Template Matrix/Vector Library User Manual
Version 0.53

Mike Jarvis

April 9, 2006

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 reasonably 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.

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 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 something. Anyway, since I did decide to write my own library, hopefully other people can benefit from my efforts and will find this 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 * v1;
   m *= 3.;
   v2 += m1*v1 + 3*v2 + m2.Transpose()*v3;
   ```

   to perform the corresponding operations.

2. **Delayed evaluation**

   Equations like the above

   ```
   v2 = m * v1;
   v2 += m * v1;
   ```

   do not create a temporary vector before assigning or adding the result to `v2`. The last equation, however:

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

   is too complicated to have a specialized routine, so it will require a couple temporary `Vector`s. Generally, if you write out one operation at a time, no temporary will be needed. More complicated equations like this will give the right answer, but may not be quite as efficient as if you write the code one operation at a time.

3. **Template**

   Of course, all of our matrix and vector classes are templates, so we can have

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

   or whatever.

4. **Mix complex/real**

   One can multiply a real matrix by a complex vector without having to copy the matrix to a new complex matrix. Likewise for the other arithmetic operations. However, we do 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. However, if the overlapping storage is not at the first element, the code will not recognize it, and you may get the wrong answer. For example:

   ```
   v.SubVector(5,15) += v.SubVector(0,10);
   ```

   will produce the wrong result, since elements `5-10` are overwritten before they are used. You can force the code to make a copy by writing:

   ```
   v.SubVector(5,15) += Vector<double>(v.SubVector(0,10));
   ```

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

   Both C- and Fortran-style (ie. zero-based or one-based) indexing are possible for element access of your 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 with $v(10)$.

   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.

7. **Sparse matrices**

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

   Some more sparse varieties which are planned, but not available yet, include: block diagonal, arbitrary sparse, symmetric/hermitian band, symmetric/hermitian block diagonal, block sparse, skew-symmertric/hermitian.

8. **Flexible storage**

   Both row-major and column-major storage are possible as an optional extra template parameter. (For band-diagonal matrices, we also allow diagonal-major storage.) This aids I/O which may require a particular format to mesh with another prgram. Also, some algorithms are faster for one storage or another, so it can be worth switching storage and checking the speed difference.

9. **Alias checking**

   Expressions such as `m *= m` pose a problem for most matrix libraries, since no matter what order you do the calculation, you will be overwriting elements that you need for later calculations. TMV automatically recognizes that 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 = LowerTriMatrixViewOf(m) * UpperTriMatrixViewOf(m)` will work correctly, even though there are three different types of matrices involves, since the address of the upper-left corner of each matrix is the same. (This expression doesn't 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 incorrect, since the vectors start at different locations, so it won't notice the aliasing, and 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 enough 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 `Quad` or somesuch), we try to make the code as fast as possible for a generic machine.

    This feature can be turned off at compile time if desired with the compilation flag -DNOBLAS, although we do not recommend it if BLAS is available on your machine, and speed is important for your application.

11. **LAPACK**

    When possible, the code can call LAPACK routines which may be faster than the native TMV code. For types which 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 also turns off the LAPACK calls as well.) For almost all algorithms, the TMV code is just 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 modern compilers. So

if you don't want to deal with getting LAPACK up and running, it generally won't be too bad, speedwise, to turn off the LAPACK calls.

Currently, the exception is the singular value decomposition. I have not yet implemented either the divide-and-conquer algorithm, nor the new RRR (Relatively Robust Representation) algorithm by Dhillon. So if the code is spending a significant amount of time doing SVD's, it may be worth having it call the LAPACK routines.

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. Some of the the sparse matrices are not included in this. See their sections below for the name of the file 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 10x10 `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;
```

In what follows, we will often omit the `tmv::` part of the designation, assuming that an appropriate `using` statement has been included in the code.

Also, throughout most of the documentation, we 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 `Quad`, for example, the main requirements are that both:

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

be defined appropriately, where `T` is your type name.

Some functions 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`".

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 which requires `sqrt` or `epsilon` for `int`s will result in a runtime error.

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 to distinguish them.

`Vector` inherits from its base class `GenVector` (ie. "generic vector"). Most operations which 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 which 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 which 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 which 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 that same subvector as above would now be `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 is that `GenVector` and `VectorComposite` 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 `GenVector` as a `ConstVectorView<T,FortranStyle>` or instantiate the `VectorComposite` as a `Vector<T,FotranStyle>`.

2.1 Constructors

Here, `T` represents the data type of the elements of the vector (eg. `double, int, complex<double>`, 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>(size_t n)`

  Makes a `Vector` of size `n` with uninitialized 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>(size_t n, T x)`

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

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

  Makes a `Vector` which copies the elements of `vv`. For the first one, `n` specifies the length. The other two get the length from `vv`.

- `tmv::Vector<T,index> v = BasisVector<T,index>(size_t n, size_t 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::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 optional step parameter allows a non-unit step between successive vector elements in memory.

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

  Copy the `Vector v2`. `v2` 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.

2.2 Access

- `size_t v.size()`

  Returns the size (length) of `v`.

- `v[i]`
  `v(i)`

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

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

  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 `VectorView`, then the return type is an object which is an lvalue (ie. it is assignable), but which may not be `T&`. Specifically, it has the type `typename tmv::Vector<T>::reference`. For a real-typed `VectorView`, it is just `T&`. But for a complex-typed `VectorView`, the return type is an object which 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, so you should declare them as:

  ```
  typename tmv::VectorView<T>::iterator
  ```

etc. instead.

If you are optimizing code using iterators of a `VectorView` or `ConstVectorView`, you may want to recast the return type into one of the following:

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

`VIt` is a mutable iterator, and `CVIt` is a const-iterator. `Unit` indicates that the step size is 1, while `Step` is general but slightly slower. `NonConj/Conj` indicates whether the view you are iterating over is the conjugate of the underlying memory. Since there can only be one return type of `v.begin()` and the rest, the type returned has to allow for either conjugation, and arbitrary steps. But the type is castable into these specializations, which can be worth it for speed sensitive portions of the code.

For regular `Vector`s, the iterators are already `Unit` and `NonConj`, so there is no need to recast them.

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

  This returns a pointer to the start of the memory. You should only need this if you are writing a highly-optimized function of a `Vector`. In this case, two other methods may be useful. `v.step()` returns the (`int`) step between elements in memory. And `v.isconj()` returns (`bool`) whether a `VectorView` is the conjugate of the actual elements in memory.

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 which 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 is akin to the `const_iterator` and `iterator` types in the standard template library.

One slightly non-intuitive thing about `VectorView`s 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. A `VectorView` is inherently an object which 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` will be 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 (Fotran-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 every other element to 0, starting with the first, 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-styel 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 subvector 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 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 `Vectors`, 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();
  ```

- `v.Flatten()`

  This probably isn't very useful for most users, but it is useful internally. It returns a real view to the real and imaginary elements of a complex `Vector`. It requires that `v` have unit step. The returned view has twice the length of `v` and also has unit step.

  The reason it is useful internally is that it allows code such as:

  ```
  tmv::Vector<complex<double> > v(10,complex<double>(1,3));
  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.

2.4 Functions of a Vector

Functions which 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 which 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()
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()`
  `RT Norm1(v)`
  `RT v.SumAbsElements()`
  `RT SumAbsElements(v)`

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

- `RT v.Norm2()`
  `RT Norm2(v)`
  `RT v.Norm()`
  `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()`
  `RT NormSq(v)`

  The square of the 2-norm of `v`: $||v||_2^2 = \sum_i |v(i)|^2$.

- `RT v.NormInf()`
  `RT NormInf(v)`
  `RT v.MaxAbsElement()`
  `RT MaxAbsElement(v)`

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

- `RT v.MaxAbsElement(size_t* i=0);`
  `RT MaxAbsElement(v,size_t* i=0);`
  `RT v.MinAbsElement(size_t* i=0);`
  `RT MinAbsElement(v,size_t* i=0);`
  `T v.MaxElement(size_t* i=0);`
  `T MaxElement(v,size_t* i=0);`
  `T v.MinElement(size_t* i=0);`
  `T MinElement(v,size_t* i=0);`

  The maximum/minimum element either by absolute value (the first four cases) or actual value (the last four cases). For complex `Vector`s, 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 optional for all of these function. If it is present (and not 0), then `*i` is set to the index of the max/min element returned.

- `T v.SumElements()`
  `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 is probably not very good programming style.

A possibly better use of this feature might be the following:

```
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`. ie. 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()`

  Conjugate each element. 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(sizet i)`

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

- `v.Swap(size_t i1, size_t i2)`

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

- `v.Permute(const size_t* p)`
  `v.Permute(const size_t* p, size_t i1, size_t i2)`
  `v.ReversePermute(const size_t* p)`
  `v.ReversePermute(const size_t* p, size_t i1, size_t 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: Permutation arrays (`p`) always use the C-style convention, even if `v` uses Fortran-style indexing.

- `v.Sort(size_t* p, tmv::ADType ad, tmv::COMPType 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 then C-style convention even if `v` uses Fortran-style indexing.

  The second parameter is `ad`, which should be `tmv::ASCEND` or `tmv::DESCEND`. It determines whether the sorted `Vector` will have its elements in ascending or descending order.

  The third parameter is `comp`, which should be `tmv::REAL_COMP, tmv::ABS_COMP, tmv::IMAG_COMP`, or `tmv::ARG_COMP`. It determines what component of the elements to use for the sorting. Only the first two make sense for non-complex `Vector`s. The second and third parameters may be omitted, in which case they default to `tmv::ASCEND` and `tmv::REAL_COMP`.

- `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, never $O(1)$.

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.

```
v2 = -v1;

v2 = x * v1;
v2 = v1 * x;
v2 = v1 / x;

v3 = v1 + v2;
v3 = v1 - v2;

v *= x;
v /= x;

v2 += v1;
v2 -= v1;

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 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 VectorView<T>& v2)`

  Performs the calculation `v2 = 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 a temporary vector, and so may be less efficient than one would 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;
tmv::Vector<T>* v2;
is >> v2;
```

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, a runtime error occurs. The second version allows you to get the size from the input. `v2` will be created with `new` (so you should subsequently `delete` it) according to whatever size the input `Vector` is.

Often, it is convenient to output only those values which 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 Matrices

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

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

`GenMatrix` in turn inherits from `BaseMatrix`. All of the various sparse `Matrix` classes also inherit from `BaseMatrix`. `BaseMatrix` has virtual declarations for the functions which can be performed on any kind of `Matrix` regardless of 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 `tmv::RowMajor` or `tmv::ColMajor`.

And, as for `Vector`, there is also a template argument indicating which indexing convention you want the matrix to use which may be either `tmv::CStyle` or `tmv::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 which take a pair of indices to define a range use the C convention of "one-past-the-end" for 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 which take range arguments take the arguments to be the first and last element in the range. So, `m.SubMatrix(1,3,4,6)` returns that 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`.) The only thing to watch out for is 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,FotranStyle>`.

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 `RowMajor` is assumed. If you want to specify `FortrandStyle` indexing, you need to include the storage argument.

3.1 Constructors

We use `stor` to indicate the storage template argument, which must be either `tmv::RowMajor` or `tmv::ColMajor`. And `index` indicates either `tmv::CStyle` or `tmv::FortranStyle`. If the `index` argument is omitted, `tmv::CStyle` is assumed. If the `stor` argument is omitted, `tmv::RowMajor` is assumed (and in this case, you must also omit `index`.

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

  Makes a `Matrix` with `nrows` rows and `ncols` columns with uninitialized 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>(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>(size_t nrows, size_t ncols, const T* mm)`
  `tmv::Matrix<T,stor,index>(size_t nrows, size_t ncols, const std::valarray<T>& mm)`
  `tmv::Matrix<T,stor,index>(size_t nrows, size_t ncols, const std::vector<T>& mm)`

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

If `stor` is `tmv::RowMajor`, then the elements of `mm` 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 `tmv::ColMajor`, then the elements of `mm` are taken to be in column-major order.

- `tmv::Matrix<T,stor,index>(const std::vector<std::vector<T> >& mm)`

  Makes a `Matrix` with elements `m(i,j) = mm[i][j]`. The size of the `Matrix` is taken from the sizes of the `vectors`. (If `index` is `FortranStyle`, then `m(i,j) = mm[i-1][j-1]`.)

- `tmv::MatrixView<T,index> m = tmv::MatrixViewOf(T* mm, size_t nrows, size_t ncols, StorageType stor)`
  `tmv::ConstMatrixView<T,index> m = tmv::MatrixViewOf(const T* mm, size_t nrows, size_t ncols, StorageType stor)`

  Makes a `MatrixView` (see §3.3 below) which refers to the exact elements of `mm`, not copying them to new storage.

- `tmv::MatrixView<T,index> m = tmv::RowVectorViewOf(Vector<T>& v)`
  `tmv::MatrixView<T,index> m = tmv::RowVectorViewOf(const VectorView<T>& v)`
  `tmv::ConstMatrixView<T,index> m = tmv::RowVectorViewOf(const GenVector<T>& v)`

  Makes a view of the `Vector` which treats it as a $1 \times n$ `Matrix` (ie. a single row). The first (non-const) version may also take a `VectorView` argument.

- `tmv::MatrixView<T,index> m = tmv::ColVectorViewOf(Vector<T>& v)`
  `tmv::MatrixView<T,index> m = tmv::ColVectorViewOf(const VectorView<T>& v)`
  `tmv::ConstMatrixView<T,index> m = tmv::ColVectorViewOf(const Vector<T>& v)`

  Makes a view of the `Vector` which treats it as an $n \times 1$ `Matrix` (ie. a single column). The first (non-const) version may also take a `VectorView` argument.

- `tmv::Matrix<T,stor,index> m1 = m2`
  `tmv::Matrix<T,stor,index> m1(const tmv::GenMatrix<T2>& m2)`

  Copy the `Matrix m2`. `m2` 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.

3.2 Access

- `size_t m.nrows()`
  `size_t m.ncols()`
  `size_t m.colsize()`
  `size_t m.rowsize()`

  Returns the size of each dimension of `m`. Note: `nrows = colsize` and `ncols = rowsize`.

- `m(i,j)`
  `m[i][j]`

  Returns the `i,j` element of `m`. ie. 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, the upper-left element is `m(1,1)`, the lower-left is `m(nrows,1)`, the upper-right is `m(1,ncols)`, and the lower-right is `m(nrows,ncols)`.

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

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 `MatrixView`, then the return type is an object which is an lvalue (ie. it is assignable), but which may not be `T&`. It has the type `typename tmv::MatrixView<T>::reference`, (which is the same as `typename tmv::VectorView<T>::reference`). It is equal to `T&` for real `MatrixView`s, but is more complicated for complex `MatrixView`s since it needs to keep track of the possibility of conjugation.

- `m.row(size_t i)`
  `m.col(size_t 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.

- `m.row(size_t i, size_t j1, size_t j2)`
  `m.col(size_t j, size_t i1, size_t 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 `VectorView` containing the elements `[m(2,3), m(3,3), m(4,3), m(5,3)]`.

  With Fotran-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)]`.)

- `m.diag()`
  `m.diag(int i)`
  `m.diag(int i, size_t k1, size_t k2)`

  Return the diagonal or one of the sub-/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 returns a subset of the diagonal with `k1 = 0` or 1 (for C-style or Fortran-style respectively) starting at the left or top edge of the `Matrix`.

- `m.SubVector(size_t i, size_t j, int istep, int jstep, size_t size)`

  If the above methods aren't sufficient to obtain the `VectorView` you need, we provide this function which returns a view through the `Matix` 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.

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

  This returns a pointer to the start of the memory. You should only need this if you are writing a highly-optimized function of a `Matrix`. In this case, some other methods may be useful:

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

  These return the step in memory between successive elements in a column or row, respectively (ie. in the `i` and `j` directions).

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

The first one returns whether `m` is a view of the conjugate of the actual elements in memory. The other two return whether the storage is row-major or column-major respectively. (You could also figure this out from the steps, but these can be more convenient.)

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. As with a `VectorView`, it can also be a conjugation of the original elements.

And again, just like a `VectorView`, there are two `Matrix` view classes: `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 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::RowMajor,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(size_t i1,size_t i2)`
  `m.Cols(size_t j1,size_t 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 or "dagger") 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()`. Our `Adjoint` is usually written $m^\dagger$, and is sometimes referred to as the Hermitian conjugate or (rarely) tranjugate. Our 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 over using `m.Adjoint()`.

- `m.View()`

  Returns a view of a `Matrix`. As with the `Vector View()` function, it is mostly useful for writing a function that can take either a `Matrix&` argument or a `const 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();
  ```

- `tmv::BaseMatrix<T>* m.NewCopy()`
  `tmv::BaseMatrix<T>* m.NewView()`
  `tmv::BaseMatrix<T>* m.NewTranspose()`
  `tmv::BaseMatrix<T>* m.NewConjugate()`
  `tmv::BaseMatrix<T>* m.NewAdjoint()`

  If you are dealing with objects which are only known to be `BaseMatrix`es, (ie. they could be a `Matrix` or a `DiagMatrix` or a `SymMatrix`, etc.) then things like `m.Transpose()` don't know what

return type to give. That means, they can't be defined for a `BaseMatrix`. So these methods are available to a `BaseMatrix` and are defined in each specific kind of `BaseMatrix` to return a pointer to the right kind of object. The returned object is made using `new`, so be sure to `delete` it.

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

  This returns a view of all of the elements of the `Matrix` as a single vector. It is mostly used internally for functions which do the same thing to all of the elements, ignoring the matrix structure, such as `MaxAbsElement` or `ConjugateSelf`. However, it may be useful for some users.

  For an actual `Matrix`, it is always allowed. 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 these two methods are legal.

3.4 Functions of a Matrix

Functions which 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 which 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:

```
RT m.Norm()
RT Norm(m)
```

are equivalent. In each case, `m` can be any `GenMatrix`. As a reminder, `RT` refers to the real type associated with `T`.

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

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

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

  The 2-norm of `m`: $||m||_2 =$ the largest singular value of m, which is also the square root of the largest eigenvalue of $(m^\dagger m)$ Be warned that this function can be fairly expensive if you have not already performed an SV decomposition `m`.

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

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

- `RT m.NormF()`
  `RT NormF(m)`
  `RT m.Norm()`
  `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 NormSq(m)`

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

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

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

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

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

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

  The determinant of `m`. For speed issues regarding this function, see §3.6 below on division.

- `minv = m.Inverse()`
  `minv = Inverse(m)`
  `m.Inverse(minv)`

  Set `minv` to the inverse of `m`. Again, see the division section for speed issues here. 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. (See §3.6.4 and §3.6.8 on pseudo-inverses and singular matrices below).

- `m.InverseATA(tmv::Matrix<T>& cov);`

  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 of the elements of 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 below). Thus, we include this function which sets the argument `cov` to the equivalent of `Inverse(m.Adjoint()*m)`, but generally does so much more efficiently than doing this explicitly.

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();`

  Clear the `Matrix m`. ie. Set each element to 0.

- `m.SetAllTo(T x);`

  Set each element 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 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 a conjugated version 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(size_t i1, size_t i2)`
  `m.SwapCols(size_t j1, size_t j2)`

  Swap the corresponding elements of two rows or two columns.

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

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

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

```
m2 = -m1

m2 = x * m1
m2 = m1 * x
m2 = m1 / x

m3 = m1 + m2
m3 = m1 - m2

m *= x
```

```
m /= x

m2 += m1
m2 -= m1

v2 = m * v1
v2 = v1 * m
v *= m

m3 = m1 * m2
m2 *= m1
```

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

For the product of two vectors, there are two orientation choices that make some sense. 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 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
```

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

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

Next, 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
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 Subroutines

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

The limit to how complicated the right hand side can be 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`. If you do use these, note that the last parameters are `MatrixView`s, rather than `Matrix`es. So you would need to call them with `m.View()` if `m` is a `Matrix`. (For `MultMV`, it would be `v.View()`.)

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

  Performs the calculation `m *= x`.

- `MultMV(T x1, const GenMatrix<Tm>& m, const GenVector<Tv1>& v1,`
  `T x2, const VectorView<T>& v2)`

  Performs the calculation `v2 = x1*m*v1 + x2*v2`.

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

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

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

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

- `Rank1Update(T x1, const GenVector<T1>& v1, const GenVector<T2>& v2,`
  `int x2, const MatrixView<T>& m)`

  Performs the calculation `m = x1*(v1^v2) + x2*m`. Note: `x2` must be either 0 or 1.

3.6 Matrix Division

One of the main things people 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 want 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 is much more common than swapping A and x, so it makes sense to use this definition for 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:

```
v2 = v1 / m
m3 = m1 / m2
v /= m
m2 /= m1
m2 = x / m1

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, which is not (numerically) a good way to perform these calculations. The appropriate decomposition (see §3.6.3 below) is used to calculate `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 (is "tall"), then there is in general no solution. And if A has more columns than rows (is "short"), then there are an infinite number of solutions.

The tall case is more common. In this case, one is not looking for an exact solution for x, but rather the value of x which 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 short case is not so common, but can still be defined reasonably. The value returned by `x = b/A` in this case is the value of `x` which 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(tmv::Matrix<T>& cov)
```

to perform the calculation efficiently. (Make sure you save the decomposition with `A.SaveDiv()` - see §3.6.5 on efficiency below.)

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 special matrices. 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, `tmv::LU` is the default for square matrices, and `tmv::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 is tall with dimensions $M \times N$, 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 A is short, 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 below.)

   There are two algorithms for doing a QRP decomposition, controlled by the global variable:
   `bool tmv::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 (the default), then there will be no diagonal element of R below and right of one which is much smaller in absolute value, where "much" means the ratio will be at most $\epsilon^{1/4}$.

4. **Singular Value Decomposition**: (`dt = tmv::SV`) $A = USV$ where U is unitary (or column-unitary if A is tall), S is diagonal with all real, non-negative values, which decrease along the diagonal, and V is unitary (or row-unitary if A is short). The singular value decomposition is most useful for matrices which are singular or nearly so. We will discuss this decomposition in more detail in §3.6.8 below.

3.6.4 Pseudo-inverse

If `m` is not square, then `m.Inverse()` should return what is called the pseudo-inverse. If `m` is tall, then `m.Inverse() * m` is the identity matrix. But `m * m.Inverse()` is not an identity. If `m` is short, then the opposite holds.

Some features of the pseudo-inverse (here we use X to represent the pseudo-inverse of M):

$$MXM = M$$
$$XMX = X$$
$$(MX)^T = MX$$
$$(XM)^T = XM$$

For singular matrices (square or not), 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 tall matricces isn't quite correct. It satisfies the first three of the above equations, but not the last one (for short M, the third equation fails). For SV, however, the returned matrix is the true pseudo-inverse and all four equations are satisfied.

For singular matrices, QR will fail to give a good pseudo-inverse, QRP will be close (again failing only the last equation), 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 as had already been calculated for `x` when calculating `y`, so this second division would be very fast. However, what if somewhere in the `[...]`, the `Matrix m` was modified? Then using the same decomposition would be incorrect - a new decomposition would be needed for the new `Matrix`.

One solution is 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 different view objects can point to the same data. It would be difficult to make sure that any change in one view invalidates all other views to the same data.

Our solution is to err on the side of correctness over efficiency and to always recalculate the decomposition by default. Of course, this can be quite inefficiencient, 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 `Matrix m` will not change anymore.

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.

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, in which case it leaves it there.)

Finally, there is another efficiency issue which may 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`:

```
m.DivideInPlace()
```

3.6.6 Determinants

The calculation of the determinant of a matrix requires similar calculations as one of 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 (all of which are similarly easy).

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();
  size_t* m.LUD().GetP();
  bool m.LUD().IsTrans();
  ```

  `GetL` and `GetU` return L and U. `GetP` returns the permutation array, P, in a form which can be used as an argument to the routines like `m.PermuteRows(P)`. Finally, `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.LUD().IsTrans() ? m2.Transpose() : m2.View();
  m2v = m.LUD().GetL() * m.LUD().GetU();
  m2v.ReversePermuteRows(m.LUD().GetP());
  ```

- 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. `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();
size_t* 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 matrices in a form which is immediately usable. `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());
m2v *= m.SVD().GetV();
```

3.6.8 Singular matrices

If a matrix is singular (ie. 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. Numerically, a matrix which is close to singular may end up with 0's from round-off errors.

Or, more commonly, a singular or nearly singular matrix will just have numerically very small values rather than 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 which 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 close to singular, and will provide unreliable results.

Singular value decompositions provides a way to deal with nearly singular matrices. There are a number of methods of the `SVD` object which can help diagnose and fix potential problems with using a singular matrix.

First, the 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 nearly 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 an infinite number of solutions to $Ax = b$. (Or, for a tall matrix, there are an infinite number of choices for x which 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: $Ax' = Ax$.

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} which 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 which can be used to set which singular values to set to 0:

```
m.SVD().Thresh(RT thresh)
m.SVD().Top(size_t 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());
```

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 non-zero with

```
size_t 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 small compared to 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 an $O(N^3)$ calculation to calculate the SVD, rather than the $O(N^2)$ calculation that the other norms need. This is fairly expensive if you have not already calculated the SVD (and saved it with `m.SaveDiv()`).

If you just want to calculate the 2-norm, or even all of 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 may want U or V, but not both for some purpose. We provide some other `DivTypes` for these cases:

```
m.DivideUsing(tmv::SVS)
m.DivideUsing(tmv::SVU)
m.DivideUsing(tmv::SVV)
```

`SVS` only keeps S. `SVU` keeps S and U, but not V. `SVV` keeps S and V, but not U. For all three, an error will result if you try to then use the decomposition for division.

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;
tmv::Matrix<T>* m2;
is >> m2;
```

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` (so you should subsequently `delete` it) according to whatever size the input `Matrix` is.

Often, it is convenient to output only those values which 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`.

4 Sparse Matrices

Most functions and methods for the various sparse matrix varieties work the same as the for regular dense matrices. In these cases, we will just list the functions that are allowed for each sparse matrix, with the implicit assumption that the effect is 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 sparse 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 other classes which inherit from `GenDiagMatrix` are: `ConstDiagMatrixView`, `DiagMatrixView`, and `DiagMatrixComposite`, which is the base class for the various arithmetic operations which return a (logical) `DiagMatrix`.

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> m(size_t n)`

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

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

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

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

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

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

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

- `tmv::DiagMatrixView<T,index> m = DiagMatrixViewOf(Matrix<T>& mm)`
  `tmv::DiagMatrixView<T,index> m = DiagMatrixViewOf(const MatrixView<T>& mm)`
  `tmv::ConstDiagMatrixView<T,index> m = DiagMatrixViewOf(const GenMatrix<T>& mm)`

  Makes a `DiagMatrixView` of the diagonal portion of `mm` (namely the main diagonal).

- `tmv::DiagMatrixView<T,index> m = DiagMatrixViewOf(Vector<T>& v)`
  `tmv::DiagMatrixView<T,index> m = DiagMatrixViewOf(const VectorView<T>& v)`
  `tmv::ConstDiagMatrixView<T,index> m = DiagMatrixViewOf(const GenVector<T>& v)`

  Makes a `DiagMatrixView` with `v` as the diagonal of the matrix.

- `tmv::DiagMatrixView<T,index> m = tmv::DiagMatrixViewOf(T* vv, size_t size)`
  `tmv::ConstDiagMatrixView<T,index> m = tmv::DiagMatrixViewOf(const T* vv,`
  `size_t size)`

  Make a `DiagMatrix` with the actual memory elements, `vv` as the diagonal.

- `tmv::DiagMatrix<T,index> m1 = m2`
  `tmv::DiagMatrix<T,index> m1(const GenDiagMatrix<T2>& m2)`

  Copy the `DiagMatrix m2`. `m2` 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.

4.1.2 Access

```
size_t m.nrows()
size_t m.ncols()
size_t m.colsize()
size_t m.rowsize()
size_t m.size()
```

where `m.size() = m.colsize() = m.rowsize()`

```
m(i,i)
m(i)
```

where `m(i) = m(i,i)`. For the mutable `m(i,j)` version, `i` must equal `j`. If `m` is not mutable, then `m(i,j)` wth `i`$\neq$`j` returns the value 0.

```
m.diag()
```

4.1.3 Views

```
m.SubDiagMatrix(int i1, int i2, int istep = 1)
```

This is equivalent to `DiagMatrixViewOf(m.diag().SubVector(i,j,istep))`.

```
m.Real()
m.Imag()
m.Transpose()
m.Conjugate()
m.Adjoint()
m.View()
tmv::BaseMatrix<T>* m.NewCopy()
tmv::BaseMatrix<T>* m.NewView()
tmv::BaseMatrix<T>* m.NewTranspose()
tmv::BaseMatrix<T>* m.NewConjugate()
tmv::BaseMatrix<T>* m.NewAdjoint()
```

4.1.4 Functions

```
RT m.Norm1() = Norm1(m)
RT m.Norm2() = Norm2(m)
RT m.NormInf() = NormInf(m)
RT m.NormF() = NormF(m) = m.Norm() = Norm(m)
RT m.MaxAbsElement() = MaxAbsElement(m)
```

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

```
RT m.NormSq() = NormSq(m)
T m.Trace() = Trace(m)
T m.Det() = Det(m)
minv = m.Inverse() = Inverse(m)
m.Inverse(minv)
m.InverseATA(tmv::Matrix<T>& cov)
```

Since the inverse of a `DiagMatrix` is likewise a `DiagMatrix`, we also provide a version of the `Inverse` syntax which allows minv to be a `DiagMatrix`. Likewise for `InverseATA`.

```
m.Zero()
m.SetAllTo(T x)
m.Clip(RT thresh)
m.SetToIdentity(T x = 1)
m.ConjugateSelf()
m.TransposeSelf()
Swap(m1,m2)
```

4.1.5 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
d2 = d1 / x

d3 = d1 + d2
d3 = d1 - d2
m2 = m1 + d
m2 = d + m1
m2 = m1 - d
m2 = d - m1

d *= x
d /= x

d2 += d1
d2 -= d1
m += d
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
d2 = d1 - x
d2 = x - d1
d += x
d -= x
```

The subroutines for `DiagMatrix` arithmetic are:

```
MultXM(T x, const DiagMatrixView<T>& m)
MultMV(T x1, const GenDiagMatrix<Tm>& m, const GenVector<Tv1>& v1,
        T x2, const VectorView<T>& v2)
AddMM(T x1, const GenDiagMatrix<T1>& m1, T x2, const DiagMatrixView<T>& m2)
MultMM(T x1, const GenDiagMatrix<T>& m1, const GenMatrix<T>& m2,
        T x2, const MatrixView<T>& m3)
MultMM(T x1, const GenMatrix<T>& m1, const GenDiagMatrix<T>& m2,
        T x2, const MatrixView<T>& m3)
MultMM(T x1, const GenDiagMatrix<T>& m1, const GenDiagMatrix<T>& m2,
        T x2, const DiagMatrixView<T>& m3)
```

4.1.6 Division

The division operations are:

```
v2 = v1 / d
m2 = m1 / d
m2 = d / m1
d3 = d1 / d2
v /= d
d2 /= d1
m /= d
```

plus all of these with `%` rather than `/`.

There is only one allowed `DivType` for a `DiagMatrix`, `tmv::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. ie. no extra storage is needed, and all of the `m.DivideInPlace()`, `m.SaveDiv()`, etc. are irrelevant.

If a `DiagMatrix` is singular, you can check easilty with `m.Singular()`, but there is no direct way to treat the division as a SVD and skip any divisions by 0. I suppose it would be easy to add that functionality, since a `DiagMatrix` is also already an SV decomposition, ignoring the fact that S should be all real and positive, which is a small detail. But I haven't done so yet.

4.1.7 Input/Output

The simplest output is the usual:

```
os << m
```

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:

```
m.WriteCompact(os)
```

which outputs in the format:

```
D size ( m(0,0)  m(1,1)  m(2,2)  ...  m(size-1,size-1) )
```

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

```
tmv::DiagMatrix<T> m(size);
is >> m;
tmv::DiagMatrix<T>* m2;
is >> m2;
```

One can 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`.

As usual, `ConstUppertriMatrixView`, `UpperTriMatrixView`, and `UpperTriMatrixComposite` also inherit from the base class `GenUpperTriMatrix`. Similarly for `GenLowerTriMatrix`.

Almost all of the routines are analogous between `UpperTriMatrix` and `LowerTriMatrix`, so we generally only list each routine once (the `UpperTriMatrix` version for definiteness). If there is a difference, we will point it out using `U` and `L` respectively for the two types.

All 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`.

If `index` is omitted, `CStyle` is assumed. If `stor` is omitted (in which case `index` must also be omitted), `RowMajor` is assumed. If `dt` is omitted (in which case `stor` and `index` must both be omitted), `NonUnitDiag` is assumed.

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.

4.2.1 Constructors

- `tmv::UpperTriMatrix<T,dt,stor,index> m(size_t n)`

  Makes an `n` $\times$ `n` `UpperTriMatrix` with uninitialized values. If debugging is turned on (ie. not turned off with -DNDEBUG), then the values are in fact initialized to 888.

- `tmv::UpperTriMatrix<T,dt,stor,index>(size_t n, T x)`

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

- `tmv::UpperTriMatrix<T,dt,stor,index>(const GenMatrix<T>& mm)`
  `tmv::UpperTriMatrix<T,dt,stor,index>(const GenUperTriMatrix<T>& mm)`

  Makes an `UpperTriMatrix` which copies the corresponding values of `mm`. Note that the second one is allowed to have `mm` 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::UpperTriMatrixView<T,index> m = UpperTriMatrixViewOf(
          Matrix<T>& mm, DiagType dt)
  tmv::UpperTriMatrixView<T,index> m = UpperTriMatrixViewOf(
          const MatrixView<T>& mm, DiagType dt)
  tmv::ConstUpperTriMatrixView<T,index> m = UpperTriMatrixViewOf(
          const GenMatrix<T>& mm, DiagType dt)
  tmv::UpperTriMatrixView<T,index> m = UpperTriMatrixViewOf(
          UpperTriMatrix<T>& mm, DiagType dt)
  tmv::UpperTriMatrixView<T,index> m = UpperTriMatrixViewOf(
          const UpperTriMatrixView<T,index>& mm, DiagType dt)
  tmv::ConstUpperTriMatrixView<T,index> m = UpperTriMatrixViewOf(
          const GenUpperTriMatrix<T,index>& mm, DiagType dt)
  ```

  Make an `UpperTriMatrixView` of the corresponding portion of `mm`. Note that when `mm` is an `UpperTriMatrix` with `NonUnitDiag`, these may be used (via `dt`) to view it as an `UpperTriMatrix` with `UnitDiag`.

- ```
  tmv::UpperTriMatrixView<T,index> m = tmv::UpperTriMatrixViewOf(T* mm,
          size_t size, DiagType dt, StorageType stor)
  tmv::ConstUpperTriMatrixView<T,index> m = tmv::UpperTriMatrixViewOf(
          const T* mm, size_t size, DiagType dt, StorageType stor)
  ```

  Make a `UpperTriMatrixView` with the actual memory elements, `mm` as the upper triangle. One wrinkle here is that if `dt` is `UnitDiag`, then `mm` 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). In fact, `mm` must be of length `size` $\times$ `size`, so all of the lower triangle elements must be in memory, even though they are never used.

- `tmv::UpperTriMatrix<T,dt,stor,index> m1 = m2`
  `tmv::UpperTriMatrix<T,dt,stor,index> m1(const GenUpperTriMatrix<T2>& m2)`

  Copy the `UpperTriMatrix m2`. `m2` 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.

4.2.2 Access

```
size_t m.nrows()
size_t m.ncols()
size_t m.colsize()
size_t m.rowsize()
size_t m.size()
m(i,j)
m.row(size_t i,size_t j1,size_t j2)
m.col(size_t i,size_t j1,size_t j2)
m.diag()
m.diag(int i)
m.diag(int i,size_t k1,size_t 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 or lower triangle shape of `m`. If `dt` is `UnitDiag`, then the range may not include the diagonal element. Similarly, `m.diag()` is valid only if `dt` is `NonUnitDiag`.

```
m.SubVector(size_t i, size_t j, int istep, int jstep, size_t size)
```

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

4.2.3 Views

```
m.SubMatrix(int i1, int i2, int j1, int j2)
m.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 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.

```
m.SubTriMatrix(int i1, int i2, int istep = 1)
```

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

```
m.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.

```
m.Real()
m.Imag()
m.Transpose()
m.Conjugate()
m.Adjoint()
m.View()
tmv::BaseMatrix<T>* m.NewCopy()
tmv::BaseMatrix<T>* m.NewView()
tmv::BaseMatrix<T>* m.NewTranspose()
tmv::BaseMatrix<T>* m.NewConjugate()
tmv::BaseMatrix<T>* m.NewAdjoint()
```

Note that the transpose and adjoint of a `LowerTriMatrix` is an `UpperTriMatrixView` and vice versa.

4.2.4 Functions

```
RT m.Norm1() = Norm1(m)
RT m.Norm2() = Norm2(m)
RT m.NormInf() = NormInf(m)
RT m.NormF() = NormF(m) = m.Norm() = Norm(m)
RT m.NormSq() = NormSq(m)
RT m.MaxAbsElement() = MaxAbsElement(m)
```

```
T m.Trace() = Trace(m)
T m.Det() = Det(m)
minv = m.Inverse() = Inverse(m)
m.Inverse(minv)
m.InverseATA(tmv::Matrix<T>& cov)
```

Since the inverse of an `UpperTriMatrix` is likewise an `UpperTriMatrix`, we also provide a version of the `Inverse` syntax which allows `minv` to be an `UpperTriMatrix`.

```
m.Zero()
m.SetAllTo(T x)
m.Clip(RT thresh)
m.SetToIdentity(T x = 1)
m.ConjugateSelf()
Swap(m1,m2)
```

4.2.5 Arithmetic

In addition to `x`, `v`, and `m` from before, we now add `U` and `L` for a `UpperTriMatrix` and `LowerTriMatrix` respectively.

```
U2 = -U1
L2 = -L1

U2 = x * U1
U2 = U1 * x
U2 = U1 / x
L2 = x * L1
L2 = L1 * x
L2 = L1 / x

U3 = U1 + U2
U3 = U1 - U2
m2 = m1 + U
m2 = U + m1
m2 = m1 - U
m2 = U - m1
L3 = L1 + L2
L3 = L1 - L2
m2 = m1 + L
m2 = L + m1
m2 = m1 - L
m2 = L - m1
m = L + U
m = U + L
m = L - U
m = U - L

U *= x
U /= x
L *= x
```

```
L /= x

U2 += U1
U2 -= U1
m += U
m -= U
L2 += L1
L2 -= L1
m += L
m -= L

v2 = U * v1
v2 = v1 * U
v *= U
v2 = L * v1
v2 = v1 * L
v *= L

U3 = U1 * U2
m2 = U * m1
m2 = m1 * U
L3 = L1 * L2
m2 = L * m1
m2 = m1 * L
m = U * L
m = L * U
U2 *= U1
m *= U
L2 *= L1
m *= L

U2 = U1 + x
U2 = x + U1
U2 = U1 - x
U2 = x - U1
L2 = L1 + x
L2 = x + L1
L2 = L1 - x
L2 = x - L1
U += x
U -= x
L += x
L -= x
```

The subroutines for `UpperTriMatrix` and `LowerTriMatrix` arithmetic are:

```
MultXM(T x, const UpperTriMatrixView<T>& m)
MultXM(T x, const LowerTriMatrixView<T>& m)

MultMV(T x1, const GenUpperTriMatrix<Tm>& m, const GenVector<Tv1>& v1,
```

```
        T x2, const VectorView<T>& v2)
MultMV(T x1, const GenLowerTriMatrix<Tm>& m, const GenVector<Tv1>& v1,
        T x2, const VectorView<T>& v2)

AddMM(T x1, const GenUpperTriMatrix<T1>& m1, T x2, const UpperTriMatrixView<T>& m2)
AddMM(T x1, const GenLowerTriMatrix<T1>& m1, T x2, const LowerTriMatrixView<T>& m2)
AddMM(T x1, const GenUpperTriMatrix<T1>& m1, T x2, const MatrixView<T>& m2)
AddMM(T x1, const GenLowerTriMatrix<T1>& m1, T x2, const MatrixView<T>& m2)
AddMM(T x1, const GenLowerTriMatrix<T1>& m1, T x2, const GenUpperTriMatrix<T2>& m2,
        const MatrixView<T>& m3)

MultMM(T x1, const GenUpperTriMatrix<T>& m1, const GenMatrix<T>& m2,
        T x2, const MatrixView<T>& m3)
MultMM(T x1, const GenLowerTriMatrix<T>& m1, const GenMatrix<T>& m2,
        T x2, const MatrixView<T>& m3)
MultMM(T x1, const GenMatrix<T>& m1, const GenUpperTriMatrix<T>& m2,
        T x2, const MatrixView<T>& m3)
MultMM(T x1, const GenMatrix<T>& m1, const GenLowerTriMatrix<T>& m2,
        T x2, const MatrixView<T>& m3)
MultMM(T x1, const GenUpperTriMatrix<T>& m1, const GenUpperTriMatrix<T>& m2,
        T x2, const UpperTriMatrixView<T>& m3)
MultMM(T x1, const GenLowerTriMatrix<T>& m1, const GenLowerTriMatrix<T>& m2,
        T x2, const LowerTriMatrixView<T>& m3)
MultMM(T x1, const GenUpperTriMatrix<T>& m1, const GenLowerTriMatrix<T>& m2,
        T x2, const MatrixView<T>& m3)
MultMM(T x1, const GenLowerTriMatrix<T>& m1, const GenUpperTriMatrix<T>& m2,
        T x2, const MatrixView<T>& m3)
```

4.2.6 Division

The division operations are:

```
v2 = v1 / U
v2 = v1 / L
m2 = m1 / U
m2 = m1 / L
m2 = U / m1
m2 = L / m1
U3 = U1 / U2
L3 = L1 / L2
m = U / L
m = L / U
v /= U
v /= L
U2 /= U1
L2 /= L1
m /= U
m /= L
```

plus all of these with `%` rather than `/`.

There is only one allowed `DivType` for an `UpperTriMatrix` or a `LowerTriMatrix`, `tmv::LU`. And, since it is also the default behavior, there is no reason to ever use this function. 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. ie. 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 treat the division as a SVD and skip any divisions by 0. So trying to divide by a singular triangle matrix will result in a runtime error.

4.2.7 Input/Output

The simplest output is the usual:

```
os << m
```

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 size
( m(0,0)  m(0,1)  m(0,2)  ...  m(0,size-1) )
( m(1,1)  m(1,2)  ...  m(1,size-1) )
...
( m(size-1,size-1) )
```

For an `LowerTriMatrix`,

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

outputs in the format:

```
L size
( m(0,0) )
( m(1,0)  m(1,1) )
...
( m(size-1,0)  m(size-1,1) ... m(size-1,size-1) )
```

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

```
tmv::UpperTriMatrix<T> U(size);
tmv::LowerTriMatrix<T> L(size);
is >> U >> L;
tmv::UpperTriMatrix<T>* U2;
tmv::LowerTriMatrix<T>* L2;
is >> U2 >> L2;
```

One can write small values as 0 with

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

4.3 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 out 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`.

As usual, `ConstSymMatrixView`, `SymMatrixView`, and `SymMatrixComposite` also inherit from the base class `GenSymMatrix`.

Usually, the symmetric/hermitian question does not affect the use of the classes. (It does affect the actual calculations performed of course.) So we will just refer here to `SymMatrix` and point out whenever a `HermMatrix` acts differently.

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`. `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.

If `index` is omitted, `CStyle` is assumed. If `stor` is omitted (in which case `index` must also be omitted), `RowMajor` is assumed. If `uplo` is omitted (in which case `stor` and `index` must both be omitted), `Upper` is assumed.

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.

4.3.1 Constructors

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

  Makes an `n` $\times$ `n` `SymMatrix` with uninitialized values. If debugging is turned on (ie. not turned off with -DNDEBUG), then the values are in fact initialized to 888.

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

  Makes an `n` $\times$ `n` `SymMatrix` with all values equal to `x`. For the `HermMatrix` version of this, `x` must be real.

- `tmv::SymMatrix<T,uplo,stor,index>(const GenMatrix<T>& mm)`

  Makes a `SymMatrix` which copies the corresponding values of `mm`.

- ```
  tmv::SymMatrixView<T,index> m = SymMatrixViewOf(Matrix<T>& mm, UpLoType uplo)
  tmv::SymMatrixView<T,index> m = SymMatrixViewOf(const MatrixView<T>& mm,
          UpLoType uplo)
  tmv::ConstSymMatrixView<T,index> m = SymMatrixViewOf(const GenMatrix<T>& mm,
          UpLoType uplo)
  tmv::SymMatrixView<T,index> m = HermMatrixViewOf(Matrix<T>& mm, UpLoType uplo)
  tmv::SymMatrixView<T,index> m = HermMatrixViewOf(const MatrixView<T>& mm,
  ```
  ```

```
        UpLoType uplo)
tmv::ConstSymMatrixView<T,index> m = HermMatrixViewOf(const GenMatrix<T>& mm,
        UpLoType uplo)
```

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

- ```
  tmv::SymMatrixView<T,index> m = tmv::SymMatrixViewOf(T* mm, size_t size,
          UpLoType uplo, StorageType stor)
  tmv::ConstSymMatrixView<T,index> m = tmv::SymMatrixViewOf(const T* mm,
          size_t size, UpLoType uplo, StorageType stor)
  tmv::SymMatrixView<T,index> m = tmv::HermMatrixViewOf(T* mm, size_t size,
          UpLoType uplo, StorageType stor)
  tmv::ConstSymMatrixView<T,index> m = tmv::HermMatrixViewOf(const T* mm,
          size_t size, UpLoType uplo, StorageType stor)
  ```

  Make a `SymMatrixView` with the actual memory elements, `mm` in either the upper or lower triangle. `mm` must be of length `size` $\times$ `size`, even though only about half of the values are actually used,

- ```
  tmv::SymMatrix<T,uplo,stor,index> m1 = m2
  tmv::SymMatrix<T,uplo,stor,index> m1(const GenSymMatrix<T2>& m2)
  ```

  Copy the `SymMatrix m2`. `m2` 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.

4.3.2 Access

```
size_t m.nrows()
size_t m.ncols()
size_t m.colsize()
size_t m.rowsize()
size_t m.size()
m(i,i)
m.row(size_t i,size_t j1,size_t j2)
m.col(size_t i,size_t j1,size_t j2)
m.diag()
m.diag(int i)
m.diag(int i,size_t k1,size_t k2)
m.SubVector(size_t i, size_t j, int istep, int jstep, size_t size)
m.ptr()
m.cptr()
m.isherm()
m.issym()
m.isconj()
m.isrm()
m.iscm()
m.isupper()
```

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 `m`. 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 do so in too steps:

```
m.row(i,0,i) = ...
m.row(i,i,ncols) = ...
```

4.3.3 Views

```
m.SubMatrix(int i1, int i2, int j1, int j2)
m.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 submatrix must be completely within the upper or lower triangle.

```
m.SubSymMatrix(int i1, int i2, int istep = 1)
```

This returns a `SymMatrixView` whose upper-left corner is `m(i1,i1)`, and whose lower-right corner is `m(i2-istep,i2-istep)`. If `istep` $\neq 1$, then it is the step in both the `i` and `j` directions.

```
m.UpperTri()
m.LowerTri()
```

These return an `UpperTriMatrixView` or `LowerTriMatrixView` respectively of the corresponding portion of the `SymMatrix`. Both of these are valid calls regardless of which triangle stores the actual data for `m`.

```
m.Real()
m.Imag()
m.Transpose()
m.Conjugate()
m.Adjoint()
m.View()
tmv::BaseMatrix<T>* m.NewCopy()
tmv::BaseMatrix<T>* m.NewView()
tmv::BaseMatrix<T>* m.NewTranspose()
tmv::BaseMatrix<T>* m.NewConjugate()
tmv::BaseMatrix<T>* m.NewAdjoint()
```

Note that the imaginary part of a complex hermitian matrix is anti-symmetric, so `m.Imag()` is illegal for `HermMatrix`es. To manipulate the imaginary part of a `HermMatrix`, you could use `m.UpperTri().Imag()`, or perhaps `m.UpperTri().OffDiag().Imag()`, since the diagonal elements are all real.

4.3.4 Functions

```
RT m.Norm1() = Norm1(m)
RT m.Norm2() = Norm2(m)
RT m.NormInf() = NormInf(m)
RT m.NormF() = NormF(m) = m.Norm() = Norm(m)
RT m.NormSq() = NormSq(m)
RT m.MaxAbsElement() = MaxAbsElement(m)
T m.Trace() = Trace(m)
T m.Det() = Det(m)
minv = m.Inverse() = Inverse(m)
m.Inverse(minv)
m.InverseATA(tmv::Matrix<T>& cov)
```

Since the inverse of a SymMatrix is also a SymMatrix, we also provide additional methods which allow minv to be a SymMatrix.

```
m.Zero()
m.SetAllTo(T x)
m.Clip(RT thresh)
m.SetToIdentity(T x = 1)
m.ConjugateSelf()
m.TransposeSelf()
m.SwapRowsCols(size_t i1, size_t i2)
Swap(m1,m2)
```

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

The new method, SwapRowsCols, would be equivalent to

```
m.SwapRows(i1,i2);
m.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.

4.3.5 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
s2 = s1 / x

s3 = s1 + s2
s3 = s1 - s2
m2 = m1 + s
m2 = s + m1
m2 = m1 - s
m2 = s - m1

s *= x
s /= x

s2 += s1
s2 -= s1
m += s
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
s2 = s1 - x
s2 = x - s1
s += x
s -= x

s = v ^ v
s += v ^ v
s = m * m.Transpose()
s += m * m.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 one (called a "rank-k update"), the `m`'s need to be the same data, and for a complex hermitian `s`, `Transpose` should be replaced with `Adjoint`. Also, for these rank-k updates, m may be a `Matrix`, `UpperTriMatrix`, or `LowerTriMatrix`

There are a couple of issues currently with `SymMatrix` arithmetic. Specifically, if an arithmetic result could potentially result in a `SymMatrix`, then we use a class derived from `SymMatrixComposite`. This means that it will try to cast itself as a `SymMatrix` if all else fails. However, some of these combinations may not actually result in a `SymMatrix`. Some examples:

- `s1 + s2` where both are complex, but one is really a `SymMatrix` and the other is a `HermMatrix`. You should be allowed to assign this sum to a `Matrix`, but that is currently not allowed.
- `x * s` where `x` is complex with a non-zero imaginary component and `s` is hermitian. Again you should be allowed to assign this to a `Matrix`, but that is not allowed. Furthermore, in this case, the equation `m = x*s*m2` would not even be allowed, since the compiler first tries to combine `x*s`, which is illegal. However this problem would be solved by putting parentheses around `s*m2`. The equation `m = x*(s*m2)` correctly calls the appropriate version of `MultMM` below.

The subroutines for `SymMatrix` arithmetic are:

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

MultMV(T x1, const GenSymMatrix<Tm>& m, const GenVector<Tv1>& v1,
       T x2, const VectorView<T>& v2)

AddMM(T x1, const GenSymMatrix<T1>& m1, T x2, const SymMatrixView<T>& m2)
AddMM(T x1, const GenSymMatrix<T1>& m1, T x2, const MatrixView<T>& m2)

MultMM(T x1, const GenSymMatrix<T>& m1, const GenMatrix<T>& m2,
       T x2, const MatrixView<T>& m3)
MultMM(T x1, const GenMatrix<T>& m1, const GenSymMatrix<T>& m2,
       T x2, const MatrixView<T>& m3)
MultMM(T x1, const GenSymMatrix<T>& m1, const GenSymMatrix<T>& m2,
       T x2, const MatrixView<T>& m3)

Rank1Update(T x1, const GenVector<T1>& v, int x2, const SymMatrixView<T>& m)
```

```
RankKUpdate(T x1, const GenMatrix<T1>& m1, int x2, const SymMatrixView<T>& m2)
RankKUpdate(T x1, const GenLowerTriMatrix<T1>& m1, int x2,
        const SymMatrixView<T>& m2)
RankKUpdate(T x1, const GenUpperTriMatrix<T1>& m1, int x2,
        const SymMatrixView<T>& m2)
```

4.3.6 Division

The division operations are:

```
v2 = v1 / s
m2 = m1 / s
m2 = s / m1
m = s1 / s2
v /= s
m /= s
```

plus all of these with `%` rather than `/`.

`SymMatrix` has three possible choices for the division decomposition:

1. `m.DivideUsing(tmv::LU)` will perform something similar to the LU decomposition for regular matrices. Instead, it does what is called an LDL or Bunch-Kauffman 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:

   ```
   m.HermLUD().GetL()
   m.HermLUD().GetD()
   m.HermLUD().GetP()
   ```

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

   ```
   Matrix<T> m2 = m.HermLUD().GetL() * m.HermLUD().GetD();
   m2 *= m.HermLUD().GetL().Adjoint();
   m2.ReversePermuteRows(m.HermLUD().GetP());
   m2.ReversePermuteCols(m.HermLUD().GetP());
   ```

   For a complex, symmetric `m`, you would need to replace `HermLUD` with `SymLUD` and `Adjoint` with `Transpose`. Real symmetric matrices use the `HermLUD` class.

2. `m.DivideUsing(tmv::CH)` will perform a Cholesky decomposition. `m` 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, but U is the adjoint of L, and there is no permutation. It is 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 $m = A^\dagger A$ where A is a matrix with at least as many rows as columns. Then m 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 m is nearly singular. Even if it doesn't fail, if `m` is nearly singular, the calculation may be numerically unstable, and Bunch-Kauffman may be a better choice.

The only advantage of Cholesky over Bunch-Kauffman is speed. If you know your matrix is positive-definite, the Cholesky decomposition is the fastest way to do division.

To access this decomposition, use:

```
m.CHD().GetL()
```

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

```
Matrix<T> m2 = m.CHD().GetL() * m.CHD().GetL().Adjoint()
```

3. `m.DivideUsing(tmv::SV)` will perform a singular value decomposition.

   In this case $U = V^T$ for symmetric matrices or $V^\dagger$ for hermitian matrices. The one slight change in how we make the decomposition for hermitian matrices (and real symmetric matrices) is that we allow the elements of S to be negative, whereas singular values are really defined to be positive. So if you want to use them as singular values, you just need to remember to take the absolute value.

   The reason for this is that $m = V^\dagger S V$ is an eigenvector decomposition, in addition to a singular value decomposition. This has important uses in its own right beyond just division, so we didn't want to spoil that. The elements along the diagonal of S are the eigenvalues, and the columns of V are the corresponding eigenvectors.

   To access this decomposition, use:

   ```
   m.HermSVD().GetU()
   m.HermSVD().GetS()
   m.HermSVD().GetV()
   ```

   The product of these three (using `DiagMatrixViewOf(m.HermSVD().GetS())` for S) should result in a matrix numerically very close to `m`. For a complex, symmetric `m`, you would need to replace `HermSVD` with `SymSVD`. Also, the values in S for the `SymSVD` version are all set to be positive (since they aren't eigenvalues anyway).

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

   ```
   m.HermSVD().Thresh(RT thresh)
   m.HermSVD().Top(size_t nsing)
   m.HermSVD().Condition()
   m.HermSVD().GetKMax()
   ```

   (Likewise for m.SymSVD().)

   There are also the `SVS`, `SVU`, and `SVV` options for the decomposition which cannot be used for division, but which can access parts of the decomposition.

The routines

```
m.SaveDiv()
m.SetDiv()
m.ReSetDiv()
m.UnSetDiv()
m.DivideInPlace()
```

work the same as for regular `Matrix`es.

4.3.7 Input/Output

The simplest output is the usual:

```
os << m
```

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`:

```
m.WriteCompact(os)
```

outputs in the format:

```
H/S size
( m(0,0) )
( m(1,0)  m(1,1) )
...
( m(size-1,0)  m(size-1,1) ... m(size-1,size-1) )
```

where `H/S` means either `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> m_sym(size);
tmv::HermMatrix<T> m_herm(size);
is >> m_sym >> m_herm;
tmv::SymMatrix<T>* m2_sym;
tmv::HermMatrix<T>* m2_herm;
is >> m2_sym >> m2_herm;
```

One can write small values as 0 with

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

4.4 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 which 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`. As usual, the other classes which inherit from `GenBandMatrix` are `ConstBandMatrixView`, `BandMatrixView`, and `BandMatrixComposite`.

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. `RowMajor` and `CStyle` are the defaults if these parameters (or just `index`) are omitted.

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 which 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.

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

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

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

First, notice that all three storage methods require 4 extra locations in memory which do not hold any actual matrix data. (They require a total of 31 memory addresses for only 27 that are used.) This is because we want to have the same step size between consecutive row elements for every row. Likewise for the columns (which in turn implies that it is also true for the diagonals).

For $N \times N$ square matrices, the total memory needed is $(N-1)*(nlo+nhi+1)+1$, which wastes only $(nlo-1)*nlo/2+(nhi-1)*nhi/2$ 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 which take the matrix information from an array (`T*, vector<T>`, or `valarray<T>`), the start of the array needs to be at the start of the lowest sub-diagonal. However, the access function `m.ptr()` returns the address of the upper left element. So if you need to write some low-level routines for `DiagMajor BandMatrix`es, this will be important to know.

4.4.1 Constructors

- `tmv::BandMatrix<T,stor,index> m(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 (ie. not turned off with -DNDEBUG), then the values are initialized to 888.

- `tmv::BandMatrix<T,stor,index>(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>(size_t nrows, size_t ncols, int nlo, int nhi,
          const T* mm)
  tmv::BandMatrix<T,stor,index>(size_t nrows, size_t ncols, int nlo, int nhi,
          const std::valarray<T>& mm)
  tmv::BandMatrix<T,stor,index>(size_t nrows, size_t ncols, int nlo, int nhi,
          const std::vector<T>& mm)
  ```

  Makes a `BandMatrix` which copies the elements from `mm`. See the discussion above about the different storage types to see what order these elements should be. The function `tmv::BandStorageLength` will tell you how long `mm` must be. The elements which don't fall in the bounds of the actual matrix are not used and may be left undefined.

- ```
  tmv::BandMatrix<T,stor,index>(const GenMatrix<T>& mm, int nlo, int nhi)
  tmv::BandMatrix<T,stor,index>(const GenBandMatrix<T>& mm, int nlo, int nhi)
  ```

  Makes an `BandMatrix` which copies the corresponding values of `mm`. For the second one, `nlo` and `nhi` must not be larger than those for `mm`.

- ```
  tmv::BandMatrixView<T,index> m = BandMatrixViewOf(Matrix<T>& mm,
          int nlo, int nhi)
  tmv::BandMatrixView<T,index> m = BandMatrixViewOf(const MatrixView<T>& mm,
          int nlo, int nhi)
  tmv::ConstBandMatrixView<T,index> m = BandMatrixViewOf(const GenMatrix<T>& m,
          int nlo, int nhi)
  tmv::BandMatrixView<T,index> m = BandMatrixViewOf(BandMatrix<T>& mm,
          int nlo, int nhi)
  tmv::BandMatrixView<T,index> m = BandMatrixViewOf(const BandMatrixView<T>& mm,
          int nlo, int nhi)
  tmv::ConstBandMatrixView<T,index> m = BandMatrixViewOf(
          const GenBandMatrix<T>& m, int nlo, int nhi)
  ```

  Makes an `BandMatrixView` of the corresponding portion of `mm`.

  Note: if you want a `BandMatrix` which includes the whole upper triangle plus only a few sub-diagonals (or vice versa), you are better off memory-wise making a regular `Matrix` and then viewing the portion you want as a `BandMatrixView`. This is because the memory requirements (for a square matrix anyway) are identical for a regular `Matrix` and a `BandMatrix` with `nlo` + `nhi` = N. For a Hessenberg matrix (upper triangle plus one sub-diagonal), this is already the case. So with any more sub-diagonals, you are actually using more memory with a `BandMatrix` than with a `Matrix`. But you can still get the algorithmic speed-up of the `BandMatrix` by viewing the portion you want as a `BandMatrixView` using the above constructor.

- ```
  tmv::BandMatrixView<T,index> m = tmv::BandMatrixViewOf(T* mm, size_t ncols,
          size_t nrows, int nlo, int nhi, StorageType stor)
  ```

```
tmv::ConstBandMatrixView<T,index> m = tmv::BandMatrixViewOf(const T* mm,
        size_t ncols, size_t nrows, int nlo, int nhi, StorageType stor)
```

Make a `BandMatrixView` using the actual memory elements, `mm`.

- ```
  tmv::BandMatrix<T,stor,index> m = UpperBiDiagMatrix(const GenVector<T>& v1,
          const GenVector<T>& v2)
  tmv::BandMatrix<T,stor,index> m = LowerBiDiagMatrix(const GenVector<T>& v1,
          const GenVector<T>& v2)
  tmv::BandMatrix<T,stor,index> 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> m1 = m2
  tmv::BandMatrix<T,stor,index> m1(const GenBandMatrix<T2>& m2)
  ```

  Copy the `BandMatrix m2`. `m2` 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.

4.4.2 Access

```
size_t m.nrows()
size_t m.ncols()
size_t m.colsize()
size_t m.rowsize()
m(i,i)
m.row(size_t i,size_t j1,size_t j2)
m.col(size_t i,size_t j1,size_t j2)
m.diag()
m.diag(int i)
m.diag(int i,size_t k1,size_t k2)
m.SubVector(size_t i, size_t j, int istep, int jstep, size_t size)
m.ptr()
m.cptr()
m.isconj()
m.isrm()
m.iscm()
m.isdm()
```

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 `m`.

4.4.3 Views

```
m.SubMatrix(int i1, int i2, int j1, int j2)
m.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 submatrix must be completely within the band.

```
m.SubBandMatrix(int i1, int i2, int j1, int j2, int newnlo, int newhi)
m.SubBandMatrix(int i1, int i2, int j1, int j2, int newnlo, int newhi,
        int istep, int jstep)
```

This returns a `BandMatrixView` of the some subset of a `BandMatrix`. The `newnlo` and `newnhi` parameters do not need to be different from the existing `nlo` and `nhi` if `i1 = j1` (ie. 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 `m` 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 `m.SubBandMatrix(0,5,1,6,0,2)`.

```
m.Rows(int i1,int i2)
m.Cols(int j1,int j2)
m.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.

```
m.Real()
m.Imag()
m.Transpose()
m.Conjugate()
m.Adjoint()
m.View()
tmv::BaseMatrix<T>* m.NewCopy()
tmv::BaseMatrix<T>* m.NewView()
tmv::BaseMatrix<T>* m.NewTranspose()
tmv::BaseMatrix<T>* m.NewConjugate()
tmv::BaseMatrix<T>* m.NewAdjoint()
m.LinearView()
m.ConstLinearView()
```

Be careful with the last two, since the vector views that are returned will include the extra elements which are not part of the matrix and which therefore may not be defined.

4.4.4 Functions

```
RT m.Norm1() = Norm1(m)
RT m.Norm2() = Norm2(m)
RT m.NormInf() = NormInf(m)
RT m.NormF() = NormF(m) = m.Norm() = Norm(m)
RT m.NormSq() = NormSq(m)
RT m.MaxAbsElement() = MaxAbsElement(m)
T m.Trace() = Trace(m)
T m.Det() = Det(m)
minv = m.Inverse() = Inverse(m)
m.Inverse(minv)
m.InverseATA(tmv::Matrix<T>& cov)
```

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

```
m.Zero()
m.SetAllTo(T x)
m.Clip(RT thresh)
m.SetToIdentity(T x = 1)
m.ConjugateSelf()
m.TransposeSelf()
Swap(m1,m2)
```

4.4.5 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
b2 = b1 / x

b3 = b1 + b2
b3 = b1 - b2
m2 = m1 + b
m2 = b + m1
m2 = m1 - b
m2 = b - m1

b *= x
b /= x

b2 += b1
b2 -= b1
m += b
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
b2 = b1 - x
b2 = x - b1
b += x
b -= x
```

The subroutines for `BandMatrix` arithmetic are:

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

MultMV(T x1, const GenBandMatrix<Tm>& m, const GenVector<Tv1>& v1,
        T x2, const VectorView<T>& v2)

AddMM(T x1, const GenBandMatrix<T1>& m1, T x2, const BandMatrixView<T>& m2)
AddMM(T x1, const GenBandMatrix<T1>& m1, T x2, const MatrixView<T>& m2)

MultMM(T x1, const GenBandMatrix<T>& m1, const GenMatrix<T>& m2,
        T x2, const MatrixView<T>& m3)
MultMM(T x1, const GenMatrix<T>& m1, const GenBandMatrix<T>& m2,
        T x2, const MatrixView<T>& m3)
MultMM(T x1, const GenBandMatrix<T>& m1, const GenBandMatrix<T>& m2,
        T x2, const BandMatrixView<T>& m3)
```

4.4.6 Division

The division operations are:

```
v2 = v1 / b
m2 = m1 / b
m2 = b / m1
m = b1 / b2
v /= b
m /= b
```

plus all of these with `%` rather than `/`.

`BandMatrix` has three possible choices for the division decomposition:

1. `m.DivideUsing(tmv::LU)` does a normal LU decomposition.

   This 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.

   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`) and 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.

   This is the default decomposition to use for a square `BandMatrix` if you don't specify anything.

   To access this decomposition, use:

   ```
   m.BandLUD().IsTrans()
   m.BandLUD().GetL()
   m.BandLUD().GetU()
   m.BandLUD().GetP()
   ```

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

```
Matrix<T> m2(m.nrows,m.ncols);
MatrixView<T> m2v = m.BandLUD().IsTrans() ? m2.Transpose() : m2.View();
m2v = m.BandLUD().GetL() * m.BandLUD().GetU();
m2v.ReversePermuteRows(m.BandLUD().GetP());
```

2. `m.DivideUsing(tmv::QR)` will perform a QR decomposition.

   The same kind of convolutions need to be done to perform this in place as for the LU decomposition.

   This is the default method for a non-square `BandMatrix`.

   To access this decomposition, use:

   ```
   m.BandQRD().IsTrans()
   m.BandQRD().GetQ()
   m.BandQRD().GetR()
   ```

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

   ```
   Matrix<T> m2(m.nrows,m.ncols);
   MatrixView<T> m2v = m.BandLUD().IsTrans() ? m2.Transpose() : m2.View();
   m2v = m.BandLUD().GetL() * m.BandLUD().GetU();
   m2v.ReversePermuteRows(m.BandLUD().GetP());
   ```

3. `m.DivideUsing(tmv::SV)` will perform a singular value decomposition.

   This cannot be done in place.

   To access this decomposition, use:

   ```
   m.BandSVD().GetU()
   m.BandSVD().GetS()
   m.BandSVD().GetV()
   ```

   The product of these three (using `DiagMatrixViewOf(m.BandSVD().GetS())` for S) should result in a matrix numerically very close to `m`.

   There are the same control and access routines as for a regular SVD:

   ```
   m.BandSVD().Thresh(RT thresh)
   m.BandSVD().Top(size_t nsing)
   m.BandSVD().Condition()
   m.BandSVD().GetKMax()
   ```

   There are also the `SVS`, `SVU`, and `SVV` options for the decomposition which cannot be used for division, but which can access parts of the decomposition.

The routines

```
m.SaveDiv()
m.SetDiv()
m.ReSetDiv()
m.UnSetDiv()
m.DivideInPlace()
```

work the same as for regular `Matrix`es.

4.4.7 Input/Output

The simplest output is the usual:

```
os << m
```

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`:

```
m.WriteCompact(os)
```

outputs in the format:

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

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

```
tmv::BandMatrix<T> m(nrows,ncols,nlo,nhi);
is >> m;
tmv::BandMatrix<T>* m2;
is >> m2;
```

One can write small values as 0 with

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

5 Errors and Exceptions

There are two kinds of errors that I 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 which 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`.
- Calling `Rank1Update` with `x2` not 0 or 1.
- Viewing a `Matrix` as a `HermMatrix` when the diagonal isn't real.
- Calling `m.LUD()`, `m.QRD()`, etc. for a `Matrix` which 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, that is.)

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 in the depths of the various algorithms, not in these $O(1)$ time checks that the dimensions are correct and such.

The other kind of error is one 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 which 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 problems with the data or files used by the code. So they could still happen even after the code has been thoroughly tested.

Both kinds of errors currently throw the same exception: `tmv::tmv_error`, which is an empty class with no further information. They also output a string which indicates what the error is. Not the best C++ error handling, I know. I should really derive some more useful classes from this with more information about what cased the error, and only print the string if `tmv_error` isn't caught. But for now, feel free to catch `tmv_error` if you want.

6 Advanced Usage

6.1 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 or a matrix respectively. However, occasionally you may want to multiply each element in a vector by the corresponding element in another vector: $v(i) = v(i) * v2(i)$.

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 $v2(i) = x * v1(i) * v2(i)$, and the second performs $v3(i) = v3(i) + x * v1(i) * v2(i)$ for $i = 0..N-1$ (where N is the size of the vectors).

There is no operator overloading for `Vector`s which is equivalent to these. But they are actually equivalent to the following forms:

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

respectively. These statements inline to the above function calls automatically. Depending on your preference, they may or may not be clearer as to what you are doing.

There is currently no corresponding function for `Matrix`es. However, so long as the matrices you would want to do this on are the same storage type, you can perform $m2(i,j) = x * m1(i,j) * m2(i,j)$ by using the `LinearView` method:

```
ElementProd(x,m1.ConstLinearView(),m2.LinearView());
```

6.2 QR Decomposition, Update, Downdate

Often, it can be useful to create and deal with the QR decomposition directly, rather than just relying on the division routines. Primarily, this is because of 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, the matrix version of this is: $Ax = b$, where A is a matrix with more rows than columns (ie. it is tall). (A is commonly called the "design matrix".) It may be the case that you have more rows (ie. 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 square 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.

The better solution is normally to do use the QR decomposition, $A = QR$, so

$$\begin{aligned} QRx &= b \\ x &= R^{-1} Q^\dagger b \end{aligned}$$

But if A is too large to fit in memory, then so is Q.

The 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 a Cholesky decomposition of $A^\dagger A$. So this method produces a more accurate answer for x than the normal equation does.

How do you calculate R?

First, we use the lemma that a product of unitary matrices is also unitary. This implies that if we can calculate something like: $A = Q_0 Q_1 Q_2 ... Q_n R$, then this R is what we want.

So, consider breaking A into a submatrix, A_0, which can fit into memory, plus the remainder, A_1, which we know how to compute, but haven't done so yet.

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

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 was tall, 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 decomposition to get R.

For numerical reasons, the fewer such iterations you do, the better. So you should try to make the top submatrix as large as possible each time, given the amount of memory available.

The routines to do this with the TMV library are:

```
QR_Decompose(const MatrixView<T>& A, const VectorView<T>& Qbeta);
UpperTriMatrix<T> R = UpperTriMatrixViewOf(A);
QR_Update(const UpperTriMatrixView<T>& R, const GenMatrix<T>& A)
```

The first subroutine performs the QR decomposition of `A` in place, with Q stored in a compact form below the diagonal of `A` and in the extra vector `Qbeta`. Since we don't need Q, you can throw away `Qbeta` after this call. `R` is returned in the upper triangle of `A` as shown.

Then subsequent updates would use the second subroutine, `QR_Update`. For this routine, `A` represents any number of additional rows to add to the QR decomposition. If you want, you could even just start with `R` set to an identity matrix, and only use `QR_Update`, rather than calling `QR_Decompose` for the first pass.

Furthermore, it is common to do some kind of outlier rejection with a least-square solution. This is basically the opposite: 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:

```
bool QR_Downdate(const UpperTriMatrixView<T>& R, const GenMatrix<T>& A)
```

Now `A` represents the rows to remove from the QR decomposition. Note that this only finds the new R, not Q.

It is possible for the downdate to fail if the matrix A does not represent rows of the matrix which 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` returns `false`. A successful downdate returns `true`.

6.3 Other SymMatrix operations

There are three routines which we provide for `SymMatrix`, which do not have any corresponding shorthand with the usual arithmetic operators.

The first two are:

```
Rank2Update(T x1, const GenVector<T1>& v1, const GenVector<T2>& v2,
        int x2, const SymMatrixView<T>& m)
Rank2KUpdate(T x1, const GenMatrix<T1> m1, const GenMatrix<T2>& m2,
        int x2, const SymMatrixView<T>& m3)
```

These are similar to the `Rank1Update` and `RankKUpdate` routines. A rank-2 update calculates

```
m = x1 * (v1 ^ v2) + x1 * (v2 ^ v1) + x2 * m
m = x1 * (v1 ^ v2.Conjugate()) + conj(x1) * (v2 ^ v1.Conjugate()) + x2 * m
```

for a symmetric or hermitian `m` respectively. Likewise, a rank-2k update calculates:

```
m3 = x1 * m1 * m2.Transpose() + x1 * m2 * m1.Transpose() + x2 * m3
m3 = x1 * m1 * m2.Adjoint() + conj(x1) * m2 * m1.Adjoint() + x2 * m3
```

for a symmetric or hermitian `m` respectively.

We don't have an arithmetic operator shorthand for these, because, as you can see, the operator overloading required would be quite complicated. And since they are pretty rare, I decided to just let the programmer call the routines explicitly.

The other routine is:

```
SymMultMM(T x1, const GenMatrix<T>& m1, const GenMatrix<T>& m2,
        int x2, const SymMatrixView<T>& m3)
```

This calculates the usual generalized matrix product: `m3 = x1*m1*m2 + x2*m3`, 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 which the programmer can know that the product should be (at least numerically close to) symmetric, so the calculation is ok. Therefore it is provided as a subroutine.

In all three of these cases, `x2` must be either 0 or 1.

7 Obtaining and Compiling the Library

I haven't done all the official stuff needed for it yet, but the intent is for this code to be licensed according to the Gnu General Public License. See http://www.gnu.org/copyleft/gpl.html for more details about the GPL. The basic idea is that all source code is public, and you can use the code for your own purposes as much as you want. If you incorporate it into a program that you plan to distribute, then that code must be licensed according to the GPL.

Specifically, you may not incorporate it into proprietary software. If you would like to obtain a license for use with proprietary software, contact michael [at] jarvis [dot] net, and we can discuss pricing.

OK, now that you know how you are allowed to use the code, here's how you get it:

1. Go to http://www.hep.upenn.edu/~mjarvis/tmv/ for a link to a tarball with all of the source code.
2. Copy the tarball to whatever directory you want.
3. Unpack the tarball:
   ```
   gunzip tmv0.53.tar.gz
   tar xf tmv0.53.tar
   ```
4. Edit the makefile.

   There are a whole bunch of pairs of lines which define `CFLAGS1`/`ODIR`, all but one (pair) of which should be commented. There are also several definitions for `CC`, exactly one of which should be uncommented. The default uncommented lines are:
   ```
   CC= /usr/bin/g++
   CFLAGS1= -O3 -g -Wall -Werror -ansi -DNOBLAS -DNDEBUG
   ODIR= gcc-noblas
   ```
   These lines define which compiler to use, the compiler flags, and in what directory to put the *.o files.

   The `CFLAGS1` and `ODIR` lines are paired up together (along with a corresponding `CC` line) to make it easier to have multiple versions of the code. For example, you may want one with debugging turned on (These have `-debug` in the `ODIR` line, and no `-DNDEBUG` in the `CFLAGS1` line.) and one with it turned off. Likewise, you may want a version with BLAS and/or LAPACK, and another without. The structure of the makefile is intended to make that easier to do.

   Here are some compiler 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.
   - `-DNOBLAS` will not call any external BLAS or LAPACK routines
   - `-DNOLAP` will not call any external LAPACK routines
   - `-DATLAS` will only call the external LAPACK routines that are defined by ATLAS. It also sets up the LAPACK calling style for the ATLAS definitions.
   - `-DMKL` will call all the external BLAS and LAPACK routines. It also sets up the LAPACK calling style for the MKL definitions.
   - `-DNOFLOAT` will not instantiate any `<float>` classes or routines.
   - `-DLONGDOUBLE` will instantiate `<long double>` classes and routines.

- `-DXTEST` will do extra testing in the test suite. (It doesn't change anything about the library.) 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 (without `XTEST` works ok for you, you should be fine. But if you feel like doing more tests, feel free to turn it on. `-DTMVFLDEBUG` will do extra (slow) debugging. Specifically, it checks to make sure element access is always within the range of the memory that was allocated. This shouldn't ever be necessary for a released version, since I should have already discovered errors like this before the official release. But if you get strange segmentation faults, you could give it a try to help me fix the problem.
- `-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 help me fix the problem.
- `-DTMV_BLOCKSIZE=NN` will change the block size used by some routines. Unless you have a strange CPU, I doubt this will speed things up much. But feel free to play around with it if you want.

Below these, there are lines defining `BLASLIBS`, `CLIBS`, and `CFLAGS` which you may want to change. The first one especially may need to be changed depending on the BLAS library you want to use. The default is set for the Intel MKL linkage requirements. If you are compiling with the `-DNOBLAS` flag, you can just comment out the `BLASLIBS` definition (ie. add a `#` in front of it).

The rest of the makefile should be ok.

5. (advanced usage) Edit Inst and Blas files

   Note that, by default, the library will include instantiations of all classes and functions which use `<double>`, `<float>`, or `<int>` (including complex versions). The flags `-DFLOAT` and `-DLONGDOUBLE` can change this as described above. But if you want to compile routines for some other class, say some user-defined `Quad` class, then you will need to modify the file `TMV_Inst.h`. Simply add the lines:

   ```
   #define T Quad
   #include InstFile
   #undef T
   ```

   to the end of the file.

   Also, the file `TMV_Blas.h` sets up all the BLAS and LAPACK calling structure, 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.

6. Run make.

   This will make the TMV library, libtmv.a, which will be located in the directory specified by `ODIR` in the makefile. If will also make an executable called tmvtest.

   Warning: this step will take a long time to finish. So plan on letting this run for a couple hours (depending on your machine and compiler of course). Many hours for gcc 3.3. Don't worry - once the library is compiled, compiling your code which uses the library is pretty quick.

I have tested the code using g++ versions 3.3.5, 3.4.4, 4.0.0, and icc version 8.0. It should work with any ansi-compliant compiler, but no guarantees if you use one other than these. Oh, and g++ 3.3 compiles the library much slower than the others, so I would recommend upgrading if you have this version.

Known compiler issues:

- Apple's perversion of gcc does not work for compiling the TMV library. If you want to compile this on a Mac, you should download the real gcc instead. I recommend using Fink (http://fink.sourceforge.net/)
- Version 9.0.21 of the icc compiler seems to do something very strange with the the object file, TMV_TestTriArith_A.o, when compiling with `-DXTEST`. It writes the file fine, but then the linker gives an error message that it can't read it when linking. My guess is that the object file is too large, since it is the largest .o file. My hope is that a later version of 9.0 has fixed whatever this problem is (I haven't checked this yet). But the workaround in the meanwhile is to compile this file without `-DXTEST`.
- The Apple linker can't handle the size of the test suite's executable when compiled with `-DXTEST`. So on this platform, you're stuck with just the shorter test suite.

  If you really want to run the extra tests, you should do one matrix type at a time. For example, have the main in TMV_Test.cpp only run the `BandMatrix` tests. And comment out the other "`TMV_TEST*_FILES=`" lines in the makefile. Repeat for the other matrix types.

7. Run tmvtest.

   This should output a bunch of lines reading `[`*Something*`] passed all tests`. If it ends in a line that starts with `Error`, then email michael [at] jarvis [dot] net about the problem.

8. Compile you program

   Each .cpp file that uses TMV will need to have `#include "TMV.h"` near the top. (If you are using some of the sparse matrices, then you need to include their .h files as well.)

   To compile it, you will need to use the compile flag `-I [tmv directory]` when making the object file to tell the compiler where the TMV*.h files are.

   For the linking step, you need to compile with the flags `-L [tmvlib directory] -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 Intel's MKL version of the BLAS and LAPACK routines, I also 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.

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 michael [at] jarvis [dot] net.

1. **Eigenvalues and eigenvectors**

   I still need to figure out what interface I want for this to be as intuitive as possible. Probably have a contained object which contains the eigenvalue stuff similar to the way I keep the decompositions for dividing.

   The code does calculate eigenvalues and eigenvectors of hermitian matrices already, but the interface isn't intuitive. The SVD of a hermitian matrix is the eigenvector decomposition. Well almost anyway. Technically, the singular values are defined to be positive, whereas the eigenvalues may be negative. But when I calculate the SVD for a `HermMatrix`, I don't bother making them positive, since I figured people might want to access the actual eigenvalues. The columns of V are the corresponding eigenvectors.

   The code doesn't currently have any mechanism for calculating eigenvalues for non-hermitian matrices.

2. **SVD algorithm**

   So far, I have a fast bidiagonalization (or tridiagonalization for symmetric/hermitian matrices), but my calculation of the SVD of a bidiagonal matrix uses the relatively slow QR algorithm, rather than the divide-and-conquer approach, which LAPACK uses. Also, they have a fast calculation of the eigenvectors of a hermitian matrix using a "Relatively Robust Representation" algorithm, which is also much faster than my algorithm.

3. **More sparse matrix varieties**

   Block-diagonal, symmetric/hermitian band, generic sparse...

4. **Mixed sparse matrix arithmetic**

   Mixing different kinds of sparse matrices in arithmetic operations currently will not compile for many pairs of sparse matrices. For example you cannot currently multiply a `BandMatrix` by a `SymMatrix`.

   There are three work-around options which should work for most pairs. For some things you can use the "ViewAs" commands to make the matrix types match. For example an `UpperTriMatrix` times a `BandMatrix` can be accomplished with:
   `BandMatrix<T> b2 = BandMatrixViewOf(u) * b1;`

   When this isn't possible, another solution is to assign one matrix to where you want the solution and then `*=` (or whatever) the other matrix.

   You could also cast one of them as a reguar `Matrix` which will use extra storage, but if that doesn't matter to you, it's probably the easiest option.

5. **Symmetric matrix arithmetic**

   There are some issues with `SymMatrix` arithmetic (see the discussion in that section). I haven't thought of a really nice way to solve this yet. Maybe it is related to a nice solution of the previous item.

6. **Packed storage**

   Triangle and symmetric matrices. can be stored in (approximately) half the memory as a full NxN 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.

7. **SmallMatrix class**

   My algorithms are generally optimized for large matrices. This is reasonable, since most code with both large and small matrices will be limited by the speed of the large matrix calculations. However, if you are doing a lot of operations with 2×2 or 3×3 matrices, these might dominate the total calculation time.

   It would be nice to have a class `SmallMatrix<T,Nrows,Ncols>` which had the sizes as template arguments. Then I could specialize some of the calculations for common small matrices, like 2×2 and 3×3 which could be made much faster than the current versions.

   Probably also `SmallSymMatrix`, `SmallHermMatrix`, and of course `SmallVector`. Any others?

8. **Anti-symmetric and anti-hermitian matrices**

   Are these worth adding?

9. **RowMajor Bunch-Kauffman**

   The Bunch-Kauffman decomposition for RowMajor symmetric/hermitian matrices is currently $LDL^\dagger$, rather than $L^\dagger DL$. The latter should be significantly (30%?) faster. The current $LDL^\dagger$ algorithm is faster for ColMajor. (These comments are for Lower storage - the opposite holds for Upper storage.)

10. **Faster MultMV, MultMM**

    I think I can mimic some of the ATLAS-style structure for the matrix-vector product and matrix-matrix product calculations to speed up the code for non-BLAS calculations.

11. **CLAPACK**

    The LAPACK calls are currently set for the Intel MKL (Math Kernel Library), which uses LAPACK names that don't start with "`clapack_`". Nor do they have an underscore at the end. I'd like to make it easy for people to specify which LAPACK naming format they want to use to mesh with their LAPACK library: eg. start with "`clapack_`"? append an underscore? Also, the code isn't tested for any version of LAPACK other than the MKL version.

12. **Conditions**

    Currently, the SVD is the only decomposition which calculates the condition of a matrix. LAPACK seems to have routines to calculate some kind of condition from an LU decomposition (and others). I need to check out what they are doing, and probably add some similar capability.

13. **Division error estimates**

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

14. **Equilibrate matrices**

    LAPACK can equilibrate matrices before division. Again, I should include this feature too. Probably as an option (since most matrices don't really need it) as something like `m.Equilibrate()` before calling a division routine.

15. **Symmetric inverses**

    The routines for `sym_m.Inverse(SymMatrix& minv)` for the LU and SV decompositions currently do this a stupid way (accurate, but slow) - I need to speed it up.

16. **Tridiag * Matrix**

    There is a LAPACK routine for this (`lagtm`) which is not currently called.

17. **Sym LU Inverse**

    There is a LAPACK routing for this (`sytri`) which is not currently called.

18. **Element-by-element product of matrices**

    This exists for vectors. It may be worth writing versions for matrices.

19. **Exceptions**

    The error handling is fairly rudimentary, with any runtime error just throwing an empty class called `tmv_exception`. I need to flesh this out to be more specific as to what went wrong.

20. **Test I/O**

    My test suite, tmvtest, doesn't currently test the I/O. It's not that complicated, but there could be errors here that I haven't checked for yet.

21. **GPL**

    Add all the GNU General Public License text to each file. Also, I think I might need something from Penn to disclaim their rights, since most of the code was written while I have been working at Penn.

22. **Other Decompositions**

    There are some other matrix decompositions which can be interesting, but which are not useful for division. For example, the polar factorization: $A = UH$ where U is unitary and H is positive semi-definite.

23. **Documentation**

    This file could probably be a bit more comprehensive. Plus, I'll bet there are still some typos.

9 History

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 internal 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 sparse matrix options.

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. I 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 did an algorithm that used unit strides whenever possible, they didn't do any blocking which is key for large matrices.

Version 0.3 Finally, I actually named this one 0.3 at the time. (Actually, at first and also for 0.2, I was calling it JMV rather than TMV for Jarvis's Matrix/Vector library. Then I decided that was to egotistical, and T for template was better.) The big addition here was BLAS and LAPACK calls, which helped me 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 For this version, I added QR_Downdate. I needed this added functionality for a project I was working on, which is why this became a new version. Probably it only deserves a 0.01 increment, since there wasn't all that much else that was improved.

A few minor things: I improved the QRP decomposition. I added the possibility of not storing U,V for the SVD. I greatly improved the test suite, and correspondingly found and corrected a few bugs. I also added some arithmetic functionality that had been missing (like m += L*U).

Version 0.5 I finally added blocked versions of most of the algorithms, so the non-LAPACK code ran a lot faster. I also added symmetric and hermitian matrices. I allowed for loose QRP decomposition. I added the possibility of division in place, rather than copying to new storage.

Version 0.51 Some minor improvements: I sped up some functions like matrix arithmetic and assignment by adding the LinearView method which can view the matrix as one long vector. I added QR_Update. I blocked some more algorithms like TriMatrix multiplication/division, so non-BLAS code runs significantly faster (but still quite a bit slower than BLAS).

Version 0.52 The first "public" release! And correspondingly, the first with documentation and a website. A few other people had used previous versions, but since the only documentation was what I have in the .h files, it wasn't all that user-friendly.

I added SaveDiv() and related methods for division control. It used to be that the default was to reuse the matrix decomposition. This meant that naive code could get the wrong answer for routines which divide, change the matrix, and then divide again. (ReSetDiv had been available to redo the decomposition.) I decided it was better to have naive code be right, but slow, rather than fast and wrong. So now the programmer needs to explicitly say to save the decomposition.

I extended the test suite a bit and found a couple more (fairly obscure) bugs.

I also don't make as many temporary matrices for operations that use the same storage for multiple objects. For example

`m = LowerTriMatrixViewOf(m) * UpperTriMatrixViewOf(m)`
can be done in place now.

Version 0.53 By popular demand (well, request), I added the Fortran-style indexing.

Also fixed a bug in the Band LU Divider for very fat bands.