperTriMatrixView<T>& m.QRPD().GetR()
  int* m.QRPD().GetP()
  bool m.QRPD().IsTrans()
  ```

  `GetQ()` and `GetR()` return Q and R. `GetP()` returns the permutation array, 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();
  m2v.ReversePermuteCols(m.QRPD().GetP())
  ```

- For the SV decomposition, we have:

  ```
  tmv::ConstMatrixView<T>& m.SVD().GetU()
  tmv::ConstVectorView<RT>& m.SVD().GetS()
  tmv::ConstMatrixView<T>& m.SVD().GetV()
  bool m.SVD().IsTrans()
  ```

  `GetU()` and `GetV()` return U and V. `GetS()` returns a vector, which is the diagonal of the diagonal matrix S. To use S as a matrix, you can use `DiagMatrixViewOf(S)`. `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.SVD().IsTrans() ? m2.Transpose() : m2.View();
  m2v = m.SVD().GetU() * DiagMatrixViewOf(m.SVD().GetS()) *
        m.SVD().GetV();
  ```

3.6.8 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, ϵ.

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

```
bool m.Singular()
```

which basically just returns whether `m.Det() == 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, ϵ, for the underlying type (`double`, `float`, etc.), then the matrix is singular, or effectively so. Note: the value of ϵ is accessible with:

```
std::numeric_limits<T>::epsilon()
```

The inverse of this ratio, $S(0)/S(N-1)$, is known as the "condition" of the matrix (specifically the 2-condition, or κ_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 the 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 α will produce the same solution: $Ax' = Ax = b$.

The usual desired solution is the x with minimum 2-norm, $||x||_2$. With the 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[3]:

```
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());
```

[3]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 the 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 ϵ 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 a calculation of the SVD, which is relatively expensive compared to the other norms if you have not already calculated the SVD (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 §6.2 for how to do this.

3.7 Input/Output

The simplest output 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 in one of two ways:

```
tmv::Matrix<T> m(nrows,ncols);
is >> m;
std::auto_ptr<tmv::Matrix<T> > pm;
is >> pm;
```

For the first version, the `Matrix` must already be declared, which requires knowing how big it needs to be. If the input `Matrix` does not match in size, a runtime error will occur. The second version allows you to get the size from the input. `m2` will be created (with `new`) according to whatever size the input `Matrix` is.

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

```
m.Write(std::ostream& os, RT thresh)
```

which is equivalent to

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

but without requiring the temporary `Matrix`.

3.8 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 2×2 or 3×3 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`, 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.

The class `SmallMatrix` inherits from `GenSmallMatrix`, which in turn inherits from `BaseMatrix`. Likewise, the class `SmallVector` inherits from `GenSmallVector`.

All the `SmallMatrix` and `SmallVector` routines are included by:

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

3.8.1 Constructors

The template arguments `M` and `N` below are both integers and represent the size of the matrix or vector as indicated. The template argument `stor` may be either `tmv::RowMajor` or `tmv::ColMajor`, and `index` may be 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 they are omitted. (Likewise for `index` with a `Vector`.)

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

  Makes a `SmallVector` of size `N` with uninitialized values. If debugging is turned on (i.e. not turned off with -DNDEBUG), then the values are in fact initialized to 888.

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

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

- `tmv::SmallVector<T,N,index> v(const T* vv)`
  `tmv::SmallVector<T,N,index> v(const std::vector<T>& vv)`

  Makes a `SmallVector` with values copied from vv.

- `tmv::SmallVector<T,M,N,stor,index> v1(const GenVector<T>& v2)`

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

- `tmv::SmallVector<T,M,N,stor,index> v1(const GenSmallVector<T2>& 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.

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

  Makes an `M` $\times$ `N` `SmallMatrix` with uninitialized values. If debugging is turned on (i.e. not turned off with -DNDEBUG), then the values are in fact initialized to 888.

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

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

- `tmv::SmallMatrix<T,M,N,stor,index> m(const T* vv)`
  `tmv::SmallMatrix<T,M,N,stor,index> m(const std::vector<T>& vv)`
  `tmv::SmallMatrix<T,M,N,stor,index> m(`
  `        const std::vector<std::vector<T> >& vv)`

  Makes an `M` × `N` `SmallMatrix` with values copied from vv.

- `tmv::SmallMatrix<T,M,N,stor,index> m1(const GenMatrix<T>& m2)`

  Makes a `SmallMatrix` from a regular `Matrix`.

- `tmv::SmallMatrix<T,M,N,stor,index> m1(const GenSmallMatrix<T2>& 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::ConstSmallVectorView<T,N,step,isconj,index> v(`
  `        const ConstVectorView<T>& v2)`
  `tmv::SmallVectorView<T,N,step,isconj,index> v(`
  `        const VectorView<T>& v2)`
  `tmv::ConstSmallMatrixView<T,M,N,stepi,stepj,isconj,index> m(`
  `        const ConstMatrixView<T>& m2)`
  `tmv::SmallMatrixView<T,M,N,stepi,stepj,isconj,index> m(`
  `        const MatrixView<T>& m2)`

  Convert a regular `Vector` or `Matrix` view into a small variety. Note that the step size between elements along a column (`stepi`) and along a row (`stepj`) must be specified in the template arguments. Also whether or not the view is the conjugate of the actual data values (`isconj`). The parameters `isconj` and `index` may be omitted, with the defaults `false` and `CStyle` being assumed.

- `tmv::ConstSmallVectorView<T,N,1,false,index> v =`
  `        tmv::SmallVectorViewOf<N,index>(const T* v);`
  `tmv::SmallVectorView<T,N,1,false,index> v =`
  `        tmv::SmallVectorViewOf<N,index>(T* v);`
  `tmv::ConstSmallMatrixView<T,M,N,N,1,false,index> m =`
  `        tmv::SmallMatrixViewOf<M,N,RowMajor,index>(const T* m);`
  `tmv::ConstSmallMatrixView<T,M,N,1,M,false,index> m =`
  `        tmv::SmallMatrixViewOf<M,N,ColMajor,index>(const T* m);`
  `tmv::SmallMatrixView<T,M,N,N,1,false,index> m =`
  `        tmv::SmallMatrixViewOf<M,N,RowMajor,index>(T* m);`
  `tmv::SmallMatrixView<T,M,N,1,M,false,index> m =`
  `        tmv::SmallMatrixViewOf<M,N,ColMajor,index>(T* m);`

  View actual data as a `SmallVector` or `SmallMatrix`. In all cases, the `index` template parameter may be omitted, with `CStyle` being assumed.

3.8.2 Access

```
v.size()
v[i] = v(i)
m.nrows() = m.colsize()
m.ncols() = m.rowsize()
m(i,j) = m[i][j]
```

The all return a `SmallVectorView`:

```
v.Reverse()
v.Conjugate()
m.row(i)
m.col(j)
m.diag()
```

These all return a `SmallMatrixView`:

```
m.Transpose()
m.Conjugate()
m.Adjoint()
m.View()
m.Real()
m.Imag()
```

These return either a `VectorView` or a `MatrixView`:

```
v.SubVector(i1,i2)
m.row(i,j1,j2)
m.col(j,i1,i2)
m.diag(i)
m.diag(i,j1,j2)
m.SubVector(i,j,istep,jstep,size)
m.SubMatrix(i1,i2,j1,j2)
m.SubMatrix(i1,i2,j1,j2,istep,jstep)
m.Rows(i1,i2)
m.Cols(j1,j2)
m.RowPair(i1,i2)
m.ColPair(j1,j2)
```

Since the sizes of all of the returned views in this last set are not known at compile time, these methods cannot return a `SmallVectorView` or `SmallMatrixView`. If you want, you can convert these into a `SmallVectorView` or a `SmallMatrixView` by simply assigning it to one that is the correct size. e.g.
`tmv::SmallMatrixView<T,M,2> m13 = m.ColPair(1,3);`

3.8.3 Functions

`SmallVector`s and `SmallMatrix`es all have exactly the same function methods as the regular varieties. Likewise, the syntax of the arithmetic is identical. There are only a few methods that are not done inline.

First, I have not implemented any division routines inline yet. So all divisions by a `SmallMatrix` just use the regular `Matrix` division routines. This includes the related routines like `Det` and `Norm2`.

Second, reading a `SmallMatrix` or `SmallVector` from a file uses the regular `Matrix` I/O methods. Also, there is no `auto_ptr` version of these read operations, since you need to know the size of a `SmallMatrix` or `SmallVector` at compile time anyway, so there is no way to wait until the file is read to determine the size.

Last, the `Sort` command for a `SmallVector` just uses the regular `Vector` version.

4 Special Matrices

Most functions and methods for the various special matrix varieties work the same as the for regular dense, rectangular matrices. In these cases, we will just list the functions that are allowed for each special matrix with the effect understood to be the same as for a regular `Matrix`. Of course, there are usually algorithmic speed-ups, which the code will use to take advantage of the particular structure. Whenever there is a difference in how a function works, we will explain the difference.

4.1 Diagonal matrices

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

The class `DiagMatrix` inherits from `GenDiagMatrix`, which in turn inherits from `BaseMatrix`. The various views and composite classes described below also inherit from `GenDiagMatrix`.

All the `DiagMatrix` routines are included by:

```
#include "TMV_Diag.h"
```

4.1.1 Constructors

As usual, the optional `index` template argument specifies which indexing style to use.

- `tmv::DiagMatrix<T,index> d(size_t n)`

  Makes an `n` × `n` `DiagMatrix` with uninitialized values. If debugging is turned on (i.e. not turned off with -DNDEBUG), then the values along the diagonal are in fact initialized to 888.

- `tmv::DiagMatrix<T,index> d(size_t n, T x)`

  Makes an `n` × `n` `DiagMatrix` with all values along the diagonal equal to `x`.

- `tmv::DiagMatrix<T,index> m(size_t n, const T* vv)`
  `tmv::DiagMatrix<T,index> m(const std::vector<T>& vv)`

  Makes a `DiagMatrix` which copies the elements of `vv`.

- `tmv::DiagMatrix<T,index> d(const GenVector<T>& v)`

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

- `tmv::DiagMatrix<T,index> d(const GenMatrix<T>& m)`

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

- `tmv::DiagMatrix<T,index> d1(const GenDiagMatrix<T2>& 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::DiagMatrixView<T,index> d =
        DiagMatrixViewOf(VectorView<T,index> v)
  tmv::ConstDiagMatrixView<T,index> d =
        DiagMatrixViewOf(ConstVectorView<T,index> v)
  ```

  Makes a `DiagMatrixView` whose diagonal is `v`.

-
  ```
  tmv::DiagMatrixView<T,index> d =
        DiagMatrixViewOf(MatrixView<T,index> m)
  tmv::ConstDiagMatrixView<T,index> d =
        DiagMatrixViewOf(ConstMatrixView<T,index> m)
  ```
  Makes a `DiagMatrixView` whose diagonal is the main diagonal of `m`.

-
  ```
  tmv::DiagMatrixView<T,index> d =
        tmv::DiagMatrixViewOf(T* vv, size_t n)
  tmv::ConstDiagMatrixView<T,index> d =
        tmv::DiagMatrixViewOf(const T* vv, size_t n)
  ```
  Make a `DiagMatrixView` whose diagonal consists of the actual memory elements `vv`.

4.1.2 Access

```
d.nrows() = d.ncols() = d.colsize() = d.rowsize() = d.size()
d(i,j)
d(i) = d(i,i)
```

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

```
d.diag()
```

```
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.Real()
d.Imag()
```

4.1.3 Functions

```
RT d.Norm1() = Norm1(d)
RT d.Norm2() = Norm2(d) = d.DoNorm2()
RT d.NormInf() = NormInf(d)
RT d.MaxAbsElement() = MaxAbsElement(d)
```

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

```
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.Det() = Det(d)
RT d.LogDet(T* sign=0) = LogDet(d)
bool d.Singular()
dinv = d.Inverse() = Inverse(d)
d.Inverse(Matrix<T>& minv)
d.Inverse(DiagMatrix<T>& dinv)
d.InverseATA(Matrix<T>& cov)
d.InverseATA(DiagMatrix<T>& cov)
```

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

```
d.Zero()
d.SetAllTo(T x)
d.Clip(RT thresh)
d.SetToIdentity(T x = 1)
d.ConjugateSelf()
d.TransposeSelf() // null operation
d.InvertSelf()
Swap(d1,d2)
```

`InvertSelf` is new for `DiagMatrix` and calculates d^-1 in place. It is equivalent to `d = d.Inverse()`.

4.1.4 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
m2 = d * m1
m2 = m1 * d
d2 *= d1
m *= d
d2 = d1 [+-] x
d2 = x [+-] d1
d [+-]= x
d = x
m1 == m2
m1 != m2
```

4.1.5 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
```

There is only one allowed `DivType` for a `DiagMatrix`: `LU`. And, since it is also the default behavior, there is no reason to ever use this function. Furthermore, since a `DiagMatrix` is already a U matrix, the decomposition requires no work at all. Hence, it is always done in place; no extra storage is needed, and the methods `m.DivideInPlace()`, `m.SaveDiv()`, etc. are irrelevant.

If a `DiagMatrix` is singular, you can find out with `m.Singular()`, but there is no direct way to use SVD for the division and skip any divisions by 0. If you want to do this, you should use `BandMatrixViewOf(d)` to treat the `DiagMatrix` as a `BandMatrix`, which can use SVD.

4.1.6 Input/Output

The simplest output is the usual:

```
os << d
```

where `os` is any `std::ostream`. The output format is the same as for a `Matrix`, including all the 0's.

There is also a compact format:

```
d.WriteCompact(os)
```

which outputs in the format:

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

The same (compact, that is) format can be read back in the usual two ways:

```
tmv::DiagMatrix<T> d(n);
is >> d;
std::auto_ptr<tmv::DiagMatrix<T> > pd;
is >> pd;
```

where the first gives an error if `d` is the wrong size and the second allocates a new `DiagMatrix` that is the correct size.

One can also write small values as 0 with

```
m.Write(std::ostream& os, RT thresh)
m.WriteCompact(std::ostream& os, RT thresh)
```

4.2 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.

The class `UpperTriMatrix` inherits from `GenUpperTriMatrix`, and the class `LowerTriMatrix` inherits from `GenLowerTriMatrix`, both of which in turn inherit from `BaseMatrix`. The various views and composite classes described below also inherit from `GenUpperTriMatrix` and `GenLowerTriMatrix` as appropriate.

The `UpperTriMatrix` and `LowerTriMatrix` routines are included by:

```
#include "TMV_Tri.h"
```

In addition to the `T` template parameter, there are three other template parameters: `dt`, which can be either `tmv::UnitDiag` or `tmv::NonUnitDiag`; `stor`, which can be `tmv::RowMajor` or `tmv::ColMajor`; and `index`, which can be `tmv::CStyle` or `tmv::FortranStyle`. The default values for these template parameters are `NonUnitDiag`, `ColMajor`, and `CStyle` respectively.

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 allow for efficient storage of the matrices.

All of the routines are analogous for `UpperTriMatrix` and `LowerTriMatrix`, so we only list each routine once (the `UpperTriMatrix` version for definiteness).

4.2.1 Constructors

-
  ```
  tmv::UpperTriMatrix<T,dt,stor,index> U(size_t n)
  ```
  Makes an `n` $\times$ `n` `UpperTriMatrix` with uninitialized values. If debugging is turned on (i.e. not turned off with -DNDEBUG), then the values are in fact initialized to 888.

-
  ```
  tmv::UpperTriMatrix<T,dt,stor,index> U(size_t n, T x)
  ```
  Makes an `n` $\times$ `n` `UpperTriMatrix` with all values equal to `x`.

-
  ```
  tmv::UpperTriMatrix<T,dt,stor,index> U(size_t n, const T* vv)
  tmv::UpperTriMatrix<T,dt,stor,index> U(size_t n,
        const std::vector<T>& vv)
  ```
  Makes an `UpperTriMatrix` which copies the elements of `vv`.

-
  ```
  tmv::UpperTriMatrix<T,dt,stor,index> U(const GenMatrix<T>& m)
  tmv::UpperTriMatrix<T,dt,stor,index> U(const GenUperTriMatrix<T>& U2)
  ```
  Make an `UpperTriMatrix` which copies the corresponding values of `U2`. Note that the second one is allowed to have `U2` be `NonUnitDiag` but `dt = UnitDiag`, in which case only the off-diagonal elements are copied. The converse would set the diagonal of the new `UpperTriMatrix` to all 1's.

-
  ```
  tmv::UpperTriMatrix<T> U1(const GenUpperTriMatrix<T2>& 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::UpperTriMatrixView<T,index> U =
        UpperTriMatrixViewOf(MatrixView<T,I> m, DiagType dt)
  tmv::ConstUpperTriMatrixView<T,index> U =
        UpperTriMatrixViewOf(ConstMatrixView<T,I> m, DiagType dt)
  tmv::UpperTriMatrixView<T,index> U =
        UpperTriMatrixViewOf(UpperTriMatrixView<T,I> U2,
        DiagType dt)
  tmv::ConstUpperTriMatrixView<T,index> U =
        UpperTriMatrixViewOf(ConstUpperTriMatrixView<T,I> U2,
        DiagType dt)
  ```
  Make an `UpperTriMatrixView` of the corresponding portion of `m`. The last two provide a way to re-view a `NonUnitDiag` `UpperTriMatrix` as `UnitDiag`.

-
  ```
  tmv::UpperTriMatrixView<T,index> U =
        tmv::UpperTriMatrixViewOf(T* vv, size_t n,
  ```

```
          StorageType stor, DiagType dt=NonUnitDiag)
    tmv::ConstUpperTriMatrixView<T,index> U =
          tmv::UpperTriMatrixViewOf(const T* vv, size_t n,
          StorageType stor, DiagType dt=NonUnitDiag)
```

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 `n` × `n`, so all of the lower triangle elements must be in memory, even though they are never used.

4.2.2 Access

```
U.nrows() = U.ncols() = U.colsize() = U.rowsize() = U.size()
U(i,j)
U.row(int i, int j1, int j2)
U.col(int i, int j1, int j2)
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 `dt` is `UnitDiag`, then the range may not include the diagonal element. Similarly, `U.diag()` is valid only if `dt` is `NonUnitDiag`.

```
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`, except that all of the elements in the subvector or submatrix must be completely within the upper or lower triangle, as appropriate. If `dt` 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` ≠ 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.Transpose()
U.Conjugate()
U.Adjoint()
U.View()
U.Real()
U.Imag()
```

Note that the transpose and adjoint of an `UpperTriMatrix` is an `LowerTriMatrixView` and vice versa.

4.2.3 Functions

```
RT U.Norm1() = Norm1(U)
RT U.Norm2() = Norm2(U) = U.DoNorm2()
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)
T U.Trace() = Trace(U)
T U.Det() = Det(U)
RT U.LogDet(T* sign=0) = LogDet(U)
bool U.Singular()
Uinv = U.Inverse() = Inverse(U)
U.Inverse(Matrix<T>& minv)
U.Inverse(UpperTriMatrix<T>& Uinv)
U.InverseATA(Matrix<T>& cov)
bool U.Singular()
```

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.Inverse(minv)` for both argument types. Of course, similar statements hold for `LowerTriMatrix` as well.

```
U.Zero()
U.SetAllTo(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()`.

4.2.4 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
m2 = U * m1
m2 = m1 * U
m = U * L
m = L * U
U2 *= U1
m *= U
U2 = U1 [+-] x
U2 = x [+-] U1
U [+-]= x
U1 == U2
U1 != U2
```

4.2.5 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
```

There is only one allowed `DivType` for an `UpperTriMatrix` or a `LowerTriMatrix`: `LU`. And, since it is also the default behavior, there is no reason to ever specify it. Furthermore, as with a `DiagMatrix`, the decomposition requires no work at all. In fact, the ease of dividing by a upper or lower triangle matrix is precisely why the LU decomposition is useful. Hence, it is always done in place. i.e. no extra storage is needed, and all of the `m.DivideInPlace()`, `m.SaveDiv()`, etc. are irrelevant.

If an `UpperTriMatrix` or `LowerTriMatrix` is singular, you can check easily with `m.Singular()`, but there is no direct way to use SVD for the division and avoid any divisions by 0. If you want to do this use `BandMatrixViewOf(m)` to treat the `TriMatrix` as a `BandMatrix`, which can use SVD.

4.2.6 Input/Output

The simplest output is the usual:

```
os << U << L
```

where `os` is any `std::ostream`. The output format is the same as for a `Matrix`, including all the 0's.

There is also a compact format. For an `UpperTriMatrix`,

```
U.WriteCompact(os)
```

outputs in the format:

```
U n
( U(0,0)  U(0,1)  U(0,2)  ...  U(0,n-1) )
( U(1,1)  U(1,2)  ...  U(1,n-1) )
...
( U(n-1,n-1) )
```

For a `LowerTriMatrix`,

```
L.WriteCompact(os)
```

outputs in the format:

```
L n
( L(0,0) )
( L(1,0)  L(1,1) )
...
( L(n-1,0)  L(n-1,1) ... L(n-1,n-1) )
```

In each case, the compact format can be read back in the usual two ways:

```
tmv::UpperTriMatrix<T> U(n);
tmv::LowerTriMatrix<T> L(n);
is >> U >> L;
std::auto_ptr<tmv::UpperTriMatrix<T> > pU;
std::auto_ptr<tmv::LowerTriMatrix<T> > pL;
is >> pU >> pL;
```

One can write small values as 0 with

```
m.Write(std::ostream& os, RT thresh)
m.WriteCompact(std::ostream& os, RT thresh)
```

4.3 Band 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 10×5 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).

`BandMatrix` inherits from `GenBandMatrix`, which in turn inherits from `BaseMatrix`. The various views and composite classes described below also inherit from `GenBandMatrix`.

All the `BandMatrix` routines are included by:

```
#include "TMV_Band.h"
```

In addition to the `T` template parameter, we also have `stor` to indicate which storage you want to use, and the usual `index` parameter. For this class, we add an additional storage possibility: along with `RowMajor` and `ColMajor` storage, a `BandMatrix` may also be `DiagMajor`, which has unit step along the diagonals. The `index` parameter has the usual options of `CStyle` or `FortranStyle`. The default values for `stor` and `index` are `ColMajor` and `CStyle`.

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 6×6 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.

$$\text{ColMajor:} \begin{pmatrix} 0 & 5 & 10 & 15 & & \\ 1 & 6 & 11 & 16 & 21 & \\ 2 & 7 & 12 & 17 & 22 & 27 \\ & 8 & 13 & 18 & 23 & 28 \\ & & 14 & 19 & 24 & 29 \\ & & & 20 & 25 & 30 \end{pmatrix}$$

$$\text{RowMajor:} \begin{pmatrix} 0 & 1 & 2 & 3 & & \\ 5 & 6 & 7 & 8 & 9 & \\ 10 & 11 & 12 & 13 & 14 & 15 \\ & 16 & 17 & 18 & 19 & 20 \\ & & 22 & 23 & 24 & 25 \\ & & & 28 & 29 & 30 \end{pmatrix}$$

$$\text{DiagMajor:} \begin{pmatrix} 0 & 6 & 12 & 18 & & \\ -5 & 1 & 7 & 13 & 19 & \\ -10 & -4 & 2 & 8 & 14 & 20 \\ & -9 & -3 & 3 & 9 & 15 \\ & & -8 & -2 & 4 & 10 \\ & & & -7 & -1 & 5 \end{pmatrix}$$

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:

```
size_t BandStorageLength(StorageType stor, size_t nrows, size_t 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. So for the constructors that take the matrix information from an array (`T*` or `vector<T>`), the start of the array needs to be at the start of the lowest sub-diagonal.

4.3.1 Constructors

- `tmv::BandMatrix<T,stor,index> b(size_t nrows, size_t ncols,`
  `int nlo, int nhi)`

  Makes a `BandMatrix` with `nrows` rows, `ncols` columns, `nlo` sub-diagonals, and `nhi` super-diagonals with uninitialized values. If debugging is turned on (i.e. not turned off with -DNDEBUG), then the values are initialized to 888.

-
  ```
  tmv::BandMatrix<T,stor,index> b(size_t nrows, size_t ncols,
        int nlo, int nhi, T x)
  ```
  Makes a `BandMatrix` with all values equal to `x`.

-
  ```
  tmv::BandMatrix<T,stor,index> b(size_t nrows, size_t ncols,
        int nlo, int nhi, const T* vv)
  tmv::BandMatrix<T,stor,index> b(size_t nrows, size_t ncols,
        int nlo, int nhi, const std::vector<T>& vv)
  ```
  Makes a `BandMatrix` which copies the elements from `vv`. See the discussion above about the different storage types to see what order these elements should be. The function `BandStorageLength` will tell you how long `vv` must be. The elements that don't fall in the bounds of the actual matrix are not used and may be left undefined.

-
  ```
  tmv::BandMatrix<T,stor,index> b(const GenMatrix<T>& m,
        int nlo, int nhi)
  tmv::BandMatrix<T,stor,index> b(const GenBandMatrix<T>& m,
        int nlo, int nhi)
  tmv::BandMatrix<T,stor,index> b(const GenUpperTriMatrix<T>& m,
        int nhi)
  tmv::BandMatrix<T,stor,index> b(const GenLowerTriMatrix<T>& 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 GenVector<T>& v1, const GenVector<T>& v2)
  tmv::BandMatrix<T,tmv::DiagMajor> m = LowerBiDiagMatrix(
        const GenVector<T>& v1, const GenVector<T>& v2)
  tmv::BandMatrix<T,tmv::DiagMajor> m = TriDiagMatrix(
        const GenVector<T>& v1,  const GenVector<T>& v2,
        const GenVector<T>& v3)
  ```
  Shorthand to create bi- or tri-diagonal `BandMatrix`es if you already have the `Vector`s. The `Vector`s are in order from bottom to top in each case.

-
  ```
  tmv::BandMatrix<T,stor,index> b1(const GenBandMatrix<T2>& 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,index> b =
        BandMatrixViewOf(MatrixView<T,index> m, int nlo, int nhi)
  tmv::BandMatrixView<T,index> b =
        BandMatrixViewOf(BandMatrixView<T,index> m, int nlo, int nhi)
  tmv::BandMatrixView<T,index> b =
        BandMatrixViewOf(DiagMatrixView<T,index> m)
  tmv::BandMatrixView<T,index> b =
        BandMatrixViewOf(UpperTriMatrixView<T,index> m)
  tmv::BandMatrixView<T,index> b =
        BandMatrixViewOf(UpperTriMatrixView<T,index> m, int nhi)
  ```

```
tmv::BandMatrixView<T,index> b =
      BandMatrixViewOf(LowerTriMatrixView<T,index> m)
tmv::BandMatrixView<T,index> b =
      BandMatrixViewOf(LowerTriMatrixView<T,index> m, int nlo)
```

Make an `BandMatrixView` of the corresponding portion of `m`. There are also `ConstBandMatrixView` versions of all of these.

-
  ```
  tmv::BandMatrixView<T,index> b(MatrixView<T,index> m,
        int nlo, int nhi)
  tmv::BandMatrixView<T,index> b(BandMatrixView<T,index> 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 10×8, then

  ```
  tmv::BandMatrixView<T> b1(m,0,2);
  ```

  will create a 10×8 `BandMatrixView` of `m`'s diagonal plus two super-diagonals, but

  ```
  tmv::BandMatrixView<T> b2 = BandMatrixViewOf(m,0,2);}
  ```

  will instead create an 8×8 `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 10×8 `BandMatrix`, but

  ```
  tmv::BandMatrix<T> b2 = BandMatrixViewOf(m,0,2);}
  ```

  will create an 8×8 `BandMatrix`.

-
  ```
  tmv::BandMatrixView<T> b =
        tmv::BandMatrixViewOf(T* vv, size_t ncols, size_t nrows,
            int nlo, int nhi, StorageType stor)
  tmv::ConstBandMatrixView<T> b =
        tmv::BandMatrixViewOf(const T* vv, size_t ncols, size_t nrows,
            int nlo, int nhi, StorageType stor)
  ```

  Make a `BandMatrixView` of the actual memory elements, `vv`.

4.3.2 Access

```
b.nrows() = b.colsize()
b.ncols() = b.rowsize()
b(i,j)
b.row(int i, int j1, int j2)
b.col(int i, int j1, int j2)
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`.

```
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`, except that the entire subvector or submatrix must be completely within the band.

```
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` of a subset of a `BandMatrix`. The `newnlo` and `newnhi` parameters do not need to be different from the existing `nlo` and `nhi` if `i1 = j1` (i.e. the new main diagonal is part of the old main diagonal). However, if you are moving the upper left corner off of the diagonal, you need to adjust `nlo` and `nhi` appropriately. For example, if `b` is a 6×6 `BandMatrix` with 2 sub-diagonals and 3 super-diagonals (like our example above), the 3 super-diagonals may be viewed as a `BandMatrixView` with `b.SubBandMatrix(0,5,1,6,0,2)`.

```
b.Rows(int i1,int i2)
b.Cols(int j1,int j2)
b.Diags(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 6×6 `BandMatrix` with 2 sub- and 3 super-diagonals, we could instead use `m.Diags(1,4)`. The last 3 rows would be `m.Rows(3,6)`. Note that this wold be a 3×5 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 automatically set appropriately to include the entire band. (This is only non-trivial for non-square band matrices.) They are equivalent to `b.Diags(0,b.nhi()+1)` and `b.Diags(-b.nlo(),1)` respectively.

```
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 equivalent to `b.Diags(1,b.nhi()+1)` and `b.Diags(-b.nlo(),0)` respectively. 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.Real()
b.Imag()
```

4.3.3 Functions

```
RT b.Norm1() = Norm1(b)
RT b.Norm2() = Norm2(b) = b.DoNorm2()
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)
T b.Trace() = Trace(b)
T b.Det() = Det(b)
RT b.LogDet(T* sign) = LogDet(b)
minv = b.Inverse() = Inverse(b)
b.Inverse(Matrix<T>& minv)
b.InverseATA(Matrix<T>& cov)
```

The inverse of a `BandMatrix` is not (in general) banded. So `minv` here must be a regular `Matrix`.

```
b.Zero()
b.SetAllTo(T x)
b.Clip(RT thresh)
b.SetToIdentity(T x = 1)
b.ConjugateSelf()
b.TransposeSelf()
Swap(b1,b2)
```

4.3.4 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
m2 = b * m1
m2 = m1 * b
m *= b
b2 = b1 [+-] x
b2 = x [+-] b1
b [+-]= x
b = x
b1 == b2
b1 != b2
```

4.3.5 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 division decomposition:

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()
   int* b.LUD().GetP()
   ```

   The following should result in a matrix numerically very close to `b`.

   ```
   tmv::Matrix<T> m2 = b.LUD().GetL() * b.LUD().GetU();
   m2.ReversePermuteRows(b.LUD().GetP());
   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 convolutions need to be done to perform this in place as for the LU decomposition.

   To access this decomposition, use:

   ```
   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().GetL() * b.QRD().GetU();
m2v.ReversePermuteRows(b.QRD().GetP());
```

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.BandSVD().GetU()
   tmv::ConstDiagMatrixView<RT> b.BandSVD().GetS()
   tmv::ConstMatrixView<T> b.BandSVD().GetV()
   ```

   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:

   ```
   b.BandSVD().Thresh(RT thresh)
   b.BandSVD().Top(int nsing)
   RT b.BandSVD().Condition()
   int b.BandSVD().GetKMax()
   ```

The routines

```
b.SaveDiv()
b.SetDiv()
b.ReSetDiv()
b.UnSetDiv()
bool b.DivIsSet()
b.DivideInPlace()
```

work the same as for regular `Matrix`es.

4.3.6 Input/Output

The simplest output is the usual:

```
os << b
```

where `os` is any `std::ostream`. The output format is the same as for a `Matrix`.
There is also a compact format for a `BandMatrix`:

```
b.WriteCompact(os)
```

outputs in the format:

```
B nrows ncols nlo nhi
( b(0,0)  b(0,1)  b(0,2) ... b(0,nhi) )
( b(1,0)  b(1,1)  b(1,2) ... b(1,nhi+1) )
...
( b(nlo,0)  b(nlo,1) ...  b(nlo,nlo+nhi) )
...
( b(nrows-nhi-1,nrows-nlo-nhi-1) ... b(nrows-nhi-1,ncols-1) )
...
( b(nrows-1,nrows-nlo-1)  ... b(nrows-1,ncols-1) )
```

If nrows is not equal to ncols, then the above isn't exactly accurate. But the essence is the same: all the values in the band from each row are output one row at a time.

The same compact format can be read back in the usual two ways:

```
tmv::BandMatrix<T> b(nrows,ncols,nlo,nhi);
is >> b;
std::auto_ptr<tmv::BandMatrix<T> > pb;
is >> pb;
```

One can write small values as 0 with

```
b.Write(std::ostream& os, RT thresh)
b.WriteCompact(std::ostream& os, RT thresh)
```

4.4 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.

Both classes inherit from GenSymMatrix, which has an internal parameter to keep track of whether it is symmetric or hermitian. GenSymMatrix in turn inherits from BaseMatrix. The various views and composite classes described below also inherit from GenSymMatrix.

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 T template parameter, there are three other template parameters: uplo, which can be either tmv::Upper or tmv::Lower; stor, which can be either tmv::RowMajor or tmv::ColMajor; and index, which can be either tmv::CStyle or tmv::FortranStyle. The parameter uplo 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 values for these are Lower, ColMajor, and CStyle respectively.

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 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.

4.4.1 Constructors

- `tmv::SymMatrix<T,uplo,stor,index> s(size_t n)`
  `tmv::HermMatrix<T,uplo,stor,index> s(size_t n)`

  Makes an n × n SymMatrix or HermMatrix with uninitialized values. If debugging is turned on (i.e. not turned off with -DNDEBUG), then the values are in fact initialized to 888.

- `tmv::SymMatrix<T,uplo,stor,index> s(size_t n, T x)`
  `tmv::HermMatrix<T,uplo,stor,index> s(size_t 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,uplo,stor,index> s(size_t n, const T* vv)`
  `tmv::SymMatrix<T,uplo,stor,index> s(size_t n,`
  `      const std::vector<T>& vv)`
  `tmv::HermMatrix<T,uplo,stor,index> s(size_t n, const T* vv)`
  `tmv::HermMatrix<T,uplo,stor,index> s(size_t n,`
  `      const std::vector<T>& vv)`

  Makes a `SymMatrix` or `HermMatrix` which copies the elements from `vv`.

- `tmv::SymMatrix<T,uplo,stor,index> s(const GenMatrix<T>& m)`
  `tmv::HermMatrix<T,uplo,stor,index> s(const GenMatrix<T>& m)`
  `tmv::SymMatrix<T,uplo,stor,index> s(const GenDiagMatrix<T>& d)`
  `tmv::HermMatrix<T,uplo,stor,index> s(const GenDiagMatrix<T>& d)`

  Makes a `SymMatrix` or `HermMatrix` which copies the corresponding values of `m`.

- `tmv::SymMatrix<T,uplo,stor,index> s1(const GenSymMatrix<T2>& s2)`
  `tmv::HermMatrix<T,uplo,stor,index> s1(const GenSymMatrix<T2>& s2)`
  `s1 = s2`

  Copy the `GenSymMatrix s2`, 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. If `T` and `T2` are complex, then `m1` and `m2` need to be either both symmetric or both hermitian.

- `tmv::SymMatrixView<T,index> s =`
  `      tmv::SymMatrixViewOf(MatrixView<T,index> m, UpLoType uplo)`
  `tmv::ConstSymMatrixView<T,index> s =`
  `      tmv::SymMatrixViewOf(ConstMatrixView<T,index> m, UpLoType uplo)`
  `tmv::SymMatrixView<T,index> s =`
  `      tmv::HermMatrixViewOf(MatrixView<T,index>& m, UpLoType uplo)`
  `tmv::ConstSymMatrixView<T,index> s =`
  `      tmv::HermMatrixViewOf(ConstMatrixView<T,index>& m, UpLoType uplo)`

  Make a `SymMatrixView` of the corresponding portion of `m`.

- `tmv::SymMatrixView<T,index> s =`
  `      tmv::SymMatrixViewOf(T* vv, size_t n, UpLoType uplo,`
  `      StorageType stor)`
  `tmv::ConstSymMatrixView<T,index> s =`
  `      tmv::SymMatrixViewOf(const T* vv, size_t n, UpLoType uplo,`
  `      StorageType stor)`
  `tmv::SymMatrixView<T,index> s =`
  `      tmv::HermMatrixViewOf(T* vv, size_t n, UpLoType uplo,`
  `      StorageType stor)`
  `tmv::ConstSymMatrixView<T,index> s =`
  `      tmv::HermMatrixViewOf(const T* vv, size_t 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 $n \times n$, even though only about half of the values are actually used,

4.4.2 Access

```
s.nrows() = s.ncols() = s.colsize() = s.rowsize() = s.size()
s(i,j)
```

```
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) = ...
```

```
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`, except that the entire subvector or submatrix must be completely within 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)`. 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)
```

Both of these are valid, regardless of which triangle stores the actual data for `s`.

```
s.Transpose()
s.Conjugate()
s.Adjoint()
s.View()
s.Real()
s.Imag()
```

Note that the imaginary part of a complex hermitian matrix is skew-symmetric, so `s.Imag()` is illegal for a `HermMatrix`. If you need to deal with the imaginary part of a `HermMatrix`, you would have to view them as a triangle matrix with `s.UpperTri().Imag()`. Or, since the diagonal elements are all real. you could also use `s.UpperTri().OffDiag().Imag()`.

4.4.3 Functions

```
RT s.Norm1() = Norm1(s)
RT s.Norm2() = Norm2(s) = s.DoNorm2()
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)
T s.Trace() = Trace(s)
T s.Det() = Det(s)
RT s.LogDet(T* sign=0) = LogDet(s)
bool s.Singular()
sinv = s.Inverse() = Inverse(s)
s.Inverse(Matrix<T>& sinv)
s.Inverse(SymMatrix<T>& sinv)
s.InverseATA(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.Inverse(minv)` for both argument types.

```
s.Zero()
s.SetAllTo(T x)
s.Clip(RT thresh)
s.SetToIdentity(T x = 1)
s.ConjugateSelf()
s.TransposeSelf()
Swap(s1,s2)
```

`TransposeSelf` does nothing to a `SymMatrix` and is equivalent to `ConjugateSelf` for a `HermMatrix`.

```
s.SwapRowsCols(int i1, int i2)
s.PermuteRowsCols(const int* p)
s.ReversePermuteRowsCols(const int* p)
s.PermuteRowsCols(const int* p, int i1, int i2)
s.ReversePermuteRowsCols(const int* p, 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. The permute methods performs a series of these combined swaps.

4.4.4 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
m2 = s * m1
m2 = m1 * s
m *= s
s2 = s1 [+-] x
s2 = x [+-] s1
s [+-]= x
s = x
s1 == s2
s1 != s2
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()`.

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. This can become an issue when doing complicated arithmetic operations with complex hermitian or symmetric matrices.

Some operations with a `GenSymMatrix`, as far as the compiler can tell, may or may not result in another `SymMatrix`. For example, a complex scalar times a complex `HermMatrix` is not hermitian (or symmetric), but a complex scalar times a `SymMatrix` is still symmetric. Likewise the sum of two `GenSymMatrix`es may or may not be hermitian or symmetric.

So the complex `SymMatrixComposite` class[4], which is the return type of these and similar operations, is derived from `GenMatrix` rather than from `GenSymMatrix`. This means that if you let it self-instantiate, rather than assign it to a `SymMatrix`, it will instantiate as a regular `Matrix`. However, it is assignable to a `SymMatrix`, despite deriving from `GenMatrix`, so if your operation should be allowed, then the assignment will work fine.

Most of the time, this won't matter, since you will generally assign the result to either a `SymMatrix` or a `Matrix` as appropriate. 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. (The `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`:

[4]Note: all of these comments only apply to complex `SymMatrix` and `HermMatrix` arithmetic. The real version of the `SymMatrixComposite` does derive from `GenSymMatrix`, since there is no ambiguity in that case.

```
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;
```

4.4.5 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()
   Matrix<T> s.LUD().GetD()
   int* s.LUD().GetP()
   ```

   The following should result in a matrix numerically very close to `s`.

   ```
   Matrix<T> m2 = s.LUD().GetL() * s.LUD().GetD() *
         s.LUD().GetL().Transpose();
   m2.ReversePermuteRows(s.LUD().GetP());
   m2.ReversePermuteCols(s.LUD().GetP());
   ```

   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 §5.4 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,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().GetV()
```

   The following should result in a matrix numerically very close to `s`.

```
Matrix<T> m2 = s.SVD().GetU() * s.SVD().GetS() * s.SVD().GetV()
```

   For a complex symmetric `s`, the situation is not as convenient. In principle, one could find a decomposition $s = USU^T$ where S is a complex diagonal matrix, but such a decomposition is not easy to find. So for complex symmetric matrices, we just do the normal SVD: $s = USV$, although the algorithm does use the symmetry of the matrix to speed up portions of the algorithm relative that that for a generic matrix.

   The access is also necessarily different, since the object returned by `s.SVD()` implicitly assumes that `V = U.Adjoint()` (modulo some sign changes), so we need a new accessor: `s.SymSVD()`. Its `GetS` and `GetV` methods return Views rather than instantiated matrices.

   Both versions also have the same control and access routines as a regular SVD:

```
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()
s.DivIsSet()
s.DivideInPlace()
```

work the same as for regular `Matrix`es.

4.4.6 Input/Output

The simplest output is the usual:

```
os << s
```

where `os` is any `std::ostream`. The output format is the same as for a `Matrix`.

There is also a compact format for a `SymMatrix`:

```
s.WriteCompact(os)
```

outputs in the format:

```
H/S n
( s(0,0) )
( s(1,0)  s(1,1) )
...
( s(n-1,0)  s(n-1,1) ... s(n-1,n-1) )
```

where `H/S` means either the character `H` or `S`, which indicates whether the matrix is hermitian or symmetric. In each case, the same compact format can be read back in the usual two ways:

```
tmv::SymMatrix<T> s(n);
tmv::HermMatrix<T> h(n);
is >> s >> h;
std::auto_ptr<tmv::SymMatrix<T> > ps;
std::auto_ptr<tmv::HermMatrix<T> > ph;
is >> ps >> ph;
```

One can write small values as 0 with

```
s.Write(std::ostream& os, RT thresh)
s.WriteCompact(std::ostream& os, RT thresh)
```

4.5 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 $m = m^\dagger$.

Both classes inherit from `GenSymBandMatrix`, which in turn inherits from `BaseMatrix`. The various views and composite classes described below also inherit from `GenSymBandMatrix`.

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 T template parameter, there are three other template parameters: uplo, which can be either tmv::Upper or tmv::Lower; stor, which can be one of tmv::RowMajor, tmv::ColMajor, or tmv::DiagMajor; and index, which can be either tmv::CStyle or tmv::FortranStyle. The default values for these are Lower, ColMajor, CStyle.

The storage size required is the same as for the BandMatrix of the upper or lower band portion. 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. So for the constructors that take the matrix information from an array (T* or vector<T>), the start of the array needs to be at the start of the lowest sub-diagonal.

4.5.1 Constructors

-
  ```
  tmv::SymBandMatrix<T,uplo,stor,index> sb(size_t n, int nlo)
  ```
  Makes an n × n SymBandMatrix with nlo off-diagonals and with uninitialized values. If debugging is turned on (i.e. not turned off with -DNDEBUG), then the values are initialized to 888.

-
  ```
  tmv::SymBandMatrix<T,uplo,stor,index> sb(size_t n, int nlo, T x)
  ```
  Makes an n × n SymBandMatrix with nlo off-diagonals and with all values equal to x.

-
  ```
  tmv::SymBandMatrix<T,uplo,stor,index> sb(size_t n, int nlo,
         const T* vv)
  tmv::SymBandMatrix<T,uplo,stor,index> sb(size_t n, int nlo,
         const std::vector<T>& vv)
  ```
  Makes a SymBandMatrix which copies the elements from vv.

-
  ```
  tmv::SymBandMatrix<T,uplo,stor,index> sb(const GenMatrix<T>& m,
         int nlo)
  tmv::SymBandMatrix<T,uplo,stor,index> sb(const GenSymMatrix<T>& m,
         int nlo)
  tmv::SymBandMatrix<T,uplo,stor,index> sb(const GenBandMatrix<T>& m,
         int nlo)
  tmv::SymBandMatrix<T,uplo,stor,index> sb(const GenSymBandMatrix<T>& 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.

- `tmv::SymBandMatrix<T,uplo,DiagMajor> sb = SymTriDiagMatrix<uplo>(`
  `      const GenVector<T>& v1, const GenVector<T>& v2)`
  `tmv::SymBandMatrix<T,uplo,DiagMajor> sb = SymTriDiagMatrix<uplo>(`
  `      const GenVector<T>& v1, const GenVector<T>& v2)`

  Shorthand to create a symmetric tri-diagonal `BandMatrix` if you already have the `Vectors`. The main diagonal is `v1` and the off-diagonal is `v2`.

- `tmv::SymBandMatrix<T> sb1(const GenSymBandMatrix<T2>& 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> sb =
        SymBandMatrixViewOf(MatrixView<T> m, UpLoType uplo, int nlo)
  tmv::SymBandMatrixView<T> sb =
        SymBandMatrixViewOf(SymMatrixView<T> m, int nlo)
  tmv::SymBandMatrixView<T> sb =
        SymBandMatrixViewOf(BandMatrixView<T> m, UpLoType uplo,
        int nlo = (uplo==Upper ? m.nhi() : n.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> sb =
        tmv::SymBandMatrixViewOf(T* vv, size_t n, int nlo,
            UpLoType uplo, StorageType stor)
  tmv::ConstSymBandMatrixView<T> sb =
        tmv::SymBandMatrixViewOf(const T* vv, size_t n, int nlo,
            UpLoType uplo, StorageType stor)
  tmv::SymBandMatrixView<T> sb =
        tmv::HermBandMatrixViewOf(T* vv, size_t n, int nlo,
            UpLoType uplo, StorageType stor)
  tmv::ConstSymBandMatrixView<T> sb =
        tmv::HermBandMatrixViewOf(const T* vv, size_t n, int nlo,
            UpLoType uplo, StorageType stor)
  ```

  Make a `SymBandMatrixView` of the actual memory elements, `vv`.

4.5.2 Access

```
sb.nrows() = sb.ncols() = sb.colsize() = sb.rowsize() = sb.size()
sb.nlo() = sb.nhi()
sb(i,j)
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.

```
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, int newnlo, int newhi)
sb.SubBandMatrix(int i1, int i2, int j1, int j2, int newnlo, int newhi,
      int istep, int jstep)
sb.Diags(int k1, int k2)
sb.UpperBand()
sb.LowerBand()
sb.UpperBandOff()
sb.LowerBandOff()
```

These work the same as for a `BandMatrix`, 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.SymDiags(int newnlo)
```

Since `Diags` 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.Real()
sb.Imag()
```

Note that the imaginary part of a complex hermitian band matrix is skew-symmetric, so `sb.Imag()` is illegal for a `HermBandMatrix`. If you need to manipulate the imaginary part of a `HermMatrix`, you could use `sb.UpperBandOff().Imag()` (since all the diagonal elements are real).

4.5.3 Functions

```
RT sb.Norm1() = Norm1(sb)
RT sb.Norm2() = Norm2(sb) = sb.DoNorm2()
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)
T sb.Trace() = Trace(sb)
T sb.Det() = Det(sb)
RT sb.LogDet(T* sign) = LodDet(sb)
sinv = sb.Inverse() = Inverse(sb)
sb.Inverse(Matrix<T>& minv)
sb.Inverse(SymMatrix<T>& sinv)
sb.InverseATA(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.Zero()
sb.SetAllTo(T x)
sb.Clip(RT thresh)
sb.SetToIdentity(T x = 1)
sb.ConjugateSelf()
sb.TransposeSelf()
Swap(sb1,sb2)
```

4.5.4 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
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
sb1 == sb2
sb1 != sb2
```

4.5.5 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()
   int* 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().GetL() * sb.LUD().GetU();
   m2v.ReversePermuteRows(sb.LUD().GetP());
   ```

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 Bunch-Kaufman 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().GetV()
   ```

   (As for `SymMatrix`, a complex symmetric matrix needs to use the accessor `SymSVD()` instead, whose `GetS` and `GetV` 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().GetV()
   ```

   Both versions also have the same control and access routines as a regular SVD:

   ```
   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()
sb.DivideInPlace()
```

work the same as for regular `Matrix`es.

4.5.6 Input/Output

The simplest output is the usual:

```
os << sb
```

where `os` is any `std::ostream`. The output format is the same as for a `Matrix`.

There is also a compact format for a `BandMatrix`:

```
sb.WriteCompact(os)
```

outputs in the format:

```
hB/sB n nlo
( sb(0,0)  )
( sb(1,0)  sb(1,1)  )
...
( sb(nlo,0)  sb(nlo,1) ...  sb(nlo,nlo) )
( sb(nlo+1,1)  sb(nlo+1,2) ...  sb(nlo+1,nlo+1) )
...
( sb(n-1,n-nlo-1)  ... sb(n-1,n-1) )
```

where `hB/sB` means either `hB` or `sB`, which indicates whether the matrix is hermitian or symmetric.

The same compact format can be read back in the usual two ways:

```
tmv::SymBandMatrix<T> sb(n,nlo);
tmv::HermBandMatrix<T> hb(n,nlo);
is >> sb >> hb;
std::auto_ptr<tmv::SymBandMatrix<T> > psb;
std::auto_ptr<tmv::HermBandMatrix<T> > phb;
is >> psb >> phb;
```

One can write small values as 0 with

```
sb.Write(std::ostream& os, RT thresh)
sb.WriteCompact(std::ostream& os, RT thresh)
```

5 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.
- Calling `m.LUD()`, `m.QRD()`, etc. for a `Matrix` that does not have that decomposition set (and saved).

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 -DNDEBUG. 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, but these are only done if the code is compiled with the `-DXTEST` flag.

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.

These errors are always checked for even if -DNDEBUG 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 (i.e. without the `&`), 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`:

5.1 FailedAssert

`tmv::FailedAssert` 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 are absolutely sure that the assert failed due to a bug in the TMV code rather than your own code, please let me know at mike_jarvis@users.sourceforge.net.

5.2 Singular

`tmv::Singular` 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, and your results will just be unreliable.

5.3 ReadError

`tmv::ReadError` 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.

5.4 NonPosDef

`tmv::NonPosDef` 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 `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::GenSymMatrix<T>& m)
{
    assert(m.isherm());
    try {
        HermMatrix<T> m2 = m;
        CH_Decompose(m2.View());
    }
    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% 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 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 §6.4 below will also throw the exception `NonPosDef` when it fails.

6 Advanced Usage

6.1 Eigenvalues and eigenvectors

The eigenvalues of a matrix are important quantities in many matrix applications. A number, λ, is an eigenvalue of a square matrix, A, if for some non-zero vector v,

$$Av = \lambda v \tag{1}$$

in which case v is the called an eigenvector corresponding to the eigenvalue λ. 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

$$Av - \lambda v = 0 \tag{2}$$
$$(A - \lambda I)v = 0 \tag{3}$$
$$\det(A - \lambda I) = 0 \quad (\text{or } \; v = 0) \tag{4}$$

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[5]. 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 Λ to be a diagonal matrix with the values of λ along the diagonal, then we have (for non-defective matrices)

$$AV = V\Lambda \tag{5}$$

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,

$$A = V\Lambda V^{-1} \tag{6}$$
$$\Lambda = V^{-1}AV \tag{7}$$

This is known as "diagonalizing" the matrix A. The determinant and trace are preserved by this procedure, which implies two more properties of eigenvalues:

$$\det(A) = \prod_{k=1}^{N} \lambda_k \tag{8}$$
$$\text{tr}(A) = \sum_{k=1}^{N} \lambda_k \tag{9}$$

[5]The multiplicity of the eigenvalue is generally 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[6].

So far, the TMV library can only find the eigenvalues and eigenvectors of hermitian matrices. The routines to do so are

```
void Eigen(const GenSymMatrix<T>& A,
      const MatrixView<T>& V, const VectorView<RT>& lambda);

void Eigen(const GenSymBandMatrix<T>& A,
      const MatrixView<T>& V, const VectorView<RT>& lambda);
```

where `V.col(i)` is the eigenvector corresponding to each eigenvalue `lambda(i)`. The original matrix `A` can be obtained from

```
A = V * DiagMatrixViewOf(lambda) * V.Adjoint();
```

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 SymMatrixView<T>& A, const VectorView<RT>& lambda);

void Eigen(const GenSymBandMatrix<T>& A, const VectorView<RT>& lambda);
```

Note that the first one uses the input matrix `A` as workspace and destroys the input matrix in the process.

6.2 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 the routines below, the matrix being decomposed is input as `A`, and we list the routines for all of the allowed types for `A` for each kind of decomposition. If `A` is listed as a "`Gen`" type, such as `GenBandMatrix<T>`, then that means the input matrix is not changed by the decomposition routine. If `A` is listed as a "`View`" type, such as `BandMatrixView<T>`, then that means the input matrix is changed.

In some cases where `A` is a `View` type, one of the decomposition components is returned in the location of `A`, or some part of it, overwriting the input matrix. In these cases, there will be a line indicating this after the function (e.g. `L = A.LowerTri()`). 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).

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.

Finally, a word about the permutations. In TMV, permutations are defined as a series of row or column swaps. I haven't made a `Permutation` class yet to make it easy to use these permutations. But the code snippets which show how to recreate the input matrices from the decompositions should be sufficient to describe how to use the permutations as given.

[6]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.

- LU Decomposition

  (Matrix, BandMatrix)

  $A \to PLU$ where L is lower triangular, U is upper triangular, and P is a permutation.

  ```
  void LU_Decompose(const MatrixView<T>& A, int* P);
  L = A.LowerTri(UnitDiag);
  U = A.UpperTri(NonUnitDiag);

  void LU_Decompose(const GenBandMatrix<T>& A,
        const LowerTriMatrixView<T>& L,
        const BandMatrixView<T>& U, int* P);
  ```

  In the second case, U must have U.nhi() = A.nlo()+A.nhi(), and L should be UnitDiag.

  The original matrix A can be obtained from:

  ```
  A = L * U;
  A.ReversePermuteRows(P);
  ```

- Cholesky Decomposition

  (HermMatrix, HermBandMatrix)

  $A \to LL^\dagger$, where L is lower triangular, and A is hermitian.

  ```
  void CH_Decompose(const SymMatrixView<T>& A);
  L = A.LowerTri();

  void CH_Decompose(const SymBandMatrixView<T>& A);
  L = A.LowerBand();
  ```

- Bunch-Kaufman Decomposition

  (HermMatrix, SymMatrix)

  If A is hermitian, $A \to PLDL^\dagger P^T$, and if A is symmetric, $A \to PLDL^T P^T$, 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 1×1 and 2×2 blocks, which means that there are no two consecutive non-zero elements along the off-diagonal.

  ```
  void LDL_Decompose(const SymMatrixView<T>& A,
        const SymBandMatrixView<T>& D, int* P);
  L = A.LowerTri(UnitDiag);
  ```

  The original matrix A can be obtained from:

  ```
  A = L * D * (A.isherm() ? L.Adjoint() : L.Transpose());
  A.ReversePermuteRows(P);
  A.ReversePermuteCols(P);
  ```

- Tridiagonal LDL† Decomposition

  (`HermBandMatrix`, `SymBandMatrix` with `nlo = 1`)

  This decomposition for symmetric or hermitian tri-diagonal matrices is similar to the Bunch-Kaufman decomposition: $A \rightarrow LDL^\dagger$ 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. Thus, this one does not have any 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.[7]

  ```
  void LDL_Decompose(const SymBandMatrixView<T>& A);
  (L = A.LowerBand()).diag().SetAllTo(T(1));
  D = DiagMatrixViewOf(A.diag());
  ```

  The original matrix `A` can be obtained from:

  ```
  A = L * D * (A.isherm() ? L.Adjoint() : L.Transpose());
  ```

- 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(const MatrixView<T>& A,
        const UpperTriMatrixView<T>& R);
  Q = A;

  void QR_Decompose(const GenBandMatrix<T>& A,
        const MatrixView<T>& Q, const BandMatrixView<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(const MatrixView<T>& A);
  R = A.UpperTri();

  void QR_Decompose(const GenBandMatrix<T>& A,
        const BandMatrixView<T>& R);
  ```

- QRP Decomposition

  (`Matrix`)

  $A \rightarrow QRP$, 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.

[7]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 gives an error 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.

```
void QRP_Decompose(const MatrixView<T>& A,
      const UpperTriMatrixView<T>& R, int* P, bool strict=false);
Q = A;
```

As discussed in §3.6.3, there are two slightly different algorithms for doing a QRP decomposition. If `strict = true`, then the diagonal elements of R be strictly decreasing (in absolute value) from upper-left to lower-right.

If `strict` is false, however (the default), 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 ϵ 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.

The original matrix `A` is obtained from:

```
A = Q * R;
A.ReversePermuteCols(P);
```

If you only need R, the following versions is faster, since it does not fully calculate Q.

```
void QRP_Decompose(const MatrixView<T>& A, bool strict=false);
R = A.UpperTri();
```

- Singular Value Decomposition

  (`Matrix`, `SymMatrix`, `HermMatrix`, `BandMatrix`, `SymBandMatrix`, `HermBandMatrix`)

  $A \rightarrow USV$, 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.[8]

```
void SV_Decompose(const MatrixView<T>& A,
      const DiagMatrixView<RT>& S, const MatrixView<T>& V);
U = A;

void SV_Decompose(const GenSymMatrix<T>& A,
      const MatrixView<T>& U, const DiagMatrixView<RT>& S,
      const MatrixView<T>& V);

void SV_Decompose(const GenBandMatrix<T>& A,
      const MatrixView<T>& U, const DiagMatrixView<RT>& S,
      const MatrixView<T>& V);

void SV_Decompose(const GenSymBandMatrix<T>& A,
      const MatrixView<T>& U, const DiagMatrixView<RT>& S,
      const MatrixView<T>& V);
```

[8]The SVD is more commonly written as $A \rightarrow USV^T$. As far as I can tell, this seems to be a holdover from the days of Fortran programming. In Fortran, matrices are stored in column-major format. Considering that the rows of what we call V are the singular vectors, also known as principal components, of A, it made more sense for Fortran programmers to use the transpose of V which has the principal components in the columns. This complication is unnecessary in TMV. If you want the principal components stored contiguously, just make V row-major. On the other hand, decomposition with column-major storage of V is usually a bit faster, so you need to make a choice appropriate for your particular program.

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(const MatrixView<T>& A,
      const DiagMatrixView<RT>& S, const MatrixView<T>& V, false);
// U != A

void SV_Decompose(const MatrixView<T>& A,
      const DiagMatrixView<RT>& S, bool StoreU);
if (StoreU) U = A;

void SV_Decompose(const SymMatrixView<T>& A,
      const DiagMatrixView<RT>& S);

void SV_Decompose(const GenSymMatrix<T>& A,
      const DiagMatrixView<RT>& S, const MatrixView<T>& V);

void SV_Decompose(const GenSymMatrix<T>& A,
      const MatrixView<T>& U, const DiagMatrixView<RT>& S);

void SV_Decompose(const GenBandMatrix<T>& A,
      const DiagMatrixView<RT>& S);

void SV_Decompose(const GenBandMatrix<T>& A,
      const DiagMatrixView<RT>& S, const MatrixView<T>& V);

void SV_Decompose(const GenBandMatrix<T>& A,
      const MatrixView<T>& U, const DiagMatrixView<RT>& S);

void SV_Decompose(const GenSymBandMatrix<T>& A,
      const DiagMatrixView<RT>& S);

void SV_Decompose(const GenSymBandMatrix<T>& A,
      const DiagMatrixView<RT>& S, const MatrixView<T>& V);

void SV_Decompose(const GenSymBandMatrix<T>& A,
      const MatrixView<T>& U, const DiagMatrixView<RT>& S);
```

- 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(const MatrixView<T>& A,
      const SymMatrixView<T>& P);
U = A;

void Polar_Decompose(const GenBandMatrix<T>& A,
      const MatrixView<T>& U, const SymMatrixView<T>& P);
```

- Matrix Square Root

  `(HermMatrix, HermBandMatrix)`

  $A \to SS$, where A and S are each positive definite hermitian matrices.

  ```
  void SquareRoot(const SymMatrixView<T>& A);
  S = A;

  void SquareRoot(const GenSymBandMatrix<T>& A,
        const SymMatrixView<T>& S);
  ```

  If A is found to be not positive definite, a `NonPosDef` exception is thrown.

6.3 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 §3.6.2 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.

$$A = \begin{pmatrix} A_0 \\ A_1 \end{pmatrix}$$

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:

$$A^\dagger b = A_1^\dagger b_1 + A_2^\dagger b_2 + ... \tag{10}$$

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(const UpperTriMatrixView<T>& R, const MatrixView<T>& M)
```

which updates the value of R such that $R_{\text{out}}^\dagger R_{\text{out}} = R_{\text{in}}^\dagger R_{\text{in}} + M^\dagger M$. (The input matrix `M` 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.View(),b.View());

// 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
int n1 = 0, n2 = n_mem;
while (n2 < n_full)
{
    n1 = n2; n2 += n_mem;
    if (n2 > n_full) n2 = n_full;
    // (Usually, A1==A, b1==b, but not the last time through the loop.)
    tmv::MatrixView<double> A1 = A.Rows(0,n2-n1);
    tmv::VectorView<double> b1 = b.SubVector(0,n2-n1);

    // Import the next bit:
    Import_Ab(n1,n2,A1,b1);

    // Update, x, R:
    x += A1.Transpose() * b1;
    QR_Update(R.View(),A1);
}

// Finish the solution:
x /= R.Transpose();
x /= R;
```

6.4 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(const UpperTriMatrixView<T>& R, const GenMatrix<T>& M)
```

`M` represents the rows from the original matrix to remove from the QR decomposition.

It is possible for the downdate to fail (and throw an exception) if the matrix M 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,M)` basically updates R to be the Cholesky decomposition of $A^\dagger A - M^\dagger M$. The procedure fails (and throws) when this latter matrix is found not to be positive definite.

It is worth pointing out that 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 that the former one. I have not seen this algorithm discussed anywhere (although I am sure that I am not the first to come up with it), so I'll provide a short summary here.

6.4.1 The update algorithm

First lets look at the Householder algorithm for QR update (since the downdate algorithm is intimately related):

Given the initial decomposition $A_0 = Q_0 R_0$, we want to find R such that

$$A = \begin{pmatrix} A_0 \\ A_1 \end{pmatrix} = Q_1 R$$
$$\begin{pmatrix} Q_0 R_0 \\ A_1 \end{pmatrix} = Q_1 R$$
$$\begin{pmatrix} Q_0 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} R_0 \\ A_1 \end{pmatrix} = Q_1 R$$

So if we perform a QR decomposition:

$$A_2 \equiv \begin{pmatrix} R_0 \\ A_1 \end{pmatrix} = Q_2 R$$

Then this is the R we want, and

$$Q_1 = \begin{pmatrix} Q_0 & 0 \\ 0 & 1 \end{pmatrix} Q_2$$

For the following discussion, let N be the number of rows (and columns) in R_0, and M be the number of rows in A_1.

To perform the decomposition, we multiply A_2 by a series of Householder reflections on the left to zero out each column of A_1 one at a time. Householder reflections are unitary, so their product in the reverse order is Q_2:

$$\begin{pmatrix} R \\ 0 \end{pmatrix} = H_N H_{N-1} ... H_2 H_1 \begin{pmatrix} R_0 \\ A_1 \end{pmatrix}$$
$$Q_2 = H_1^\dagger H_2^\dagger ... H_{N-1}^\dagger H_N^\dagger$$

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 β is real [9], they are also hermitian: $H = H^\dagger$.

H_1 is defined for the vector $x = (R_0(1,1), 0, 0, ..., 0, 0, A_1(1,1), A_1(2,1), ..., A_1(M,1))$ 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 β (or in general an arbitrary factor $e^{i\theta}$) which is chosen to minimize rounding errors.

The product $H_1 A_2$ "reflects" the first column of A_1 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 A_1 are changed. The subsequent Householder reflections are defined similarly, each zeroing out a column of A_1, and modifying the corresponding row of R_0 and the remaining elements of A_1.

At the end of this procedure, the matrix R_0 will be changed into the matrix R. If desired, Q may also be calculated in the process, but the TMV implementation of the QR update does not calculate Q.

6.4.2 The downdate algorithm

Given the initial decomposition

$$A = \begin{pmatrix} A_0 \\ A_1 \end{pmatrix} = Q_1 R$$

we want to find R_0 such that $A_0 = Q_0 R_0$.

[9] Unfortunately, LAPACK chose to implement complex Householder matrices with y chosen to be real, which means that β is complex. Hence H is not Hermitian, which complicates a lot of the calling routines which use Householder matrices. 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 their 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 β, but not for complex β. But we are stuck with their choice, so we allow β to be complex in this discussion.

The 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. 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. The A_1 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 A_1.

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

$$|R(0,0)|^2 = |R_0(0,0)|^2 + ||A_1(1:M,0)||^2$$

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 A_1. When we have finished this process, we are left with R_0 in the place of R.

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 A_1 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 A_1. 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 BLAS 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 BLAS calls, for a bit more speed-up.

6.5 Other SymMatrix 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:

```
tmv::Rank2Update<bool add>(T x, const GenVector<T1>& v1,
      const GenVector<T2>& v2, const SymMatrixView<T>& s)
tmv::Rank2KUpdate<bool add>(T x, const GenMatrix<T1>& m1,
      const GenMatrix<T2>& m2, const SymMatrixView<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:

```
tmv::SymMultMM<bool add>(T x, const GenMatrix<T>& m1,
       const GenMatrix<T>& m2, const SymMatrixView<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.

6.6 Element-by-element product

The two usual kinds of multiplication for vectors are the inner product and the outer product, which result in a scalar and a matrix respectively. However, occasionally you may want to multiply each element in a vector by the corresponding element in another vector: $v(i) = v(i) * w(i)$.

There are two functions that should provide all of this kind of functionality for you:

```
ElementProd(T x, const GenVector<T1>& v1, const VectorView<T>& v2);
AddElementProd(T x, const GenVector<T1>& v1, const GenVector<T2>& v2,
       const VectorView<T>& v3)
```

The first performs $v_2(i) = x * v_1(i) * v_2(i)$, and the second performs $v_3(i) = v_3(i) + x * v_1(i) * v_2(i)$ for $i = 0...(N-1)$ (where N is the size of the vectors).

There is no operator overloading for `Vectors` that would be equivalent to these expressions. But they are actually equivalent to the following:

```
v2 *= x * DiagMatrixViewOf(v1);
v3 += x * DiagMatrixViewOf(v1) * v2;
```

respectively. In fact, these statements inline to the above function calls automatically. Depending on you preference and the meanings of your vectors, these statements may or may not be clearer as to what you are doing.

There are also corresponding functions for `Matrix` and for each of the special matrix types:

```
ElementProd(T x, const GenMatrix<T1>& m1, const MatrixView<T>& m2);
AddElementProd(T x, const GenMatrix<T1>& m1, const GenMatrix<T2>& m2,
       const MatrixView<T>& m3);
```

Likewise for the other special matrix classes. The first performs $m_2(i,j) = x * m_1(i,j) * m_2(i,j)$, and the second performs $m_3(i,j) = m_3(i,j) + x * m_1(i,j) * m_2(i,j)$ for every i, j in the matrix.

These don't have any `DiagMatrixViewOf` version, since the corresponding concept would require a four-dimensional tensor, and the TMV library just deals with one- and two-dimensional objects.

The matrices all have to be the same size and shape, but can have any (i.e. not necessarily the same) storage method. However, the routines are faster if the matrices use the same storage.

6.7 BaseMatrix views

If you are dealing with objects that are only known to be `BaseMatrix`es (i.e. they could be a `Matrix` or a `DiagMatrix` or a `SymMatrix`, etc.), then methods like `m.Transpose()`, `m.View()`, and such can't know what kind of object to return. So these methods can't be defined for a `BaseMatrix`.

Instead, we have the following virtual methods, which are available to a `BaseMatrix` object and are defined in each specific kind of matrix to return a pointer to the right kind of object:

```
std::auto_ptr<tmv::BaseMatrix<T> > m.NewCopy()
std::auto_ptr<tmv::BaseMatrix<T> > m.NewView()
std::auto_ptr<tmv::BaseMatrix<T> > m.NewTranspose()
std::auto_ptr<tmv::BaseMatrix<T> > m.NewConjugate()
std::auto_ptr<tmv::BaseMatrix<T> > m.NewAdjoint()
std::auto_ptr<tmv::BaseMatrix<T> > m.NewInverse()
```

`NewCopy` and `NewInverse` create new storage to store a copy of the matrix or its inverse, respectively. The other four just return views of the current matrix.

6.8 Iterators

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

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

just like for standard library containers. The specific types to which these typedefs refer are:

```
tmv::VIt<T,tmv::Unit,tmv::NonConj>
tmv::CVIt<T,tmv::Unit,tmv::NonConj>
tmv::VIt<T,tmv::Step,tmv::NonConj>
tmv::CVIt<T,tmv::Step,tmv::NonConj>
```

respectively.

`VIt` is a mutable-iterator, and `CVIt` is a const-iterator. `Unit` indicates that the step size is 1, while `Step` allows for any step size between successive elements (and is therefore slower). For the reverse iterators, the step size is -1.

This can be worth knowing if you are going to be optimizing code that uses iterators of `VectorViews`. This is because their iterators are instead:

```
tmv::VIter<T>
tmv::CVIter<T>
```

which always check the step size (rather than assuming unit steps) and always keep track of a possible conjugation.

If you know that you are dealing with a 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. This indicates that the vector view really refers to the conjugates of the values stored in the actual memory locations.

Also, if you know that your view has unit steps between elements, converting to an iterator with `tmv::Unit` will iterate faster. It is often faster to check the step size once at the beginning of the routine and convert to a unit-step iterator if possible.

All of these conversions can be done with a simple cast or constructor, such as:

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

Regular `Vector`s are always `Unit` and `NonConj`, so those iterators are already fast without using the specific `VIt` names. That is, you can just use `Vector<T>::iterator` rather than `VIt<T,Unit,NonConj>` without any performance drop.

6.9 Direct memory access

We provide methods for accessing the memory of a matrix or vector directly. This is especially useful for meshing the TMV objects with other libraries (such as BLAS or LAPACK). But it can also be useful for writing some optimized code for a particular function.

The pointer to the start of the memory for a vector can be obtained by:

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

Using the direct memory access requires that you know the spacing of the elements in memory and (for views) whether the view is conjugated or not. So we also provide:

```
int v.step() const
bool v.isconj() const
```

For matrices, the corresponding routines actually return the upper-left element of the matrix. For some matrices, (e.g. `BandMatrix<T,DiagMatrix>`) this is not actually the first element in memory. We also need to know the step size in both directions:

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

The step in the "down" direction along a column is `stepi`, and the step to the "right" along a row is `stepj`. The last two check if a matrix is `RowMajor` or `ColMajor` respectively.

For band matrices, there are also:

```
int m.diagstep() const
bool m.isdm() const
```

which return the step along the diagonal and whether the matrix is `DiagMajor`.

For symmetric/hermitian matrices, there are some more methods:

```
bool m.isherm()
bool m.issym()
bool m.isupper()
```

The first two both return `true` for real symmetric matrices, but differentiate between hermitian and symmetric varieties for complex types. The last one tells you whether the actual elements to be accessed are stored in the upper triangle half of the matrix (true) or the lower (false).

6.10 "Linear" views

Our matrices generally store the data contiguously in memory with all of the methods like `row` and `col` returning the appropriate slice through the data. Occasionally, though, it can be useful to treat the whole matrix as a single vector of elements. We use this internally for implementing routines like `SetAllTo` and matrix addition, among others. These are faster than accessing the data in ways that use the actual matrix structure.

Anyway, this same kind of access may be useful for some users of the library, so the following two methods are available:

```
tmv::VectorView<T> m.LinearView()
tmv::ConstVectorView<T> m.ConstLinearView()
```

These return a view to the elements of a `Matrix` as a single vector. 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 above methods are legal.

The same methods are also defined for `BandMatrix` (and corresponding views). 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. (For example, one cannot compute things like the minimum or maximum element this way, since the undefined elements may have very large or small values which would corrupt this calculation.)

The triangular and symmetric matrices have too much memory that is not actually used by the matrix for these to be very useful, so we do not provide them. When we eventually implement the packed storage varieties, these methods will be provided for those.

Along the same lines is another method for a `Vector`:

```
tmv::VectorView<RT> v.Flatten()
```

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

This probably isn't very useful for most users either, but it is useful internally, since it allows code such as:

```
tmv::Vector<complex<double> > v(500);
[...]
v *= 2.3;
```

to call the BLAS routine `dscal` with `x=2.3`, rather than `zscal` with `x=complex<double>(2.3,0.0)`, which would be slower.

7 Obtaining and Compiling the Library

This code is licensed using the Gnu General Public License. See §10 below for more details.

7.1 Basic installation script

The following are step-by-step instructions on how to obtain the code for the library, set it up for your system, and compile it:

1. Go to http://sourceforge.net/projects/tmv-cpp/ for a link to a tarball with all of the source code, and copy it to the directory where you want to put the TMV library.

2. Unpack the tarball:
   ```
   gunzip tmv0.61.tar.gz
   tar xf tmv0.61.tar
   ```
   This will make a directory called tmv0.61 with the subdirectories: `doc`, `examples`, `include`, `lib`, `src`, `test` along with the files `INSTALL`, `README` and `Makefile` in the top directory.

3. Edit the makefile.

   The start of the makefile lists 4 things to specify: the compiler, the include directories, the other flags to send to the compiler, and any necessary BLAS/LAPACK linkage flags. The default setup is:

   ```
   CC= g++
   INCLUDE= -Iinclude
   CFLAGS= $(INCLUDE) -O -DNOBLAS -DNDEBUG
   BLASLIBS=
   ```

   but you will probably want to change this.

   This default setup will compile using g++ without any BLAS or LAPACK library and with debugging turned off. This setup should work on any system with gcc, although it almost certainly won't be as fast as using an optimized BLAS library and/or a LAPACK library.

   You should edit these so that `CC` is your desired compiler; `INCLUDE` specifies the directories for any BLAS and LAPACK header files you want to include (you should leave `-Iinclude` there as well); `CFLAGS` contains any compiler flags you want (see below for TMV-specific flags to consider); and `BLASLIBS` specifies the libraries required for linking your BLAS and LAPACK libraries.

   After these lines, there are several commented-out examples for different systems using various BLAS and LAPACK versions, showcasing some of the compiler options described below, and giving examples of what you need for several common (or at least representative) systems. If you have a system similar to one of these, then it should be a good starting point for you to figure out what you want to use.

   See the next section below for a complete list of compiler flags that control how the TMV library is built.

4. (advanced usage) Edit Inst and Blas files

   By default, the library will include instantiations of all classes and functions that use either `double` or `float` (including complex versions of each). There are a few flags, such as `-DNO_INST_FLOAT` and `-DINST_LONGDOUBLE`, that change this as described below. But if you want to compile routines for some other class, such as a user-defined `MyQuadPrecisionType` class, then you will need to modify the file `TMV_Inst.h` (in the `src` directory). You simply need to add the lines:

   ```
   #define T MyQuadPrecisionType
   #include InstFile
   #undef T
   ```

to the end of the file before compiling. (Obviously, replace `MyQuadPrecisionType` with whatever type you want to be instantiated in the library.)

Also, the file `TMV_Blas.h` (also in the `src` directory) sets up all the BLAS and LAPACK calling structures, as well as the necessary `#include` statements. So if the BLAS or LAPACK options aren't working for your system, you may need to edit these files as well. This is especially true if your BLAS or LAPACK versions are not one of ATLAS, CLAPACK, MKL, or ACML. The comments at the beginning of `TMV_Blas.h` gives instructions on how to set up the file for other installations.

5. Type:
   ```
   make libs
   ```
   This will make the TMV libraries, `libtmv.a` and `libtmv_symband.a`, which will be located in the directory lib.

   Warning: this step may take a long time to finish. So plan on letting this run for an hour or so (depending on your machine and compiler of course). Don't worry - once the library is compiled, compiling your code using the library is pretty quick.

6. (optional) Next type:
   ```
   make tests
   ```
   This will make three executables called `tmvtest1`, `tmvtest2` and `tmvtest3` in the bin directory. This step also can take quite a while to finish, since the test code has tests for lots of different combinations of matrix types in all the various arithmetic operators.

   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 send and email to mike_jarvis@users.sourceforge.net about the problem.

   You may also want to make the example programs by typing:
   ```
   make examples
   ```
   This will make five executables called `vector`, `matrix`, `division`, `bandmatrix`, and `symmatrix`. These programs, along with their corresponding source code in the `examples` directory, give concrete examples of some of the common things you might want to do with the TMV library. They don't really try to be comprehensive, but they do give a pretty good overview of the main features, so looking at them may be a useful way to get started.

7. Compile you program

   Each .cpp file that uses TMV will need to have
   ```
   #include "TMV.h"
   ```
   at the top. If you are use anything besides the normal dense Matrix and Vector, then you need to include their .h files as well:
   ```
   #include "TMV_Diag.h"
   #include "TMV_Tri.h"
   #include "TMV_Band.h"
   #include "TMV_Sym.h"
   #include "TMV_SymBand.h"
   #include "TMV_Small.h"
   ```
   To compile, you will need to use the compile flag `-I[tmvdir]/include` when making the object file to tell the compiler where the TMV header files are.

   For the linking step, you need to compile with the flags `-L[tmvdir]/lib -ltmv -lm`. If you are using any of `SymMatrix`, `HermMatrix`, `BandMatrix`, `SymBandMatrix`, or `HermBandMatrix` in your code, then you will need to link with the flags `-L[tmvdir]/lib -ltmv_symband -ltmv -lm`.

If you are using BLAS and/or LAPACK calls from the TMV code, then you will also need to link with their libraries. For example, for my version of Intel's MKL version of the BLAS and LAPACK routines, 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.

7.2 Compiler flags

Here are some compiler define flags to consider using:

- `-DNDEBUG` will turn off debugging. My recommendation is to leave debugging on, since it doesn't slow things down too much. Then when you decide to export a version of the code that needs to be as fast as possible, recompile with this flag. That said, the internal TMV code should be pretty well debugged when you get it, so for compiling the library, you could go ahead and use this flag. I keep two versions of the library for my own use - one with and one without this flag. Then when I encounter a problem I can try linking to the debugging version to make sure it's not a bug in the TMV code.
- `-DNWARN` will turn off warnings. The code will output a few warnings to std::cout for a few things that aren't really errors, but are likely to be a mistake in coding. These can be turned off with this flag. They are automatically turned off when -DNDEBUG is specified.
- `-DNOBLAS` will not call any external BLAS or LAPACK routines
- `-DNOLAP` will not call any external LAPACK routines
- `-DATLAS` will set up the BLAS calls as defined by ATLAS. And (if -DNOLAP is not specified), it will also call the several LAPACK routines provided by ATLAS.
- `-DCLAPACK` will set up the LAPACK calls for the CLAPACK distribution. I find this version easier to get installed than the Fortran LAPACK distribution, so I would recommend using this if you don't already have a version of LAPCK installed somewhere on you system. Defining both ATLAS and CLAPACK will use the CLAPACK version for all LAPACK routines, including the one also provided by ATLAS. That is, ATLAS will only be used for its BLAS routines. If you want the ATLAS versions of its few LAPACK routines instead, the ATLAS installation instructions describe a way to get them into the CLAPACK library. Also, you should make sure the INCLUDE specification lists the CLAPACK version of clapack.h before the ATLAS version.
- `-DMKL` will call all the external BLAS and LAPACK routines as defined by the Intel Match Kernel Library. You may want to check the file `TMV_Blas.h` to make sure the `#include` statement is correct for your installation of MKL.
- `-DACML` will call all the external BLAS and LAPACK routines as defined by the AMD Core Math Library. You may want to check the file `TMV_Blas.h` to make sure the `#include` statement is correct for your installation of ACML.
- `-DNO_INST_FLOAT` will not instantiate any `<float>` classes or routines.
- `-DNO_INST_DOUBLE` will not instantiate any `<double>` classes or routines.
- `-DNO_INST_COMPLEX` will not instantiate any complex classes or routines.
- `-DINST_LONGDOUBLE` will instantiate `<long double>` classes and routines.
- `-DINST_INT` will instantiate `<int>` classes and routines.

The next set of compiler defines are not usually necessary. But if you have problems, one of these might be useful:

- `-DNOSTL` uses some workarounds for segments of code that use the STL library, but which didn't work for one of my compilers. I'm pretty sure it is because the compiler wasn't installed correctly, so I don't think you should really ever need to use this flag. But in any case, it will use a median-of-three quicksort algorithm for sorting rather than the standard library's sort. And it manually reads strings using character reads, rather than using the >> operator.

- `-DUSE_STEDC` specifies that the divide-and-conquer LAPACK algorithm, which is named `?stedc`, should be used for symmetric eigenvector calculation rather than the Relatively Robust Representation algorithm (named `?stegr`). I had problems with my installation of icc 10.1 and its Math Kernel Library. It gave segmentation faults in the `dstegr` function. This flag lets you avoid the problem.

- `-DXTEST` will do extra testing in the test suite, as well as add a few $O(N)$ time assert statements. (Most of the assert statements that are normally run only take $O(1)$ time.) I always do my pre-release tests with this turned on, but the executable gets quite large, as do many of the `TMV_Test*.o` files. So I turn it off for the release version. `XTEST` mostly does additional checks of the algorithms for different specific matrix pairs in the various arithmetic operations to test possible failure modes. If the shorter test suite (i.e. without `XTEST`) works ok for you, you should be fine.

- `-DXDEBUG` will do different extra (even slower) debugging. This one checks for incorrect results from the various algorithms by doing things the simple slow way and comparing the results to the fast blocked or recursive or in-place version to make sure the answer isn't (significantly) different. I use this one a lot when debugging new algorithms, usually on a file-by-file basis. Again, you shouldn't need this for an official release version. But if you do get wrong answers for something, you could use this to try to find the problem.

- `-DTMV_BLOCKSIZE=NN` will change the block size used by some routines. The current value is 64, which is good for many computers. The optimal value will depend on the size of your CPU's L1 cache. So if you want to try to tune the algorithms, you can modify this value to something more appropriate for your computer.

7.3 Known compiler issues

I have tested the code using the following compilers:

GNU's g++ - versions 3.4.6, 4.0.2, 4.1.2, and 4.2.2
Apple's g++ - version 4.0.1 (build 5465)
Intel's icc - versions 9.0, and 10.1
Portland's pgCC - version 6.1
Microsoft's Visual C++ - 2008 Express Edition

It should work with any ansi-compliant compiler, but no guarantees if you use one other than these[10]. 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. Email me at mike_jarvis@users.sourceforge.net.

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.

- **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 the version listed above, but if you have an older Mac and want to

[10]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.

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/).

- **icc 10.1:** My installation of the Intel Math Kernel Library that comes with icc 10.1 gave me some segmentation faults. I tracked the problems to two lapack functions: zhetrd and dstegr. The first was relatively easy to fix - zhetrd writes to one element past the end of the tau vector. Supplying it with a vector that is one element larger than should be necessary fixed the problem. However, I couldn't find a work around for the dstegr problem. It just seg faults whenever the matrix is larger than 2x2. So I now provide a compiler define option -DUSE_STEDC to use the divide-and-conquer algorithm instead of the Relatively Robust Representation algorithm of dstegr. This "fixes" the problem, at the cost of using a slower algorithm.

- **g++ -O2, version 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 piece of code 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 `-O` when you want a version that includes the TMV assert statements, and make sure to use `-DNDEBUG` when you want a more optimized version.

- **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 256K to 512K is required. (For compiling the test suite with the `-DXTEST` flag, more than 1GB of memory is recommended.)

- **Linker choking:** Some linkers have trouble with the size of the test suite's executable when compiled with `-DXTEST`, even with plenty of memory. So if this is the case on your platform, you're pretty much stuck with just the regular test suites. Even without `-DXTEST` you might need to compile some of the smaller test suites. All of the tests in `tmvtest1` can be found in `tmvtest1a`, `tmvtest1b` and `tmvtest1c`. Likewise `tmvtest2` and `tmvtest3` have `a`, `b` and `c` versions. These are compiled by typing `make test1a`, `make test1b`, etc. So if you want to run the tests on a machine that can't link the full programs, these smaller versions can help.

- **Standard Template Library:** In a few places, the TMV code uses standard template library algorithms. Normally these work fine, but a couple of the compilers I tested the code on didn't seem to have the STL correctly installed.

  On one computer, the linker complained that it couldn't find the code for sort, even though sort should be completely inlined, so the code should already have been in `TMV_Vector.o`, where I use it. Similarly, it had trouble linking the string read commands needed for reading in `SymBandMatrix`es. Again, this should be included in the object file where I use it, `TMV_SymBandMatrix.o`.

  The problem is probably due to something not being installed correctly on this computer, or maybe I just did not include the correct linkages or something. But rather than trying to get the sysadmin to find and fix the problem, I just added an option to compile with a simple median-of-three quicksort algorithm, rather than the STL `sort` command, and to read the strings in character by character. You can use this option by compiling with the flag `-DNOSTL`.

- **LAPACK warnings:** If you get warnings that look something like:

  ```
  dgeqrf requested more workspace than provided
  for matrix with m,n = 1000,20
  Given: 2560, requested 5120
  ```

then this means that your LAPACK uses a larger block size than 64. This might seem like a bug, but is actually a feature, since it is telling you that the LAPACK calls are not providing the optimal amount of memory. These warnings can be suppressed using either of the compiler flags `-DNDEBUG` or `-DNWARN`.

A better solution, however, is to change the variable `LAP_BLOCKSIZE` in the file `TMV_Blas.h` to something larger. In the above example, since 5120 (the "requested" size) is twice 2560 (the "given" size), `LAP_BLOCKSIZE` should probably be increased by a factor of 2, so 128. For your particular warning message, use the corresponding ratio (or larger) for how much to increase `LAP_BLOCKSIZE`.

- **pgCC:**
  - Apparently pgCC does not support exceptions when compiled with openmp turned on. So if you want the parallel versions of the algorithms, it is a little bit complicated: First you should compile the library without the `-mp` flag. Then delete the files `src/TMV_MultMM_*.o` and `src/TMV_*DC.o`. Then add the `-mp` flag (and make sure to also include the `-DNDEBUG` flag) and `make` again. Sorry this isn't easier, but this is the only compiler I know about that has this foible, so I haven't bothered to do something special in the Makefile to do this more easily.
  - The long double portion of the test suite fails in many places when compiled with pgCC. I have not had time to investigate why.
  - The compiler flag `-fastsse` breaks the code. It seems that the highest level of optimization that works is `-O4 -fast -Mcache_align`, although I have not verified that this in fact produced the fastest code - only that it does not seem to break the code.
- **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 mike_jarvis@users.sourceforge.net).
- **Sun CC:** I also tried to compile the library with Sun's CC compiler, version 5.3, and it didn't understand some of the template syntax used in TMV. However, this version is pretty old (2001), so it is likely that newer versions are more compliant with the C++ standard (they are now up to version 5.9). Unfortunately, I do not have access to a newer version of CC yet, so I have not been able try it. As usual, if you have managed to compile TMV with any version of Sun's CC, I would love to hear about it (at mike_jarvis@users.sourceforge.net).

8 Known Bugs and Deficiencies (aka To Do List)

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. Email me at mike_jarvis@users.sourceforge.net.

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 §4.4.4 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 §4.4.4)

   ```
   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.

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, small symmetric and hermitian...

   Maybe skew-symmetric and skew-hermitian. Are these worth adding? Let me know.

4. **Packed storage**

   Triangle 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. **Small Specializations**

   The SmallMatrix and SmallVector algorithms are all done inline, which will probably be fairly fast for small matrices, but I should look into whether further algorithmic specialization for particular N,M values would be worthwhile.

   It may also be worth specializing a few of the divisions inline. In particular, I suspect division by a 2×2 matrix would be worth doing.

   Also, they currently have virtual methods, which means that they require a vtable. I should rewrite their code to avoid this, since this does give a bit of a performance hit.

6. **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.

I think this is the only routine for which the LAPACK version is still significantly faster than the native TMV code.

7. **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.[11]

8. **Conditions**

   Currently, the 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.

9. **Division error estimates**

   LAPACK provides these. It would be nice to add something along the same lines.

10. **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.

11. **Config program**

    Currently, the user must modify the `Makefile` and possibly `TMV_Blas.h`. Most packages use some kind of configuration program that asks the installer some questions and automatically sets everything up correctly. I've never written one of these, so this is a bit daunting for me, but it would make the installation a lot easier for users of the library.

12. **More OpenMP**

    I have rewritten the MultMM code to exploit multiple threads using OpenMP pragma's. This gives a good boost in speed for non-BLAS compilations, especially since most of the calculation time for the higher-level algorithms is in the MultMM functions. Also, the SVD divide and conquer algorithm uses OpenMP. But I need to go through the whole code to see which other algorithms might benefit from parallelization.

13. **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.

14. **Overflow and underflow**

    For some algorithms, I check for overflow and underflow problems. (e.g. `Norm(m)`), but I have not put potentially problematic matrices into the test suite to systematically search for such problems. So there still may be places where matrices with very large or small values are a problem.

[11]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.

9 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 $\times$ bullet rather than the usual $\bullet$ to indicate this. Of course, the first few versions which don't have bullets were enormous rewrites and each of them broke earlier code. The $\times$ bullet is relevant for subsequent versions. 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 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, on the advice of Gary Bernstein, I changed the default behavior from saving the decomposition to not saving it. He convinced me that it was better to have naive code be correct and slow, rather than fast and wrong.
- Added in-place versions of the algorithms for $S = L^\dagger L$ and $S = LL^\dagger$.

Version 0.53 By popular demand (well, a 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. Unlike with SymSVDiv and HermSVDiv, there is no good reason to have these be separate classes.

- Reduced the number of operations that make temporary matrices when multiple objects use the same storage. For example, $M = LU$ can be done in place now, where L and U are the lower and upper triangle portions of M.
- Specialized Tridiagonal × Matrix calculation.
- Added ElementProd and AddElementProd functions for matrices (rather than just for vectors).
- Added CLAPACK and ACML as known versions of LAPACK (the LAPACK calls had previously been specific to the Intel MKL version). Also made the file `TMV_Blas.h` much better organized, so it is easier to tailor the code to a different LAPACK distribution with different calling conventions.

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. e.g. `U += B` and `U *= B` are allowed if `B` is upper-banded. There are many other expressions that are now legal statements.

× 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. The default is still to instantiate double and float. Now to turn these off, use -DNO_INST_DOUBLE and -DNO_INST_FLOAT respectively. To add int or long double instantiations, use -DINST_INT and -DINST_LONGDOUBLE respectively.

- Split up the library into `libtmv.a` and `libtmv_symband.a`. The latter includes everything for the `SymMatrix`, `HermMatrix`, `BandMatrix`, `SymBandMatrix`, and `HermBandMatrix` classes. The former includes everything else including `DiagMatrix` and `Upper/LowerTriMatrix`. Also, I got rid of `libsmalltmv.a`, which was basically the same as the new trimmed down `libtmv.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`, since it seems to be more common that column-major storage is the more efficient choice. Therefore, it makes sense to have this be the default.

× Changed a lot of `size_t` parameters to `int` - mostly variables which are indices of a matrix or vector, like `i` and `j` in `m(i,j)`. These variables used to be of type `size_t`, and now they are `int`, even though they are usually required to be positive.

The reason for this change was that the implementation of these functions often involves multiplying the index by a step size, which is allowed to be negative (and hence is an `int`), so there were lots of casts to `int` for these variables. I decided that it would be better to simply have them be `int` in the first place.

The particular change that is most likely to require modification to existing code involves permutations, which used to be `size_t` arrays, and are now `int` arrays. So if you used them, you might need to change their declarations in your code to `(int [])`.

× Removed `U.MakeUnitDiag`, since the name was counter-intuitive to what it actually did, and the functionality is more obvious through the `UpperTriMatrixViewOf` function.

- Sped up matrix multiplication for non-blas implementations, including openmp pragmas to allow for multiple threads on machines that support them. The code is now within a factor of 2 or 3 of a good optimized BLAS distribution. So it is still worth it to use BLAS if you have one available, but if you don't have one on you machine, the non-blas code is no longer a order of magnitude slower.
- Changed a few things which prevented Microsoft Visual C++ from compiling successfully. As far as I can tell, these are aspects in which VC++ is not fully compliant with the C++ standard, but there were work-arounds for all of them. 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. i.e.
`vmax = v.MaxElement(&imax)`
will work, but not
`vmax = MaxElement(v,&imax)`
However the functional form without the `imax` parameter still works as before. Basically, this was just a semantic choice. It seems to me that the meaning of the method form with the index parameter is much clearer than the functional form with two arguments.

I also added an optional `scale` parameter to `m.NormSq(scale)`. Again, this optional parameter is only allowed in the method version, not the function `NormSq(m)`

× 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. However, I have not put such very large or very small matrices into the test suite, so there are probably other places where these could be a problem. (Added this to the To Do list.)

- 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. With very large matrices, it is common for the determinant to overflow, but often the logarithm is sufficient. For example, one may be trying to minimize a likelihood function that includes the determinant of a matrix. Minimizing the log(likelihood) is equivalent, which thus involves the log of the determinant.

10 License

This software is licensed under the GNU General Public License. This license is sometimes referred to as "copyleft". This gist of the license (made more specific in the following text) is that you are allowed to copy, modify, and use TMV code with basically no restrictions as far as your own use is concerned, but with a few restrictions regarding further distribution.

The biggest restriction is that if you distribute code that uses the TMV code, then you must distribute it with a copyleft license as well. In other words, you may not use the TMV library with commercial, proprietary software.

The other main restriction on such a distribution is that you must include this document, including the below license, along with your version of the code. If you modified any portions of the library, you must indicate your modifications in the accompanying documentation files.

More information may be obtained from http://www.gnu.org/copyleft/.

This library is largely based 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.