

FPCL Linear Algebra Module
Design Notes

Leo Michelotti

December 31, 1997

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 3 |
| 2 | Key design ideas | 3 |
| 2.1 | Envelope-letter idiom | 3 |
| 2.2 | Lazy evaluation | 4 |
| 2.3 | Accessors | 6 |
| 2.4 | Double dispatching | 8 |
| 3 | Adding new data models | 10 |
| 3.1 | Assign an index. | 10 |
| 3.2 | Declare .declare | 10 |
| 3.3 | Write the new MLDXXX and/or MLCXXX header files | 10 |
| 3.4 | Write the model's source file | 11 |
| 3.5 | Update the factories | 11 |
| 4 | Implement new data models | 11 |
| 4.1 | Utility functions | 11 |
| 4.2 | .makeXXX and .declareXXX | 16 |
| 4.3 | Constructors | 17 |
| 4.4 | Registering with the DVFT | 18 |
| 4.5 | Additional arithmetic operations | 22 |
| 4.6 | Streaming methods | 25 |
| 4.7 | Special functions | 27 |
| A | Member functions of classes MLCDiag and MLDDiag | 29 |

There is no doubt that Marley was dead. This must be distinctly understood, or nothing wonderful can come of the story I am going to relate.

— Charles Dickens
A Christmas Carol

1 Introduction

Like Jacob Marley’s death, there is a central fact that must be understood or the rest of this document will make no sense whatsoever. On January 13, 1997 was held a “Fermilab Physics Class Libraries Task Force Contents and Priorities Workshop,” at which comments were solicited from the FPCLTF’s (potential) future users on desired features for the FPCL packages. A dominant requirement set for the LinearAlgebra package was that it possess a capability for taking advantage of properties belonging to certain categories of matrices. For example: finding the inverse of a 2×2 , a 3×3 , or a diagonal matrix is a trivial task and should be treated as such; storage requirements for an anti-symmetric matrix should be less than half that of a generic matrix; a matrix multiplication can be done more quickly when one of the operands happens to be diagonal; and a large body of efficient algorithms have been developed for use with sparse matrices. *FPCL’s LinearAlgebra package was principally designed as an attempt to satisfy this requirement by giving matrices access to specialized “data models,”* each possessing its own efficient computational methods. You must understand and accept that or this document will appear to describe a superb example of converting an essentially clean, simple object into a convoluted monstrosity.

In the next section we will outline the key ideas upon which are built the technique used for realizing the “data models” concept in software. That is followed by two others which describe the process of adding new “data models” – that is, new categories of matrices – to the library and implementing them. LinearAlgebra is not meant to be a closed, finished package; it is open-ended, able to incorporate new data models as they become necessary, or just desirable. Information necessary to extend the library is contained in these *Design Notes*. In writing it, I assumed that readers already will have had some familiarity with LinearAlgebra, at least to the point of having scanned its *User’s Guide*.

2 Key design ideas

A high degree of polymorphism is the key approach behind the FPCL LinearAlgebra package. The polymorphism is attributed not to the `Matrix` objects themselves but to their data. The four subsections below describe the most important strategies that were used to make polymorphism work: (a) the envelope-letter idiom, (b) lazy evaluation, (c) accessors, and (d) double dispatching.

2.1 Envelope-letter idiom

The “data model” concept was realized within the framework of the envelope-letter idiom. Here, the data associated with a matrix, i.e., the matrix elements, are *not* contained within the `Matrix` object instantiated in an application program. Rather, the `Matrix` contains a pointer to another object which actually possesses the data. The `Matrix` is considered an “envelope,” which is all that an application program is allowed to handle, while the data object is the “letter,” hidden within the “envelope” and invisible to the program. The only private datum contained within the classes `MatrixD` and `MatrixC` is this pointer.

```

class MatrixD {
...
private:
    MLD* _pml;
...
};

```

```

class MatrixC {
...
private:
    MLC* _pml;
...
};

```

MLD and MLC are abstract (pure virtual) classes (actually structs) which serve as bases from which are derived implementations of specific data models. Some relevant data associated with MLD are shown below.

```

struct MLD {
...
    int      _r, _c;
    int      _rc;
    Float8*  _pdi;
    Float8*  _pdf;
...
};

```

MLC's data are similar except that `Float8` is replaced by `Complex8`. `_r` and `_c` refer to the row and column dimensions of the matrix. `_rc` is the reference count, about which more later. The matrix elements are contained in a linear array, with first entry at address `_pdi` and last at `_pdf`. While the MLD and MLC constructors take the responsibility of setting `_r`, `_c`, and `_rc` correctly, constructors for the derived classes must set `_pdi` and `_pdf`, because the base class does not “know” how information about matrix elements is stored in the data models. We will see examples of this in Section 4.3.

2.2 Lazy evaluation

The tremendous advantage of the envelope-letter idiom is that it enables us to avoid copying data unnecessarily. To begin with, copy constructors need not copy potentially large amounts of data when a `Matrix` is passed as an argument to a function or when it is returned from a function. Only the data pointer needs to be reproduced.

```

MatrixD::MatrixD( const MatrixD& x )
{
    _pml = x._pml;
    (_pml->_rc)++;
}

```

Notice that the reference count, `_rc`, is increased, because there is a new `Matrix` that refers to the same data. Conversely, the `Matrix` destructor decreases the reference counter and destroys the data if no other `Matrix` points to it.

```

MatrixD::~MatrixD()
{
    if( --(_pml->_rc) == 0 ) delete _pml;
}

```

This is, by now, a standard technique in C++ programming.

We also avoid unnecessary duplication by the assignment operator whenever possible, using a strategy called “lazy evaluation.”

```

MatrixD& MatrixD::operator=( const MatrixD& x )
{
    if( _pml != x._pml ) {
        if( ( _pml->_r == x._pml->_r ) && ( _pml->_c == x._pml->_c ) ) {
            // If the same type, just change the pointer.
            if( typeid( *_pml ) == typeid( *(x._pml) ) ) {
                // First disconnect from old data.
                if( --(_pml->_rc) == 0 ) delete _pml;
                // Reset the data pointer and update counter
                _pml = x._pml;
                (_pml->_rc)++;
            }
            // If not the same type, then copy the data.
            else {
                PREPFORCHANGE( _pml )
                _pml->loadFrom( x._pml );
            }
        }
        // Dimensions were not correct.
        else MatrixD::error( "MatrixD::operator=( const MatrixD& )",
            "Inconsistent dimensions."
        );
    }
    return *this;
}

```

The initial checks are done to assure that the operation is valid and that a minimum amount of copying will be done. If the data models are identical, only the pointer is copied, after the original *Matrix* is disconnected from its old data. On the other hand, if the data models are different, then the matrix elements from one must be copied into the other. This is done using a virtual function, **.loadFrom**, which will be explained more fully in Section 4.1. The macro **PREPFORCHANGE** is shown below.

```

#define PREPFORCHANGE( _pml ) if( (_pml)->_rc > 1 ) {      \
                                --((_pml)->_rc);          \
                                (_pml) = (_pml)->Clone(); \
                            }

```

Defined as a macro for convenience, it comes into play *whenever the matrix elements of a Matrix are about to be altered*. It uses the pure virtual method **.Clone**, also described in Section 4.1, to reproduce

the data exactly, including the correct data model, but *only if necessary* – that is, only if more than one `Matrix` refer to the data.

2.3 Accessors

Users of a `Matrix` class reasonably expect to write statements that access individual matrix elements: statements like

```
x(i,j) = 3.1415*x(i,j);
z = x(0,0)*cos( x(2,3) + y(k,1) );
if( y(2,2) > y(1,1) ) y(1,1) = y(2,2);
```

In `LinearAlgebra`, values are written to or read from matrix elements not directly but with the help of two “accessor” classes: `MED`, for `MatrixD`, and `MEC`, for `MatrixC`. They are needed, among other reasons, because a `Matrix` does not know how matrix elements are stored by the data models. However, even if this were not a consideration, a fragment like

```
y = x;
x(2,2) += 7.5;
```

would make accessors necessary within the context of lazy evaluation, and one like

```
Matrix x( 5, 5 );
x.declareAntiSymmetric();
x( 3, 2 ) = 1.0;
```

would make them necessary, because `x(3, 2)` is not even stored.

The principal characteristics of a `Matrix` accessor are:

- (a) its data are [1] the address of the `Matrix` which invoked the accessor function, `.operator()(int i, int j)`, and [2] the arguments, `i` and `j`;
- (b) it is the object returned by the accessor function;
- (c) it possesses a conversion to a scalar, `Float8` for `MED` and `Complex8` for `MEC`; and
- (d) operators that would change a matrix element — such as `=`, `+=`, or `++` — are overloaded within accessor class.

In principle, property (c) enables the accessor to be used in any expression where one would use the corresponding scalar. In practice, this does not always work. For example, consider the following possibility.

```
Complex8 w, z;           // two complex variables
MatrixC  x( 7, 4 );     // a matrix of complex numbers
...
z = w + x( 5, 2 );      // Line 1: This works under all compilers.
z = x( 5, 2 ) + w;     // Line 2: This may or may not work, depending
                       //           on the interface for complex.
```

If the complex header file defines the addition operator as a global function,

```

class complex
{
    public:
    ...
    friend complex& operator+( const complex&, const complex& );
    ...
}

```

then Line 2 will work as well as Line 1. On the other hand, if it is a member function of the class,

```

class complex
{
    public:
    ...
    complex& operator+( const complex& );
    ...
}

```

then the compiler will issue an error message for Line 2 and refuse to compile it. Even though a conversion to complex exists, member functions are not available to the temporary complex variable created. Another example of the problems that arise is shown below.

```

Float8    w;           // a double
Complex8  z;           // a complex
MatrixD   x( 7, 4 );  // a matrix of doubles
...
z = w;           // Line 1: This works under all compilers.
z = x( 5, 2 );  // Line 2: This will not work under any compiler.

```

Here, the accessor returned by “x(5, 2)” possesses a conversion to a Float8 (i.e., double), but the declaration of the assignment operator requires a complex argument, regardless of whether it is done as a member function or as a global friend. Line 1 works, because compilers recognize that a double can be converted to a complex; Line 2 does not, because no compiler recognizes that the MED accessor can be converted to a complex in two steps, by going through the intermediate step of converting to a double. Compilers do not search for possible sequences of conversions; either it can be done in one step, or they throw an error.

There is no clean way around these problems. One can try to “overload everything,” but the number of possibilities is discouragingly staggering. Even if one could cover them all, suppose that in the future another class were introduced that accepted conversion from a double? It is impossible to anticipate everything that could occur. In any case, as a last resort, the user can always force a conversion.

```

Complex8  z;
MatrixD   x( 7, 4 );
...
z = Float8( x( 5, 2 ) );
z = Complex8( Float8( x( 5, 2 ) ) );

```

These lines may look ugly, but they do compile.

2.4 Double dispatching

Because of the variety of data models, a rather involved sequence of events takes place in executing an arithmetic statement, such as “ $z = x + y$ ”, with matrices. In order to take full advantage of whatever efficiencies might exist in the data models, both operands must be treated as polymorphic variables. Unfortunately, the C++ *virtual* mechanism is designed to apply only to one of these variables, the one on the left hand side of the operator. “Double dispatching” is the name given to a set of techniques for making both of them behave “virtually.” I have chosen the construction of “doubly virtual function tables” (DVFT) from among the possible strategies, with the hope that the process of adding new data models will scale at worst linearly with the number of already existing models. Our DVFT consists of six doubly indexed arrays, three for each genus: `MLD::SumVtbl` and `MLC::SumVtbl` to implement matrix addition, `MLD::DiffVtbl` and `MLC::DiffVtbl` to implement subtraction, and `MLD::ProdVtbl` and `MLC::ProdVtbl` for multiplication. These are static data members of the base classes `MLD` and `MLC`, from which all data models are derived. Array indices identify the particular models on either side of the operator, and the value associated with a DVFT element is the name of a function which will carry out the correct operation on those models.

To see how this works, we will trace through the actions induced by the line

```
z = x + y;
```

The source code for the addition operator is shown below.

```
MatrixD operator+ ( const MatrixD& x, const MatrixD& y )
{
    if( ( x._pml->r == y._pml->r ) &&
        ( x._pml->c == y._pml->c ) )
    {
        return MatrixD( MLD::SumVtbl[ x._pml->vtblIndex() ]
                       [ y._pml->vtblIndex() ]
                       ( x._pml, y._pml )
                       );
    }
    else
    {
        MatrixD::error( "operator+( MatrixD, MatrixD )",
                       "Inconsistent dimensions." );
        return x; // Wrong, but the compiler requires something.
    }
}
```

Before doing anything substantial, the arguments’ dimensions are tested for compatibility. That this is done now by the `Matrix` class, rather than later by the `MLD` class, is a design decision to perform all such tests *at the highest level possible*. Tests specific to particular models will have to be done at the `MLD` level, but this one is common to all types of matrices.

Entry into the DVFT is provided by the virtual function `.vtblIndex`, which provides an integer index associated with the data model. The value returned by the array is a function which takes two `const MLD*` arguments – to which we bind the variables `x._pml` and `y._pml` – and returns an `MLD*` pointer to a new data object. For example, if both arguments refer to *Diagonal* matrices, we might use a function like this:

```

MLD* SumDiagDiag( const MLD* argx, const MLD* argy )
{
    Float8 *px, *py, *pz;
    MLDDiag* pNew = new MLDDiag( argx->_r );

    px = argx->_pdi;
    py = argy->_pdi;
    pz = pNew->_pdi;
    while( pz <= pNew->_pdf ) *(pz++) = *(px++) + *(py++);

    return pNew;
}

```

The name of the function is arbitrary but is descriptive of the operation performed and the data models employed. Since the sum of two diagonal matrices is diagonal, a pointer to an `MLDDiag` is returned. In general, the type of pointer returned depends on the arguments: adding a *Generic* matrix to a *Diagonal* matrix should result in a *Generic* matrix; the product of two *Symmetric* matrices need not be *Symmetric*, but their sum or difference would be; because they belong to a (multiplicative) group, the product of two *SU2* matrices would be another *SU2* matrix, but their sum would be *Generic*.¹ The method for doing the addition itself can be used whenever two identical data models are added or subtracted. The `MLD*` pointer that is returned is used as the argument to a private `MatrixD` constructor, which finishes the task of providing the final answer.

Note: Work was begun on the LinearAlgebra module long before the FPCL exception handling procedures were developed. Invoking `MatrixD::error` was (and is) meant as a stopgap measure to be used until they were ready. Now that they are ready, these statements should be replaced.

It is not necessary that every data model know about all other data models. Their indexing gives them an “order,” and it is only necessary that that each model load the table entries for itself and all matrices at a lower order that act with it. Thus each higher order data model has only the responsibility of providing functions which do arithmetic involving itself with matrices of lower order. In this way, as newer data models are added, source code for the lower order ones need not be changed.

Arithmetic operations involving a matrix and a scalar are handled by the ordinary mechanism of virtual functions. For example, the source code for `MatrixD` operator+ (`const MatrixD&, const Float8&`) is given below.

```

inline MatrixD operator+ ( const MatrixD& x, const Float8& y )
{
    return MatrixD( x._pml->add( y ) );
}

```

START HERE, LEO!!! There is no arithmetic function `Sum_M22_M33`. However, there is a “hidden” data model, called `Base`, whose arithmetic functions are loaded by the base class constructor, `MLD::MLD`. Whenever any matrix is instantiated, this constructor first initializes all rows and columns of the doubly virtual function table with its own functions. Some of these are later replaced, as indicated earlier, but the ones that are not replaced remain in the table as default actions. For the specific case asked, this function will catch the fact that the row and column dimensions are not correct and throw an exception. Suppose,

¹We will discuss the tasks of writing these functions and assigning them to the DVFT in Section 4.4.

however, that one tried to add a 3×3 to a *Diagonal* matrix of the correct dimensionality and that the author of the *Diagonal* data model source code had forgotten (or perversely refused) to write and load the functions `Sum_M33_Diag` and `Sum_Diag_M33`. In such a case, the base class function is invoked and will produce the correct answer except that, rather than getting an *M33* result as expected, the matrix returned will be *Generic*. That will be the price of neglecting to do things right.

3 Adding new data models

We now describe in more detail the functions which must be included in implementations for new data models. For purposes of illustration, we will use source code from the file `MLDDiag.cc`, which implements a *Diagonal* data model, and a few fragments from others, such as *Symmetric* or *AntiHermitian*. A few lines also must be added to already existing files to provide an interface with the `Matrix` classes. After understanding this section the developer should have all the information necessary to add data models to the `LinearAlgebra` package.

3.1 Assign an index.

To begin with, each data model must possess a unique index for entry into the DVFT (doubly virtual function tables). This is done in the header file `Models.h`; it would be a good idea to view this file now for reference. For each `MLDXXX` (or `MLCXXX`) data model you must define a macro named `FPCL_INDEX_D_XXX` (or `FPCL_INDEX_C_XXX`), as an integer value, the index for the data model. It should be in sequence with preceding data models. The total number of data models is assigned to macros named `FPCL_MLD_TYPES` and `FPCL_MLC_TYPES`; these may be slightly larger than the actual number of data models, but they must not be smaller. Thus, the first thing that you must do is make certain that these numbers are large enough to accommodate the creation of your new data model. If not, then simply increase their value, and proceed to define `FPCL_INDEX_D_XXX` and/or `FPCL_INDEX_C_XXX`.

3.2 Declare .declare

Lines must be added to header file `Matrix.h` in order to declare the new `.declareXXX` and `.makeXXX` member functions. This is done in two places: as part of the definitions of `class MatrixD` and of `class MatrixC`. Even though these functions are members of the `Matrix` classes, they will be implemented in your *new* source files, *not* in the `MatrixD.cc` and `MatrixC.cc` files.

3.3 Write the new `MLDXXX` and/or `MLCXXX` header files

Your new `MLDXXX` (`MLCXXX`) object will be a *struct* which inherits from `MLD` (`MLC`). At the minimum, it must possess (a) constructors to enable all possible ways of declaring a `Matrix`, (b) a destructor, (c) a pointer to the data, and (d) declaration of the pure virtual functions from the base class. That last requirement can be satisfied by including the header file `MLDVirtF.h` (`MLCVirtF.h`) within the body of *struct*'s definition. Notice, by the way, that all the reference counting is handled by the base *struct*, so we need not be concerned about it here.

In addition to these minimal requirements, impure virtual functions from the base class can also be overloaded if there is reason to do so. For example, in the diagonal models we have overloaded the `.de-`

terminant, **.inverse**, and **.trace** functions because these can be carried out more efficiently within the model.

3.4 Write the model's source file

Finally, we are ready to write the source code implementing the new data model. Most, but not all, of the minimum requirements have already been declared in the header file. Because this step is more involved than the others, we will postpone working through the example until the next section.

3.5 Update the factories

The new models must be incorporated into the “factory” files, `FactoryD.cc` and/or `FactoryC.cc`. These files implement the functions

```
istream& operator>>( istream& is, MatrixC& x )
istream& operator>>( istream& is, MatrixD& x )
```

providing the procedures that read matrices from a stream. They are the only files – the only compilable units – which need to know about *all* the data models appropriate to a genus.

Just a few lines need to be added to the factory files. First, the new `MLDXXX.h` (`MLCXXX.h`) header file must be included with the others near the top. Once that is done, lines of the form

```
else if( strstr( m_type, "MLDXXX" ) != ((void*) 0) ) {
    x._pml = new MLDXXX( ... );
```

is added to the gauntlet of `else if` clauses.

4 Implement new data models

Let us now complete the piece that we omitted and consider the functions that must be included in the new implementation files, `MLDXXX.cc` and `MLCXXX.cc`. As mentioned above, we will primarily use `MLDDiag.cc` for illustration.

4.1 Utility functions

To begin with, there are a few crucial utility functions, meant to be used by other functions. Their signatures are tabulated below, for both `MLDXXX` and `MLCXXX`.

| | | | | | |
|--------|---|-------|----------|--|----|
| int | MLDXXX::vtblIndex() | const | int | MLCXXX::vtblIndex() | co |
| Float8 | MLDXXX::value(int, int) | const | Complex8 | MLCXXX::value(int, int) | co |
| bool | MLDXXX::setValue(int, int, const Float8&) | | bool | MLCXXX::setValueFlt(int, int, const Float8&) | |
| | | | bool | MLCXXX::setValueCmp(int, int, const Complex8&) | |
| MLD* | MLDXXX::Clone() | const | MLC* | MLCXXX::Clone() | co |
| MLD* | MLDXXX::EmptyClone() | const | MLC* | MLCXXX::EmptyClone() | co |
| MLC* | MLDXXX::cloneComplex() | const | | | |
| void | MLDXXX::loadFrom(const MLD*) | | void | MLCXXX::loadFrom(const MLC*) | |
| void | MLDXXX::switchRows(int, int) | | void | MLCXXX::switchRows(int, int) | |
| void | MLDXXX::switchColumns(int, int) | | void | MLCXXX::switchColumns(int, int) | |

The **.vtblIndex** function returns the index that is the model's entry into the DVFT. This is implemented best as an inline function,

```
inline int MLDDiag::vtblIndex() const
{
    return FPCL_INDEX_D_DIAG;
}
```

in the data model's **.icc** file. Of course, the macro `FPCL_INDEX_D_DIAG` (or whatever) was previously defined in the header file `Models.h` (see above).

The function **.value** must return the value of the matrix element associated with indices provided by its arguments. It does this even when the indices correspond to data that are not actually stored. Consider the `MLDDiag` example below.

```
Float8 MLDDiag::value( int i, int j ) const
{
    if( i == j ) return _m[i];
    else return 0.0;
}
```

In class `MLDDiag` only the diagonal elements of a matrix are stored, in a private array named `_m`. As always, checking that the arguments `i` and `j` are within the correct range was already done at the `Matrix` level, so it need not be repeated at the `MLD` level; only model-specific tests are carried out here. In this case, if the two arguments are the same, the value of the correct matrix element is returned; if not, a zero is returned. Other possibilities may be more complicated.

```
Complex8 MLCAntiHermitian::value( int i, int j ) const
{
    // Note: _m[i][j] was not allocated for i >= j.
    if( i == j ) return Complex8( 0.0, 0.0 );
    else if( i < j ) return _m[i][j];
    else return - conj( _m[j][i] );
}
```

The **.setValue** methods perform the inverse process: storing values within whatever structure holds the matrix's data.

```

bool MLDDiag::setValue( int i, int j, const Float8& x )
{
    // Normally, the range check should be done
    // at a higher level.
    #if defined(ZM_MLD_DEBUG) || defined(ZM_MLD_TESTS)
    if( ( i < 0 ) || ( i > _r ) ||
        ( j < 0 ) || ( j > _c )
        ) {
        MLD::error( "MLDDiag::setValue",
                    "Indices out of bounds." );
        return 1;
    }
    #endif

    if( i != j ){
        MLD::error( "MLDDiag::setValue",
                    "Cannot change off-diagonal element." );
        return 1;
    }

    _m[i] = x;
    return 0;
}

```

Here, because data are going to be stored, we have allowed for an extra level of paranoia. By defining one of two macros, the user can activate an extra level of range checking for debugging programs. Unlike `MLDDiag::value`, which only needs to return a value, the model-specific test throws an error if the condition “`i = j!`” is satisfied. Otherwise, the datum provided by the third argument is stored in the appropriate location.

For the sake of a second illustration, consider what should happen in the *anti-Hermitian* data model.

```

bool MLCAntiHermitian::setValueCmp( int i, int j, const Complex8& x )
{
    if( i == j ) {
        MLC::error( "MLCAntiHermitian::setValueCmp",
                    "Cannot change diagonal element." );
        return 1;
    }
    else if( i < j ) _m[i][j] = x;
    else _m[j][i] = conj( x );
    return 0;
}

```

Notice that data models the complex genus, such as (the fictitious) `MLCAntiHermitian`, require two methods, `.setValueCmp` and `.setValueFlt`, depending on whether the third argument refers to a double or a complex variable. Unfortunately, the argument list cannot determine their full signature because these functions are declared to be virtual in the base class.

As their names suggest, functions **.Clone**, **.EmptyClone**, and **.cloneComplex**, put copies of the invoking objects on the heap. If constructors have been written correctly, these methods can be implemented easily using `memcpy`.

```
MLD* MLDDiag::Clone() const
{
    static MLD* q;
    q = new MLDDiag( _r );
    memcpy( q->_pdi, _pdi, _size );
    return q;
}
```

.EmptyClone is like this except that data are not transferred.

```
inline MLD* MLDDiag::EmptyClone() const
{
    return new MLDDiag( _r );
}
%
```

Finally, genus MLD models possess a third function, **.cloneComplex**, which returns a pointer to the complex version of itself.

```
MLC* MLDDiag::cloneComplex() const
{
    static MLC*      q;
    static Complex8* pf;
    static Float8*   pi;
    q = new MLCdiag( _r );
    for( pi = _pdi, pf = q->_pdi;
        pi <= _pdf;
        pf++, pi++ )
    {
        *pf = *pi;
    }
    return q;
}
```

The **.loadFrom** methods copy data from one MLD to another regardless of the original model. Consider, for example, an implementation of `MLDDiag::loadFrom`.

```
void MLDDiag::loadFrom( const MLD* x )
{
    static int i;

    if( ( _r == x->_r ) && ( _c == x->_c ) ) {
        for( i = 0; i < _r; i++ ) {
            _m[i] = x->value( i, i );
        }
    }
}
```

```

    }
  }
  // Dimensions were not correct.
  else {
    MLD::error( "MLDDiag::loadFrom( MLD* )",
               "Dimensions do not match."
               );
  }
}

```

(Here I have violated the principle of carrying out the range tests at a higher level; I don't remember why. Perhaps this is a mistake, but if so, it can't hurt.) Notice that the method stores the data regardless of whether or not the argument *x* points to a diagonal matrix. Similarly, the method

```

void MLDSymmetric::loadFrom( const MLD* x )
{
  static int i;

  if( ( _r == x->_r ) && ( _c == x->_c ) ) {
    for( i = 0; i < _r; i++ ) {
      for( j = i; j < _c; j++ ) {
        _m[i][j] = ( x->value( i, j ) + x->value( j, i ) ) / 2.0;
      }
    }
  }
  else {
    MLD::error( "MLDSymmetric::loadFrom( MLD* )",
               "Dimensions do not match."
               );
  }
}

```

would convert the data in *x* into that of a *Symmetric* matrix.

Lastly, the **.switchRows** and **.switchColumns** member functions perform the operations indicated by their names. Apart from the **.setValue** methods, they are the only ones in this set that alter data attached to the object. The functions manipulate the data, regardless of how it is encrypted into the model, so that the result is what one would get by switching the rows or columns of a matrix. If the data model is itself invalidated by this operation – as, indeed, most will be – an error should be thrown.

```

void MLDDiag::switchRows( int, int )
{
  MLD::error( "MLDDiag::switchRows",
             "You cannot do this, you moron." );
}

```

4.2 .makeXXX and .declareXXX

The first order of business is to provide the Matrix member functions, **.makeXXX** and **.declareXXX**, for transforming a generic matrix into the new model. This is purposely put here, rather than in the file `MatrixD.cc` (or `MatrixC.cc`), to prevent the necessity of linking to all possible data models when using matrices in application programs. The simpler of the two is **.makeXXX**, which forces the operation (almost) regardless of the matrix's data. As an example, consider the function `MatrixD::makeDiagonal()`.

```
MatrixD& MatrixD::makeDiagonal()
{
    static MLDDiag* pNewMLD;
    static int      i;

    if( _pml->_r != _pml->_c ) {
        // This call to an error function should
        // be replaced with a legitimate
        // exception handler.
        MLD::error( "MatrixD::makeDiagonal",
                   "Matrix is not square." );
    }

    if( typeid( *_pml ) != typeid( MLDDiag ) ) {
        pNewMLD = new MLDDiag( _pml->_r, 0.0 );

        for( i = 0; i < _pml->_r; i++ ) {
            pNewMLD->_m[i] = _pml->value(i,i);
        }

        if( --( _pml->_rc ) == 0 ) delete _pml;
        _pml = pNewMLD;
    }

    return *this;
}
```

A minimal check is done to make certain that the original matrix is indeed square. If it is, and if the data model is not already `MLDDiag`, a new `MLDDiag` model is constructed having the appropriate number of rows and zero along the diagonal. Of course, the arguments of the constructor are at the discretion of the model's designer. Data from the old model are then loaded into the new. The virtual member function **.value** is assured to return the correct number regardless of the old data model used. `MLDDiag` stores *only* the diagonal elements in an array named `_m`.

The all-important final step in this process is to connect the Matrix object with its new data. First, the old data must be disconnected. The reference counter is decreased and, if it reaches zero – which means that no other matrix possesses these data – then the old data *must be deleted*. Failure to do this will result in a memory leak. Finally, the Matrix's data pointer is set to the new model's address, and a reference to the Matrix itself is returned.

Notice that a `.makeXXX` function forces the transformation. In this case, the matrix need not have been diagonal originally, and after the function is invoked, it has no way of recovering the off-diagonal elements it once possessed. On the other hand, the corresponding function `.declareXXX` will not operate unless the original matrix actually has the properties of the new data model. Thus, `MatrixD::declareDiagonal()` will not perform its function unless all off-diagonal elements of the matrix invoking it vanish. All the checks that assure that a matrix's data actually conform to the new model must be made within the corresponding `MatrixD::declareXXX` program.

```
MatrixD& MatrixD::declareDiagonal()
{
    static int i, j, r;

    if( _pml->_r != _pml->_c ) {
        MLD::error( "MatrixD::declareDiagonal",
                   "Matrix is not square." );
    }

    if( typeid( *_pml ) == typeid( MLDDiag ) ) {
        return *this;
    }

    r = _pml->_r;
    for( i = 0; i < r; i++ ) {
        for( j = 0; j < r; j++ ) {
            if( i == j ) continue;
            if( _pml->value( i, j ) != 0.0 ) {
                error( "MatrixD::declareDiagonal",
                       "Matrix is not diagonal."
                );
            }
        }
    }

    return makeDiagonal();
}
```

Since the final line is an invocation of `MatrixD::makeDiagonal()`, the initial checks could actually have been omitted here, as they are done in the other function anyway. What could not have been omitted is the third test, that all off-diagonal elements vanish. Again, to avoid knowing about all possible kinds of models, we use the virtual member function `.value` to access these data. If the test is passed, `MatrixD::makeDiagonal()` is invoked to complete the transformation. (Of course, it need not have been written this way; the transformation could have been incorporated explicitly into this function.)

4.3 Constructors

There is a certain amount of freedom in designing a model's constructor interface. In the example above, we required an `MLDDiag` constructor with two arguments: the number of rows and the initial value along

the diagonal. Here is the implementation of that constructor.

```
MLDDiag::MLDDiag( int r, const Float8& initval )
: MLD( r, r )
{
    static int opsDone = initOps();
    static int i;

    _m = new Float8 [r];
    for( i = 0; i < r; i++ ) {
        _m[i] = initval;
    }

    _pdi = _m;
    _pdf = &( _m[r-1] );
    _size = r*sizeof( Float8 );
}
```

Of course, this constructor invokes an appropriate base class constructor before entering its body. Its first obligation is to register the existence of its model with the virtual function table. This is done with the virtual member function `MLDDiag::initOps()`, which will be described in detail below. `MLDDiag::initOps()` returns an `int` value which is stored in a static variable, `opsDone`. This assures that `MLDDiag::initOps()` is invoked *once and only once*, i.e., the first time that this constructor is itself invoked.²

After registering with the appropriate DVFT, memory is allocated for data storage. In this case, only the diagonal elements of a matrix are being stored, so a new array is created to hold these numbers. (Of course, its name has already been specified in the header file `MLDDiag.h`.) Further, the constructor allows initialization with a value passed as a parameter, so that is done as well.

The final three lines in this constructor fulfill its responsibility to its base class. Every data model constructor must assign values to the base class member variables: (a) `_pdi`, the starting address of the (matrix element) data, (b) `_pdf`, the address of the last item of (matrix element) data, and (c) `_size`, the number of bytes of (matrix element) data stored. The example shown above indicates how this should be done. Failure to do it correctly will result in run-time errors for which no error messages or exception handling are provided. The base class simply assumes that this task *is* done correctly; in any case, it has no way of checking.

4.4 Registering with the DVFT

A new data model must be registered with the doubly virtual function tables (DVFT). This is done with a method `int MLDXXX::initOps()` which returns a totally arbitrary integer value. (For the reason, see Section ???.) “Registering” means that this method must load into the DVFT the names of functions which determine how arithmetic is to be carried out within the model. These functions could be either static member functions of the `MLDXXX` class or global functions; clearly, the former option is better. Each function should have a name suggestive of the operations to be carried out, each takes two `const MLD*` (or `const MLC*`) arguments, and each returns a new `MLD*` (or `MLC*`). For example, consider the following, which specifies how two diagonal matrices are to be multiplied together.

²In reality, it is called once for each `MLDDiag` constructor that exists. Since there are only a few of them, this is not a serious problem.

```

MLD* MLDDiag::ProdDiagDiag( const MLD* argx, const MLD* argy )
{
    Float8 *px, *py, *pz;
    MLDDiag* ret = new MLDDiag( argx->_r );

    pz = ret->_pdi;
    px = argx->_pdi;
    py = argy->_pdi;
    while( pz <= ret->_pdf ) *(pz++) = (*(px++)) * (*(py++));

    return ret;
}

```

This is a particularly simple example in which it is not even necessary to recast the arguments. Let us try something slightly more involved, such as adding a diagonal matrix to an antisymmetric matrix.

```

MLD* MLDAntiSymmetric::SumDiagAsym( const MLD* argx, const MLD* argy )
{
    static    MLDDiag*  x;
    static    int       i, r;

    r = argy->_r;
    MLDGeneric* ret    = new MLDGeneric( r, r );

    #ifndef NO_DYNAMIC_CAST
    x = dynamic_cast<const MLDDiag*>( argx );
    #endif
    #ifdef NO_DYNAMIC_CAST
    x = (const MLDDiag*)( argx );
    #endif

    ret->loadFrom( argy );
    for( i = 0; i < r; i++ ) ret->_m[i][i] = x->_m[i];
    return ret;
}

```

The `MLDDiag` data model precedes `MLDAntiSymmetric`, so any arithmetic methods involving `MLDAntiSymmetric` are written within its implementation. This is the general rule, which avoids having to modify old source files: “earlier” data models possess no information about “later” data models. In this example, we must return an `MLDGeneric*`, a pointer to data for a *Generic* data model. There is no way of knowing whether a more specialized data model would be correct. We recast `argx` as an `MLDDiag*` so that we can later access its data array `_m` by name. (The program did not have to be written in this manner; this is done by way of example.) The computation is accomplished by first loading the data from the antisymmetric matrix into `ret`. This leaves the task of adding data from the diagonal matrix, but that is simplified, since an antisymmetric matrix has zeroes along the diagonal. Thus, the diagonal elements of the answer are the same as those of the diagonal matrix summand.

A similar function, `MLDAntiSymmetric::SumAsymDiag`, would have to be written to handle the arguments in reverse order. In this case, since addition is commutative, and if an extra function call here or there is not considered important, it can be implemented merely by calling the other.

```
MLD* MLDAntiSymmetric::SumAsymDiag( const MLD* argx, const MLD* argy )
{
    return MLDAntiSymmetric::SumDiagAsym( argy, argx );
}
```

Of course, this would not be the case with matrix multiplication functions, such as `MLDAntiSymmetric::ProdDiagAsym` and `MLDAntiSymmetric::ProdAsymDiag`.

We consider one more example.

```
MLD* MLDDiag::ProdM22Diag( const MLD* argx, const MLD* argy )
{
    static MLDDiag*   x;
    static int        i, j;
    static Float8     multiplier;

    MLD22* ret = new MLD22;
    ret.loadFrom( argx );

    #ifndef NO_DYNAMIC_CAST
    y = dynamic_cast<const MLDDiag*>( argy );
    #endif
    #ifdef NO_DYNAMIC_CAST
    y = (const MLDDiag*)( argy );
    #endif

    for( j = 0; j < 2; j++ ) {
        multiplier = y->_m[j];
        for( i = 0; i < 2; i++ )
        {
            ret->_m[i][j] *= multiplier;
        }
    }
    return ret;
}
```

Here, it makes sense to return a 2×2 data model as the product of a 2×2 with a *Diagonal* matrix. In each case, the developer must discern which model should be returned.

Now, please note: even if `MLDXXX::initOps()` *does nothing* except return a value, *arithmetic will still take place*. This is because the base class, `MLD`, possesses its own `MLD::initOps()` method that loads a default set of arithmetic functions into the DVFT. The code below shows one of these.

```
MLD* ProdBaseBase( const MLD* x, const MLD* y )
{
    static int    i, j, k, r, c;
```

```

static Float8 answer;

r = x->_r;
c = y->_c;

MLDGeneric* pgen = new MLDGeneric( r, c );

for( i = 0; i < r; i++ ) for( j = 0; j < c; j++ ) {
    answer = 0.0;
    for( k = 0; k < x->_c; k++ ) {
        answer += x->value( i, k ) * y->value( k, j );
    }
    pgen->_m[i][j] = answer;
}

return pgen;
}

```

The default functions may not work as efficiently as specialized arithmetic functions, and, by default, what they return is always a *Generic* data model, but by not replacing them in the DVFT the code developer effectively states that they are good enough.

On the other hand, if they are *not* good enough, then the new arithmetic functions must be loaded into the DVFT arrays, *SumVtbl*, *DiffVtbl*, and *ProdVtbl*, which respectively handle addition, subtraction, and multiplication of matrices. As has been mentioned several times already, this is the responsibility of the member function `int MLDXXX::initOps()` (or `int MLCXXX::initOps()`), which assigns the names of the arithmetic functions to the correct elements in the DVFT arrays. For example:

```

int MLDDiag::initOps()
{
    for( int i = FPCL_INDEX_D_BASE;
        i < FPCL_MLD_TYPES;
        i++ ) {
        SumVtbl [ FPCL_INDEX_D_DIAG ][ i ] = SumDiagBase;
        SumVtbl [ i ][ FPCL_INDEX_D_DIAG ] = SumBaseDiag;
        DiffVtbl [ FPCL_INDEX_D_DIAG ][ i ] = DiffDiagBase;
        DiffVtbl [ i ][ FPCL_INDEX_D_DIAG ] = DiffBaseDiag;
        ProdVtbl [ FPCL_INDEX_D_DIAG ][ i ] = ProdDiagBase;
        ProdVtbl [ i ][ FPCL_INDEX_D_DIAG ] = ProdBaseDiag;
    };

    SumVtbl [ FPCL_INDEX_D_DIAG ][ FPCL_INDEX_D_DIAG ] = SumDiagDiag;
    DiffVtbl [ FPCL_INDEX_D_DIAG ][ FPCL_INDEX_D_DIAG ] = DiffDiagDiag;
    ProdVtbl [ FPCL_INDEX_D_DIAG ][ FPCL_INDEX_D_DIAG ] = ProdDiagDiag;

    SumVtbl [ FPCL_INDEX_D_GEN ][ FPCL_INDEX_D_DIAG ] = SumGenDiag;
    SumVtbl [ FPCL_INDEX_D_DIAG ][ FPCL_INDEX_D_GEN ] = SumDiagGen;
    DiffVtbl [ FPCL_INDEX_D_GEN ][ FPCL_INDEX_D_DIAG ] = DiffGenDiag;
}

```

```

DiffVtbl[ FPCL_INDEX_D_DIAG ][ FPCL_INDEX_D_GEN ] = DiffDiagGen;
ProdVtbl[ FPCL_INDEX_D_GEN ][ FPCL_INDEX_D_DIAG ] = ProdGenDiag;
ProdVtbl[ FPCL_INDEX_D_DIAG ][ FPCL_INDEX_D_GEN ] = ProdDiagGen;

SumVtbl [ FPCL_INDEX_D_M22 ][ FPCL_INDEX_D_DIAG ] = SumM22Diag;
SumVtbl [ FPCL_INDEX_D_DIAG ][ FPCL_INDEX_D_M22 ] = SumDiagM22;
DiffVtbl[ FPCL_INDEX_D_M22 ][ FPCL_INDEX_D_DIAG ] = DiffM22Diag;
DiffVtbl[ FPCL_INDEX_D_DIAG ][ FPCL_INDEX_D_M22 ] = DiffDiagM22;
ProdVtbl[ FPCL_INDEX_D_M22 ][ FPCL_INDEX_D_DIAG ] = ProdM22Diag;
ProdVtbl[ FPCL_INDEX_D_DIAG ][ FPCL_INDEX_D_M22 ] = ProdDiagM22;

< ... etcetera, etcetera, etcetera ... >

return 1;
}

```

Developers of data models can provide as many or as few of these as they consider useful.

4.5 Additional arithmetic operations

In addition to the DVFT, which come into play when binary arithmetic operators are sandwiched between two matrices, the virtual methods presented in the table below are invoked by other kinds of arithmetic expressions. The first column of the table contains samples of application-level expressions which would activate the corresponding methods; here, M and N represent Matrix objects and x a scalar variable (i.e., double or complex).

| | | | | |
|-----------|------|---------------------|-----------------------|-------|
| M *= N | void | MLDXXX::opMultEqMLD | (const MLD*) | |
| | void | MLCXXX::opMultEqMLC | (const MLC*) | |
| M += x | void | MLDXXX::opPlusEqFlt | (const Float8&) | |
| | void | MLCXXX::opPlusEqFlt | (const Float8&) | |
| | void | MLCXXX::opPlusEqCmp | (const Complex8& x) | |
| M -= x | void | MLDXXX::opSubEqFlt | (const Float8&) | |
| | void | MLCXXX::opSubEqFlt | (const Float8&) | |
| | void | MLCXXX::opSubEqCmp | (const Complex8&) | |
| M + x, or | MLD* | MLDXXX::add | (const Float8&) | const |
| x + M | MLC* | MLCXXX::addFlt | (const Float8&) | const |
| | MLC* | MLCXXX::addCmp | (const Complex8&) | const |
| M - x, or | MLD* | MLDXXX::subtract | (const Float8&) | const |
| x - M | MLC* | MLCXXX::subtractFlt | (const Float8&) | const |
| | MLC* | MLCXXX::subtractCmp | (const Complex8&) | const |

By way of illustration, we offer below the source code for `MLCDiag::subtractCmp`.

```

MLC* MLCDiag::subtractCmp( const Complex8& x ) const
{
    static MLCDiag* ret;
    static Complex8* p;

```

```

#ifdef NO_DYNAMIC_CAST
    ret = (MLCDiag*) ( this->Clone() );
#else
    ret = dynamic_cast<MLCDiag*>( this->Clone() );
#endif

    for( p = ret->_pdi; p <= ret->_pdf; p++ ) {
        (*p) -= x;
    }
    return ret;
}

```

As always, tests are made at the highest possible level, so confirmation that the matrix is square was done at the Matrix level, before invoking `MLCDiag::subtractCmp`. This method does not change the data in its object but returns a pointer to a new MLC data struct, `ret`, which is first constructed as a clone of the original. (The `NO_DYNAMIC_CAST` macro allows for compilers which do not understand RTTI. Their number is decreasing with time and, it is hoped, will soon go to zero.) Since only diagonal elements are carried by `MLCDiag`, the argument `x` is simply subtracted from all of them. Although it does not look it, this method is sufficient for both expressions “`M-x`” and “`x-M`,” because the sign change arising from non-commutativity will automatically be handled at a higher level.

As an example of a model containing non-diagonal elements, consider the following.

```

MLD* MLDGeneric::add(const Float8& x) const
{
    static MLDGeneric* ret;
    static int i;

#ifdef NO_DYNAMIC_CAST
    ret = (MLDGeneric*) this->Clone();
#else
    ret = dynamic_cast<MLDGeneric*>( this->Clone() );
#endif

    for( i = 0; i < ret->_r; i++ ) {
        ret->_m[i][i] += x;
    }
    return ret;
}

```

On the other hand, the above methods which return a `void` are meant to alter their objects’ data.

```

void MLDDiag::opPlusEqFlt( const Float8& x )
{
    static Float8* p;
    for( p = _pdi; p <= _pdf; p++ ) {
        (*p) += x;
    }
}

```

```

}

void MLDGeneric::opPlusEqFlt( const Float8& x )
{
    static int i;
    for( i = 0; i < _r; i++ ) {
        _m[i][i] += x;
    }
}

```

As usual, if the operation is not appropriate for the data model, an error condition should be thrown.

```

void MLDAntiSymmetric::opPlusEqFlt( const Float8& x )
{
#ifdef NO_DYNAMIC_CAST
    MLDAntiSymmetric* ret = (MLDAntiSymmetric*) this->Clone();
#else
    MLDAntiSymmetric* ret = dynamic_cast<MLDAntiSymmetric*>( this->Clone() );
#endif
    if( x == 0.0 ) {
        return ret;
    }
    else {
        MLD::error( "MLDAntiSymmetric::opPlusEqFlt",
                    "Cannot make diagonal element non-zero." );
    }
}

```

Finally, it is not at all necessary that the returned model be the same type as the original.

```

MLD* MLDAntiSymmetric::add( const Float8& x ) const
{
    static MLDGeneric* ret;
    static int i;

    ret = new MLDGeneric( _r, _c );
    ret->loadFrom( this );

    for( i = 0; i < ret->_r; i++ ) {
        ret->_m[i][i] += x;
    }
    return ret;
}

```

Here, an *AntiSymmetric* Matrix is converted to a *Generic* one by adding something along the diagonal.

4.6 Streaming methods

Two virtual functions provide the “light persistence” mechanism of reading to or writing from streams: `void MLD::writeTo(ostream&)` has a base class implementation, which acts as the default, while `MLDXXX::readFrom(istream&)` is a pure virtual function and, therefore, *must* possess a specific implementation for each data model. (Of course, the MLC genus behaves the same.) It is the responsibility of the **.writeTo** method to write its object’s data to the stream identified in its argument, and of **.readFrom** to read it back from the stream. Of course, the latter must be written in a way that takes into account how the former actually wrote the data. In particular, if the base class `MLD::writeTo` is not overloaded, then the designer must implement `MLDXXX::readFrom` following the pattern set forth in `MLDGeneric::readFrom` or `MLDDiag::readFrom`. That pattern, taken from the class `MLDGeneric`, is shown below.

```
void MLDGeneric::readFrom( istream& is )
{
    static char dummy[20];
    char byte;
    char* byte_ptr;
    static int i, j, k;
    short number;

    is >> dummy;          // Reads either "formatted" or "dump"

    // This should take care of the header interpretation. Now
    // we'll read in the actual data.

    if (strcmp(dummy, "formatted") == 0){
        // Here are the only model specific lines of code....
        // Begin: model specific code.....
        for( i = 0; i < _r; i++ ) for( j = 0; j < _c; j++ ) {
            is >> _m[i][j];
        }
        // End: model specific code.....
    }

    // Reading in a dump.....
    if ( 0 == strncmp(dummy, "dump", 4) )
    {
        // Read in the '\n' character first.....
        is.unsetf( ios::skipws );
        is >> byte;
        is.setf( ios::skipws );

        // Now read the data.....
        byte_ptr = (char*) _pdi;
        for( k = 0;
            k < _size;
```

```

        k++, byte_ptr++ )
    {
        switch(dummy[5]){ // Extract the dump base.
            case '2': is.unsetf( ios::skipws );
                    is.get(byte);
                    // is >> byte;
                    *byte_ptr = byte;
                    break;
            case '8': is >> oct >> number;
                    *byte_ptr = (char)number;
                    break;
            case '1':
                switch(dummy[6]){
                    case '0': is >> dec >> number;
                            *byte_ptr = (char)number;
                            break;
                    case '6': is >> hex >> number;
                            *byte_ptr = (char)number;
                            break;
                }
                break;
        }
    }
    is.setf( ios::skipws | ios::dec );
}

```

Only a few lines, sandwiched between comments that identify them, are specific to the model. Thus, for example, to change this code into that of `MLDDiag::readFrom`, merely substitute

```

for( i = 0; i < _r; i++ ) for( j = 0; j < _c; j++ ) {
    is >> temp;
    if( i == j ) _m[i] = temp;
}

```

in that location. As another example, to implement `MLDSymmetric::readFrom`, we might write something like

```

for( i = 0; i < _r; i++ ) for( j = 0; j < _c; j++ ) {
    is >> temp;
    if( i <= j ) _m[i][j] = temp;
}

```

On the other hand, if the designer *has* overloaded `MLD::writeTo`, then this can be done in any way desired, and `MLDXXX::readFrom` should be compatible with it. For example, we may want to overload so that only the non-zero elements of a diagonal matrix are written to the stream. Then, we could use

```

for( i = 0; i < _r; i++ ) {

```

```

    is >> _m[i];
}

```

or even

```

Float8* p = _pdi;
while( p <= _pdf ) is >> *(p++);

```

in `MLDDiag::readFrom`. The developer may also ignore all the possible formatting options, choosing only one as appropriate for the model. (If true persistence is desired, it had better be the binary dump.)

4.7 Special functions

Finally, one has the option of overloading virtual functions that are not purely virtual. For example, a `.trace` method is already provided in class `MLD`, the base class of all data models.

```

Float8 MLD::trace() const
{
    static Float8 ret;
    static int i;
    ret = 0.0;
    for( i = 0; i < _r; i++ ) ret += this->value( i, i );
    return ret;
}

```

While it works adequately by using the virtual method `.value`, it can be made slightly more efficient via overloading.

```

Float8 MLDDiag::trace() const
{
    static Float8* p;
    static Float8 ret;
    ret = 0.0;
    for( p = _pdi; p <= _pdf; p++ ) {
        ret += *p;
    }
    return ret;
}

```

Or, consider that taking the inverse of a diagonal matrix should be very simple.

```

MLD* MLDDiag::inverse() const
{
    static MLDDiag* ret;
    static int i;
    ret = new MLDDiag( _r );
    Float8* pi;
    Float8* pf;
    for( pi = _pdi, pf = ret->_pdi;

```

```
    pi < _pdf;
    pi++, pf++ )
{
    if( *pi == 0.0 ) {
        MLD::error( "MLDDiag::inverse",
                    "Matrix is singular." );
        *pf = MAXDOUBLE;
    }
    else {
        *pf = 1.0 / *pi;
    }
}
return ret;
}
```

Depending on your energy and motivation, you should explore model-specific ways of overloading the remaining, non-purely virtual methods in order to polish off the implementation of a new data model.

A Member functions of classes MLCDiag and MLDDiag

File MLDDiag.cc:

```

MatrixD& MatrixD::declareDiagonal()
MatrixD& MatrixD::makeDiagonal()

MLD* SumDiagDiag ( const MLD* argx, const MLD* argy )
MLD* DiffDiagDiag( const MLD* argx, const MLD* argy )
MLD* ProdDiagDiag( const MLD* argx, const MLD* argy )
MLD* SumBaseDiag ( const MLD* argx, const MLD* argy )
MLD* SumDiagBase ( const MLD* argx, const MLD* argy )
MLD* DiffBaseDiag( const MLD* argx, const MLD* argy )
MLD* DiffDiagBase( const MLD* argx, const MLD* argy )
MLD* ProdBaseDiag( const MLD* argx, const MLD* argy )
MLD* ProdDiagBase( const MLD* argx, const MLD* argy )

int MLDDiag::initOps()

MLDDiag::MLDDiag( int r, const Float8& initval )
MLDDiag::MLDDiag( int r, const Float8* initval )
MLDDiag::~MLDDiag()

void MLDDiag::loadFrom( const MLD* x )
void MLDDiag::readFrom( istream& is )

int MLDDiag::vtblIndex()

Float8 MLDDiag::value( int i, int j ) const
bool MLDDiag::setValue( int i, int j, const Float8& x )

MLD* MLDDiag::Clone() const
MLD* MLDDiag::EmptyClone() const
MLC* MLDDiag::cloneComplex() const

void MLDDiag::switchRows( int, int )
void MLDDiag::switchColumns( int, int )

Float8 MLDDiag::determinant() const
Float8 MLDDiag::trace() const
MLD* MLDDiag::transpose() const

MLD* MLDDiag::inverse() const
MatrixEigenData MLDDiag::eigen() const

void MLDDiag::opPlusEqFlt ( const Float8& x )

```

File MLCDiag.cc:

```

MatrixC& MatrixC::declareDiagonal()
MatrixC& MatrixC::makeDiagonal()

MLC* SumDiagDiag ( const MLC* argx,
MLC* DiffDiagDiag( const MLC* argx,
MLC* ProdDiagDiag( const MLC* argx,
MLC* SumBaseDiag ( const MLC* argx,
MLC* SumDiagBase ( const MLC* argx,
MLC* DiffBaseDiag( const MLC* argx,
MLC* DiffDiagBase( const MLC* argx,
MLC* ProdBaseDiag( const MLC* argx,
MLC* ProdDiagBase( const MLC* argx,

int MLCDiag::initOps()

MLCDiag::MLCDiag( int r, const Complex8& initval )
MLCDiag::MLCDiag( int r, const Complex8* initval )
MLCDiag::~MLCDiag()

void MLCDiag::loadFrom( const MLC* x )
void MLCDiag::readFrom( istream& is )

int MLCDiag::vtblIndex()

Complex8 MLCDiag::value( int i, int j ) const
bool MLCDiag::setValueFlt( int i, int j, const Float8& x )
bool MLCDiag::setValueCmp( int i, int j, const Complex8& x )

MLC* MLCDiag::Clone() const
MLC* MLCDiag::EmptyClone() const

void MLCDiag::switchRows( int, int )
void MLCDiag::switchColumns( int, int )

Complex8 MLCDiag::determinant() const
Complex8 MLCDiag::trace() const
MLC* MLCDiag::transpose() const
MLC* MLCDiag::dagger() const

MLC* MLCDiag::inverse() const

void MLCDiag::opPlusEqFlt ( const Float8& x )
void MLCDiag::opPlusEqCmp ( const Complex8& x )

```

```
void MLDDiag::opSubEqFlt ( const Float8& x )
void MLDDiag::opMultEqMLD ( const MLD* x )
MLD* MLDDiag::add( const Float8& x ) const
MLD* MLDDiag::subtract( const Float8& x ) const
```

```
void MLCDiag::opSubEqFlt ( const
void MLCDiag::opSubEqCmp ( const
void MLCDiag::opMultEqMLC ( const
MLC* MLCDiag::addFlt( const Float8&
MLC* MLCDiag::addCmp( const Complex
MLC* MLCDiag::subtractFlt( const FL
MLC* MLCDiag::subtractCmp( const Co
```