

FPCL Linear Algebra Module: Version 1.0

User's Guide

Leo Michelotti

November 10, 1997

The FPCL module `LinearAlgebra` contains two classes that emulate the mathematical concept of a matrix: `MatrixD` (equivalently, `Matrix`) and `MatrixC`, for matrices possessing real and complex elements respectively. We will use either “`Matrix`”, “`MatrixD`”, or “`MatrixC`” in sentences referring to the specific classes and simply “matrix” in sentences which can refer to either class or to the underlying abstract mathematical object. Correspondingly, the word “scalar” will mean either `Float8` or `Complex8` depending on the context.¹

This document is written in four sections. The first deals with the fundamental tasks of declaring and initializing matrices. Available matrix functions are described in the second. In addition to basic arithmetic, these include methods for inversion and factorization (esp., eigenanalysis). The third section discusses the concept of “data models,” which was *the key consideration* dictating the design of this package.² Finally, stream operators, which provide a mechanism for light persistence, are described in the last section.

Note: Scattered throughout this document are paragraphs in italics beginning with the word “Note.” This is an example of one. They convey information about deficiencies in Version 1.0 of the FPCL LinearAlgebra module, either features which have not yet been implemented or ones which have but perhaps not as well as they could be. It is intended that these deficiencies be corrected in later versions, at which time the corresponding paragraphs should be removed.

1 Declaration, initialization, and assignment

Matrices are declared (instantiated) with two integer arguments indicating row and column dimensions. Thus, “`Matrix x(12, 7);`” would be used to instantiate `x` as a matrix with 12 rows and 7 columns. If the matrix is square, then an alternative constructor takes *one* integer argument, specifying the dimension, and a `Float8` argument, indicating the initial value along the diagonal: for example, “`Matrix x(3, 1.0);`” would result in a 3×3 unit matrix, while either “`Matrix x(3, 3);`” or “`Matrix x(3, 0.0);`” would instantiate a 3×3 zero matrix. By default, without any arguments – as in “`Matrix a;`” – a 2×2 zero matrix is produced. Finally, an optional `Float8 []` argument can be used to initialize a `Matrix` from an array of values. These various possibilities are listed below.

¹“`Float8`” is a fixed type specified in the FPCL file `fixedtypes.h`. It denotes a variable of eight bytes emulating a real number. Almost always, this is the same as “`double`.” “`Complex8`” is not found in `fixedtypes.h`, but it should be.

²The ability to add data models to the package was introduced at the request of potential users, strongly expressed during an FPCL workshop held at Fermilab.

```

Float8 w[16];
...
Matrix a;                // By default, a 2x1 matrix of zeroes.
Matrix b( 3, 5 );        // A 3x5 matrix of zeroes.
Matrix c( 7, 3.1416 );   // A 7x7 matrix with 3.1416 on the diagonal.
Matrix d( 7 );           // A 7x1 column matrix.
Matrix e( 2, 8, w );     // The sixteen components of w are loaded
                        // into a 2x8 matrix.
Matrix f( 4, w );        // The sixteen components of w are loaded
                        // into a 4x4 matrix.

```

Similar constructions are valid for `MatrixC` objects.

```

Complex8 w[16];
...
MatrixC a;
MatrixC b( 3, 5 );
MatrixC c( 7, w[0] );
MatrixC d( 7 );
MatrixC e( 2, 8, w );
MatrixC f( 4, w );

```

After a `Matrix` has been instantiated, numbers can be assigned directly to its elements.

```

Matrix a(3,2);
a(0,0) = 0.0;   a(0,1) = 0.1;
a(1,0) = 1.0;   a(1,1) = 1.1;
a(2,0) = 2.0;   a(2,1) = 2.1;

```

Notice that the indexing begins with zero, not one; life is too short to fight against the natural indexing scheme of both C and C++, so learn to live with it. The same operator is used to access matrix elements for use.

```

Matrix a(3,2);
Float8 x = a(1,0);

```

It is also possible to initialize a matrix directly from an array *after* its instantiation by using the **.loadFrom** member function.

```

Matrix a(3,2);
Float8 numbers [] = { 0.0, 0.1,
                     1.0, 1.1,
                     2.0, 2.1 };
a.loadFrom( numbers );

```

As noted earlier, by specifying a `Float8[]` argument when declaring a `MatrixD`, the array specified is used to initialize it.³ **.loadFrom** works in a similar manner except that the `Matrix` has already been initialized. If either method is used, *you* take the responsibility of assuring that the array in the argument has the correct size. No error condition is thrown if it does not, but the application obviously may produce incorrect results.

Finally, values can be loaded into a `Matrix` from a stream. We will postpone the discussion of this option until Section 4.

³A similar statement holds for `MatrixC` except that a `Complex8[]` argument is needed.

2 Functionality

To physicists and mathematicians – indeed, to anyone except a computer scientist – a matrix is not merely a container for storing data. It must possess algebraic and analytic functionality. In this section we describe what is available in FPCL/LinearAlgebra Version 1.0.

2.1 Arithmetic

Physicists are comfortably familiar with matrices – regardless of whether the comfort is justified – so the interface for matrix arithmetic needs little introduction. The operators `+`, `*`, and `-` behave as one would naively expect when sandwiched between two `MatrixD` or `MatrixC` objects. Simple arithmetic statements like

```
Matrix a(8,5), b(5,7), c(8,7), x(8,7);
...
x = ( a*b + c );
```

need no further explanation. In addition, the operators `+=`, `-=`, and `*=` are available and work as expected. However, remember that matrix multiplication is non-commutative, so that “`x *= y;`” is equivalent to “`x = x*y;`” but *not* to “`x = y*x;`”. Finally, mixed mode arithmetic with scalar variables is permitted when it makes sense.

```
Matrix a( 4, 4 ), b( 4, 4 );
Matrix x( 3, 4 ), y( 3, 4 );
b = 1.0 + 3.2*a;    // Okay
y = 1.0 + 3.2*x;    // Wrong. A scalar cannot be added
                    // to a non-square matrix.
```

2.2 Math functions

Any math function that can be applied to a `double` or `complex` variable can also be applied to a square matrix. The one exception is `abs` (or `fabs`), for which there is no natural extension to matrices. Code fragments like

```
Matrix x( 9, 9 );
...
x = sin(x) + cos(sqrt(x));
```

are legitimate, but the application program must include the header file `MtrDMath.h` before using these functions.

Note: Math functions should be available for both the `MatrixD` and `MatrixC` classes. At the time of writing, they are only available for `MatrixD`. They also work only with matrices that can be diagonalized, although this too should not be a restriction.

2.3 Inversion

Purposely, there is no binary operator specifying division of one matrix by another. The expression “`A/B`” could be interpreted either as $B^{-1} \cdot A$ or as $A \cdot B^{-1}$.⁴ Because there is no *natural* way to resolve the am-

⁴When the FPCLTF was polled, they split down the middle as to which should be the “correct” meaning.

biguity, we are avoiding future headaches by accomplishing matrix inversions with member functions **.inverse**, **.solve**, and **.pseudoInverse**.

The **.inverse** method takes zero or one argument and returns the inverse of the *square* matrix to which it is applied.

```
MatrixC y( 3, 5 ), m( 3, 3 ), x( 3, 5 );
Complex8 det;
...
y = m * x;
x = m.inverse() * y;
m = ( 1.0 + m ).inverse( &det );
```

If what is desired is the actual inverse of a matrix, as in the last line above, then this is the method that should be used. If the address of a scalar is supplied as the argument, then the determinant of the matrix is passed back through it. (Determinants can also be evaluated using the **.determinant** member function, as described later.) On the other hand, if one needs to solve a system of linear equations, as in the next to last line, it would be better to use **.solve**.

```
Matrix y( 3, 5 ), m( 3, 3 ), x( 3, 5 );
...
y = m * x;
x = m.solve( y );
```

This possesses the advantage of solving the linear system in place, rather than first computing the inverse and then performing a matrix multiplication. The solution is returned as a matrix with the same dimensions as the argument, which is not altered.

If a matrix has more rows than columns, the **.inverse** routine will fail. However, the Moore-Penrose pseudoinverse of such matrices can be calculated using member function **.pseudoInverse**.⁵

```
MatrixD y( 3, 5 ), m( 12, 3 ), x( 3, 5 ), b( 12, 5 );
...
y = m * x + b;
x = m.pseudoInverse() * y;
x = m.solve( y );
```

Apart from machine error, both lines produce the same output. The member function **.solve** works for non-square matrices and results in the same solution as produced by pseudoinversion.

Note: Only `MatrixD::inverse()`, `MatrixC::inverse()`, `MatrixD::pseudoInverse()`, and `MatrixC::pseudoInverse()` are available for use. The others involve nothing more than simple modifications of these, but they have yet to be written.

2.4 Factorizations

Matrices possess several standard factored forms. A few of them are available in Version 1.0 of the FPCL LinearAlgebra package: eigenanalysis, singular value decomposition, and polar factorization. Others may be added to later versions, if desired. In every case, a `MatrixD` member function returns a struct that contains the factors as its member data.

⁵Formally, the Moore-Penrose pseudoinverse of an $m \times n$ matrix \underline{A} is an $n \times m$ matrix \underline{A}^\dagger such that $\underline{A}^\dagger \cdot \underline{A} = \underline{1}$ and $\underline{A} \cdot \underline{A}^\dagger$ is idempotent (i.e., a projector). Intuitively, it is the “least squares fit” matrix.

2.4.1 Eigenanalysis

Given a square matrix, \underline{X} , the objective of eigenanalysis is to find an invertible matrix \underline{E} and a diagonal matrix $\underline{\Lambda}$ such that $\underline{X} \cdot \underline{E} = \underline{E} \cdot \underline{\Lambda}$, or equivalently, $\underline{X} = \underline{E} \cdot \underline{\Lambda} \cdot \underline{E}^{-1}$. The columns of \underline{E} are the eigenvectors of \underline{X} , and the diagonal elements of $\underline{\Lambda}$ are its eigenvalues. A `MatrixD` member function `.eigen` produces a “`MatrixEigenData`” struct containing the two matrices \underline{E} and $\underline{\Lambda}$ as data members `_vectors` and `_values` respectively. One way of obtaining these matrices from the struct is as follows.

```
Matrix x( 23, 23 );
MatrixEigenData w( x.eigen() );
MatrixC e( w._vectors );
MatrixC l( w._values );
```

In general, both the matrix of eigenvectors and the diagonal matrix of eigenvalues are complex.⁶

Note: For now, only `MatrixD` objects possess the `.eigen` member function. It should be extended to `MatrixC` objects as well.

2.4.2 Singular Value Decomposition

The singular value decomposition of a real matrix is a factorization into three matrices,

$$\underline{X} = \underline{U} \cdot \underline{D} \cdot \underline{V}^T,$$

where \underline{U} and \underline{V} are orthogonal, real matrices, and \underline{D} is a positive-definite diagonal matrix. Dimensionally, if \underline{X} is an $r \times c$ `MatrixD`, then \underline{U} will be $r \times c$ and both \underline{D} and \underline{V} will be $c \times c$. Further, it is assumed that $r > c$; that is, the matrix represents an overconstrained set of linear equations. Singular value decomposition is performed by the `MatrixD` member function, `.singularValueDecomposition`.⁷ Typical usage is illustrated below.

```
MatrixD x( 11, 5 );
...
MatrixSVDData w( x.SingularValueDecomposition() );
cout << x - ( w.u * w.d * w.v.transpose() ); // To machine accuracy,
// will write a Matrix of zeroes.
```

2.4.3 Polar factorization

Polar factorization of a real, square matrix, \underline{X} , expresses it in the form, $\underline{X} = \underline{P} \cdot \underline{\Theta}$, where \underline{P} is a positive-definite matrix and $\underline{\Theta}$ is orthogonal. It can be obtained by using the `MatrixD` member function `.polar`, as illustrated in the next fragment.

```
Matrix x( 4, 4 ), r( 4, 4 ), t( 4, 4 );
...
MatrixPolarData w( x.polar() );
r = w.rho;
t = w.theta;
```

⁶Contrary to popular opinion, not all matrices are Hermitian. In fact, not all matrices can be diagonalized. If this is unfamiliar, material on Jordan’s canonical form can be found in any good linear algebra textbook.

⁷A shorter, more natural name would have been `.SVD`, which is its actual acronym in the mathematics texts. However, the FP-CLTF felt that this would be mistaken for “silicon vertex detector.”

2.5 Miscellaneous utilities

`MatrixD` and `MatrixC` possess several convenient “utility” functions. These are listed below with a minimum of comment.

Method	Return type	Return value
<code>.rows()</code>	<code>int</code>	row dimension of a matrix
<code>.columns()</code>	<code>int</code>	column dimension of a matrix
<code>.transpose()</code>	<code>Matrix</code> or <code>MatrixC</code>	transpose of a matrix
<code>.determinant()</code>	<code>Float8</code>	determinant of a square matrix
<code>.trace()</code>	<code>Float8</code> or <code>Complex</code>	trace of a square matrix

3 Data models

One of the most important requirements communicated to the FPCL Task Force was the need for special data handling of specific kinds of matrices — diagonal, tri-diagonal, anti-symmetric, and so forth. The data handling procedures should take advantage of the matrix’s properties in order to decrease storage and increase performance. Especially emphasized by potential users was the desirability of treating small square matrices in an efficient manner that bypassed the generic matrix algorithms. In response to this request, a number of initial “data models” are available, and we expect that more will be added in the future.

3.1 Specifying a data model

A data model is activated via a corresponding member function of the form `.declareXXX`. To illustrate, consider that a 3×3 unit matrix can be instantiated simply by declaring it, as in the statement “`Matrix a(3, 1.0);`”. By default this assumes a generic data model which, accordingly, will use the generic matrix algorithms. To specify the 3×3 data model, with all of its presumed advantages, we proceed as follows.

```
Matrix a( 3, 1.0 );
a.declareM33();
```

On the other hand, if a programmer’s intention is that the `Matrix` remain *diagonal* throughout an application, then it may be better to specify a diagonal data model.

```
Matrix a( 3, 1.0 );
a.declareDiagonal();
```

After invoking `.declareDiagonal`, only three numbers will be stored and manipulated in operations with the matrix `a`. This all happens automatically, behind the scenes, in a manner completely transparent to the application programmer. At the application level, syntax for manipulating the matrix is unchanged. On the other hand, statements which violate its special nature will produce errors.

```
Matrix a( 3, 3 );
a.declareDiagonal();

a(0,0) = 0.0;           // These lines are OK
a(1,1) = 1.1;
a(2,2) = 2.2;
```

```
cout << a(0,0) << a(0,1); // a(0,1) will return zero

a(0,1) = 0.1; // This will produce an error message
```

Data models cannot be combined or activated simultaneously; there is, for example, no such thing as a “3 × 3 diagonal” data model. If an urgent need for one ever arises, it could be implemented and added to the LinearAlgebra library; in the absence of such demands, no such combinations exist. The result of the lines

```
Matrix a( 3, 3 );
a.declareM33();
a.declareDiagonal(); // This line nullifies the previous one.
```

is a diagonal model only.

Appropriate use of a **.declareXXX** function is predicated on the predefined dimensions of the matrix. As examples, consider the following correct and incorrect statements.

```
Matrix a( 5, 5 ); a.declareSymmetric(); // OK
Matrix b( 3, 7 ); b.declareSymmetric(); // Wrong!
Matrix c( 4, 4 ); c.declareM44(); // OK
Matrix d( 3, 3 ); d.declareM44(); // Wrong!
Matrix e( 9, 9 ); e.declareDiagonal(); // OK
Matrix f( 3, 9 ); f.declareDiagonal(); // Wrong!
```

Non-square matrices cannot be symmetric or diagonal, and the declared dimensions of a small square matrix should match the data model invoked.

We note in passing that a matrix that is declared and assigned in the same statement will automatically assume the dimensions and data model of its initializer. This *must* happen because of the way such statements work, *viz.*, by invoking the copy constructor.

```
MatrixC a( 137, 137 );
a.declareHermitian();
MatrixC b = a;
MatrixC c(a);
```

Except for the names of the variables, the third and fourth lines are completely equivalent, and each will instantiate a matrix possessing the Hermitian data model. However, in the example below,

```
MatrixC a( 137, 137 );
a.declareHermitian();
MatrixC b( 137, 137 );
b = a;
```

the matrix `b` remains generic; its data model will not automatically become “Hermitian.”

3.2 Behavior of `.loadFromArray`

The behavior of the member function **.loadFromArray** is affected when a non-generic data model is employed. Consider this next fragment.

```

Float8 aData [] = { 1., 2., 3.,
                   4., 5.,
                   6. };
Matrix a(3,3); a.declareSymmetric(); a.loadFromArray( aData );

Float8 bData [] = { 2., 3.,
                   5.
                   };
Matrix b(3,3); b.declareAntisymmetric(); b.loadFromArray( bData );

Float8 cData [] = { 1.,
                   4.,
                   6. };
Matrix c(3,3); c.declareDiagonal(); c.loadFromArray( cData );

```

After a data model is declared, only the significant matrix elements should be provided by the array. The components of `aData` represent the upper right elements of the symmetric matrix `a`, those of `bData`, the upper right elements of the anti-symmetric matrix `b` excluding the diagonal, and those of `cData`, the non-zero elements of the diagonal matrix `c`. Even though the extra data are neither specified nor stored, the value returned by `b(0, 1)` would be 2, that returned by `b(1, 0)` would be -2 , `c(1, 0)` would be 0, and so forth.

The reason for writing `.loadFromArray` in this way is the efficiency of using a simple `memcpy` operation to transfer numbers from the array to the matrix data storage area. It also easily prevents errors from being made, such as an anti-symmetric matrix with data that are not appropriate. If the syntax seems too confusing, it might be helpful to fill in the missing matrix elements with imbedded comments.

```

Float8 bData [] = { \* 0. *\ 2., 3.,
                   \* -2. 0. *\ 5.
                   \* -3. -5. 0. *\
                   };
Matrix b(3,3); b.declareAntisymmetric(); b.loadFromArray( bData );

```

Then again, it might not.

Note: The examples given above are a bit of a fraud because no Hermitian, symmetric, or anti-symmetric data models exist yet in the `LinearAlgebra` module. At the time of writing, the only data models that actually have been implemented are 2×2 (`.declareM22`) through 6×6 (`.declareM66`), diagonal (`.declareDiagonal`), and generic (by default, or use `.makeGeneric` as illustrated below).

3.3 Changing the data model

The data model associated with a matrix can be changed any number of times in the course of a program. This is done in one of two ways: either by the `.declareXXX` member functions as above, or by invoking the member functions `.makeXXX` instead. The difference between the two is that `.makeXXX` will perform regardless of the data stored in the matrix while `.declareXXX` requires that they possess the appropriate properties. Thus, for example,

```

Matrix x( 2, 1.0 );
x.declareDiagonal();

```

would work properly, while

```
Matrix x( 2, 1.0 );
x( 0, 1 ) = 1.0;
x.declareDiagonal(); // Wrong!
```

would not work at all. On the other hand,

```
Matrix x( 2, 1.0 );
x( 0, 1 ) = 1.0;
x.makeDiagonal(); // Okay
```

would ignore the off-diagonal element and result in a diagonal matrix. However, **.makeXXX** cannot be used to change the dimensions of a matrix.

```
Matrix x( 2, 5 );
x.makeM33(); // Wrong!
```

These functions can be used in tandem. For example, if the generic matrix `x` is mathematically supposed to be symmetric at some point in a calculation but is not numerically so because of machine roundoff error, exact symmetry can be enforced with the statement “`x .makeSymmetric() .makeGeneric();`”. The first operation will enforce symmetry (by averaging off-diagonal elements) while the second reverts `x` to its generic data model before proceeding with the program. In this way, matrices can be “cleaned up,” a somewhat dangerous operation that probably should be done rarely.

If an assignment statement is written between matrices possessing different data models, the data are transferred, if it makes sense to do so, but each matrix retains its own data model.

```
Matrix a( 11, 11 ), b( 11, 11 );
Matrix c( 8, 7 );
b .declareAntisymmetric();
...
a = b; // OK, but a remains a generic matrix.
c = b; // Wrong! Dimensions are incorrect.
b = a - a.transpose();
// Will work only if the right hand side
// is antisymmetric, to within some tolerance.
```

The classes could have been designed so that the statement “`c = b;`” would be acceptable, but that possibility was voted down by the FPCL Task Force. Instead, this restriction can be bypassed, if necessary, by assigning with the **.setEqualTo** member function.

```
Matrix a( 3, 8 ), b( 11, 11 );
b .declareAntisymmetric();
...
a.setEqualTo( 8.0*b );
```

The function **setEqualTo** causes `a` to change its data model to that of `b` or, more precisely, to that of its argument. This is useful when writing a function with matrix arguments, especially if one wants to absorb the data model of an argument and, possibly, even return the answer with the same model.

```
Matrix foo( const Matrix& x )
{
    Matrix ret;
    ret.setEqualTo(x);
    ...
    return ret;
}
```

3.4 RTTI

Run time type information (RTTI) provides a means for learning the type an object given only its address (i.e., a `void*` pointer). The RTTI global function `typeid` cannot be used directly to discover the data model associated with a `Matrix`; for example, a statement like

```
Matrix x;
...
cout << typeid(x).name();
```

would produce either “MatrixD” (or “MatrixC”) as its value, regardless of the underlying data model. It should never really be necessary to know this information, but for completeness, the public member function `.typeID` provides a capability for acquiring the data model associated with a `Matrix`. It returns the same `Type_info` object as `typeid` but now referring to the data model. For example,

```
Matrix x;
...
cout << x.typeID().name();
```

would result in “MLDGeneric” being written to the output stream.

3.5 New data models

The `LinearAlgebra` module was designed to possess a natural upgrade path for adding more data models in later versions, not only the anticipated ones, like *Symmetric* and *Hermitian*, but also unanticipated ones, such as *Markov* or *Hadamard*. An effort has been made to isolate the necessary changes so that the process of adding new data models would scale no worse than linearly with the number of already existing models. It is also likely that the module’s original author will not be the one to write all the additional data models. In order to make it easier for anyone to add them as needed, instructions can be found in the companion *Design Notes*.

Note: Of course, the Design Notes document does not yet exist.

4 Streaming: a light persistence mechanism

The `LinearAlgebra` module includes stream operators, providing a “light” persistence mechanism. A small demo program,

```
#include <stdlib.h>
#include "LinearAlgebra\Matrix.h"

void main()
{
    Matrix x( 3, 4 );
    for( int i = 0; i < 3; i++ ) {
        for( int j = 0; j < 4; j++ ) {
            x( i, j ) = (j+1)*exp(10*i);
        }
    }
    cout << x;
}
```

would produce the output,

```
begin MLDGeneric 3 X 4 - formatted
 1.000e+00  2.000e+00  3.000e+00  4.000e+00
 2.203e+04  4.405e+04  6.608e+04  8.811e+04
 4.852e+08  9.703e+08  1.455e+09  1.941e+09
end
```

If the output is written to a file, the data can be read back using the stream-in operator. For example, the program,

```
#include <stdlib.h>
#include <fstream.h>
#include "LinearAlgebra\Matrix.h"

void main()
{
  Matrix x( 3, 4 );
  for( int i = 0; i < 3; i++ ) {
    for( int j = 0; j < 4; j++ ) {
      x( i, j ) = (j+1)*exp(10*i);
    }
  }
  ofstream os( "test.dat" );
  os << x;
  os.close();
  ifstream is( "test.dat" );
  Matrix y;
  cout << y;
  is >> y;
  cout << y;
}
```

would result in the console output,

```
begin MLDGeneric 2 X 2 - formatted
 0.000e+00  0.000e+00
 0.000e+00  0.000e+00
end
begin MLDGeneric 3 X 4 - formatted
 1.000e+00  2.000e+00  3.000e+00  4.000e+00
 2.203e+04  4.405e+04  6.608e+04  8.811e+04
 4.852e+08  9.703e+08  1.455e+09  1.941e+09
end
```

Streaming provides the one and only manner in which an already declared Matrix can change its dimensionality. This is even more apparent in the following example,

```
#include <stdlib.h>
#include <fstream.h>
#include "LinearAlgebra\Matrix.h"
```

```

void main()
{
  Matrix  x( 3, 4 ), y( 2, 2 );  y.declareM22();
  Matrix* z[2];  z[0] = &x;  z[1] = &y;

  int i, j, k;
  for( k = 0; k < 2; k++ ) {
    for( i = 0; i < z[k]->rows(); i++ ) {
      for( j = 0; j < z[k]->columns(); j++ ) {
        (*z[k])( i, j ) = (j+1)*exp(10*i);
      }
    }
  }

  ofstream os( "test.dat" );
  os << x << y;
  os.close();

  ifstream is( "test.dat" );
  Matrix w;
  for( i = 0; i < 2; i++ ) {
    is >> w;
    cout << w;
  }
}

```

with console output,

```

begin MLDGeneric 3 X 4 - formatted
 1.000e+00  2.000e+00  3.000e+00  4.000e+00
 2.203e+04  4.405e+04  6.608e+04  8.811e+04
 4.852e+08  9.703e+08  1.455e+09  1.941e+09
end
begin MLD22 2 X 2 - formatted
 1.000e+00  2.000e+00
 2.203e+04  4.405e+04
end

```

Both data model and dimensionality are preserved by the stream operators. While a little dangerous, this flexibility allows a user to read a Matrix from a file without requiring that he⁸ know these details ahead of time.

Note: It may be decided by higher FPCLTF authority that this capability is indeed too dangerous and should be eliminated. Personally, I hope that it is allowed to stand.

The examples written so far do not illustrate true persistence, since numbers are written to the ASCII file with only four significant digits. Streaming formats can be changed in a number of ways. True persistence is provided by specifying a binary format as follows.

⁸Yes, yes: or she.

```

#include <stdlib.h>
#include <fstream.h>
#include "LinearAlgebra\Matrix.h"

void main()
{
    Matrix  x( 2, 3 ), y;
    int i, j;
    for( i = 0; i < 2; i++ ) {
        for( j = 0; j < 3; j++ ) {
            x( i, j ) = (j+1)*exp(10*i);
        }
    }

    cout << "x:\n" << x;

    { ofstream os( "test.dat" );
      os << x;
      os.close();
      ifstream is( "test.dat" );
      is >> y;
      cout << "Difference: ASCII:\n" << x - y;
    }

    OutputFormat myformat;
    myformat.dumpbase = BIN;
    x.setOutputFormat( &myformat );

    { ofstream os( "test.dat", ios::binary | ios::out );
      os << x;
      os.close();
      ifstream is( "test.dat" );
      is >> y;
      cout << "Difference: Binary:\n" << x - y;
    }
}

```

After declaring an OutputFormat object, myformat, and setting its **.dumpbase** field to BIN, it is assigned to the Matrix x using the member function **.setOutputFormat**. The argument is myformat's address; in fact, myformat is not copied, which allows matrices to share the same OutputFormat. Notice that the output file stream itself must be opened with the flags "ios::binary | ios::out." Omitting this may or may not produce erroneous results, depending on the operating system and the size of the matrix. This demo would produce the console output:

```

x:
begin MLDGeneric 2 X 3 - formatted
 1.000e+00  2.000e+00  3.000e+00
 2.203e+04  4.405e+04  6.608e+04
end
Difference: ASCII:
begin MLDGeneric 2 X 3 - formatted

```

```

0.000e+00 0.000e+00 0.000e+00
-3.534e+00 2.932e+00 -6.026e-01
end
Difference: Binary:
begin MLDGeneric 2 X 3 - formatted
0.000e+00 0.000e+00 0.000e+00
0.000e+00 0.000e+00 0.000e+00
end

```

As expected, the binary stream provides an exact replica of the original, but the ASCII one does not. The supported possibilities for the **.dumpbase** field are: BIN for a binary dump of bits and DEC, OCT, and HEX, for ASCII files with data written in decimal, octal, and hexadecimal format respectively. In the latter three cases, `OutputFormat` objects also possess **.width** and **.precision** fields which allow the user to specify those as well.

The Matrix classes possess a default `OutputFormat` which governs the output of all matrices whose **.setOutputFormat** member functions have not been invoked. These defaults are global variables named `MLD::defaultOutputFormat`, for class `MatrixD`, and `MLC::defaultOutputFormat`, for class `MatrixC`. By changing their fields at any point in a program, the user can modify them to suit his needs, thereby affecting the (default) stream formatting for all matrices simultaneously. Even after the stream format of a matrix has been changed, it can be easily returned to the default value.

```

MatrixC z;
...
z.setOutputFormat( &MLC::defaultOutputFormat );

```