Note 155: Array Tutorial

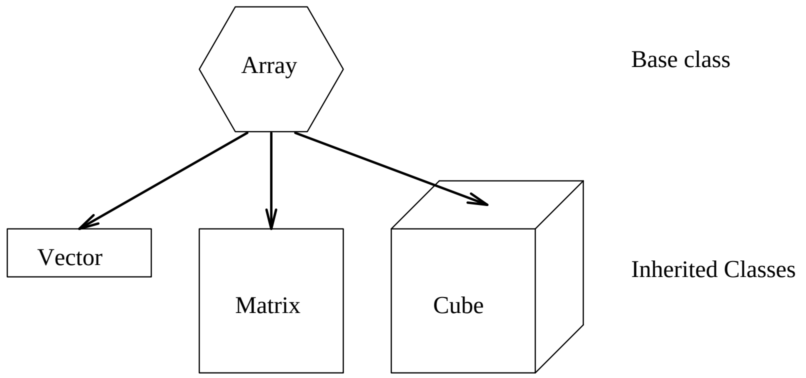
Using the AIPS++ n-dimensional array classes

M. P. Healey (DRAO)
B. E. Glendenning (NRAO)
A. G. Willis (DRAO)

1 Introduction

Many data processing operations in aperture synthesis radio astronomy involve the handling of one, two or three dimensional arrays. An example of a vector, or one dimensional array, would be a spectral line observation (intensity vs frequency at a single position on the sky). A picture of a piece of sky would be stored as a two dimensional array, or matrix. A spectral line data cube (a series of pictures, each made at a different frequency) is an example of a three dimensional array.

For AIPS++ it was decided that rather than develop specific classes to separately handle vectors, matrices and cubes, we would first develop a class that can handle a n-dimensional array, the actual number of dimensions being defined by the application programmer. Since vectors, matrices and cubes are just arrays having specific dimensions we can then define Vector, Matrix and Cube classes which inherit from the generic n-dimensional array class. That way, most of the methods which are generic to arrays, such as the standard arithmetic methods, needed only to be developed once, since derived classes inherit these methods. Methods specific to one of the derived classes, for example indexing into a Matrix, are then implemented in that derived class.



The AIPS++ Array Classes

An additional advantage of this approach is that we can create methods in other classes which define the generic Array class as an input or output parameter, but then use these methods with Vector, Matrix or Cube objects.

The purpose of this document is to provide an introduction to the AIPS++ array classes. We assume a modest understanding of C++ and object-oriented programming terminology. For instance, you should understand the meaning of the terms `constructor`, `overload` and `reference`.

There are four array classes: `Array`, an n-dimensional array class; `Vector`, a 1-dimensional array; `Matrix`, a two-dimensional array; and `Cube`, a three dimensional array.

Since classes `Vector`, `Matrix` and `Cube` are all arrays, they inherit all properties of the class `Array`, as well as adding specializations of their own. Most readers will be familiar with constructs similar to `Vector`, `Matrix`, and `Cube` from other languages; these will therefore be introduced first, followed by the more general class `Array`.

2 Classes Vector, Matrix, and Cube

These are classes which inherit properties from the class `Array`. As you would expect, they are one, two and three dimensional arrays, respectively. In order to use these classes, you must include the appropriate AIPS++ header files, `#include <aips/Vector.h>` for `Vectors`, `#include <aips/Matrix.h>` for `Matrixes`, and `#include <aips/Cube.h>` for cubes.

2.1 Declarations

Here are some example declarations, with explanations:

```
Vector<float> v;
Matrix<Complex> m;
Cube<Int> c;            //Int == int
```

The vector `v` is a zero-length vector of floating point numbers. The matrix `m` is a zero-element matrix of complex numbers. Cube `c` is a zero-element cube of integers.

```
uInt            length = 5;   //uInt == unsigned int
Int             origin = 3;
Vector<Int>     v(length, origin);
Vector<Int>     w(length);

uInt            width  = 5;
uInt            height = 6;
Int             x_origin = y_origin = 1;
Matrix<float>   m(width, height, x_origin, y_origin);
```

The vector `v` is a vector with 5 undefined integer elements, and has its origin at three. Vector `w` is the same, but by default its origin is zero. The Matrix `m` is a 5x6 matrix with origin (1,1). The same declarations extend logically to 3-dimensions for Cubes.

In all cases, if the origin is omitted, the array is zero-based.

2.2 Indexing

Indexing is achieved by use of `operator()`. For example:

```
Int i = v(5);  //if "v" is a defined Vector<Int> with an element 5, then
               //"i" is assigned the value of that element.

m(5, 2) = 3;   //assign 3 to element (5, 2) of the matrix "m".
```

2.3 Output

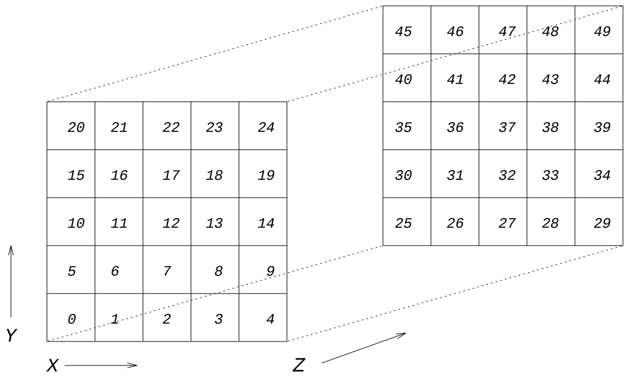
The `operator <<` is overloaded for the classes `Vector`, `Matrix`, and `Cube`. Output may be to `iostream` or `AipsIO` objects. Here are examples of the output produced: Output for a `Vector` is as expected:

```
[0, 1, 2, 3, 4]
```

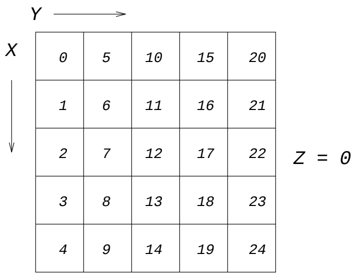
As with all AIPS++ arrays, matrices are stored in FORTRAN order. To run through consecutive elements in memory, vary the first index fastest, then the second. Here is a `Matrix` with consecutive elements labelled:

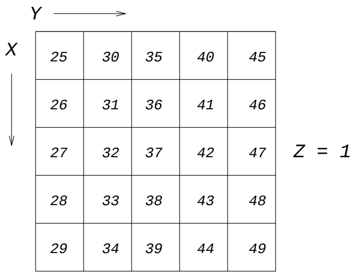
```
Axis Lengths: [5, 4]  (NB: Matrix in Row/Column order)
[ 0, 5, 10, 15
  1, 6, 11, 16
  2, 7, 12, 17
  3, 8, 13, 18
  4, 9, 14, 19 ]
```

Given a 5x5x2 `Cube`, output is explained this way:



We'd like to think of a cube in terms of matrices:





But for actual output, the columns are printed as rows:

```
[0, 0, 0][0, 1, 2, 3, 4]
[0, 1, 0][5, 6, 7, 8, 9]
[0, 2, 0][10, 11, 12, 13, 14]
[0, 3, 0][15, 16, 17, 18, 19]
[0, 4, 0][20, 21, 22, 23, 24]
[0, 0, 1][25, 26, 27, 28, 29]
[0, 1, 1][30, 31, 32, 33, 34]
[0, 2, 1][35, 36, 37, 38, 39]
[0, 3, 1][40, 41, 42, 43, 44]
[0, 4, 1][45, 46, 47, 48, 49]
```

2.4 Slicing

A slice is a reference to a portion of an array. The `Vector`, `Matrix`, and `Cube` classes can be sliced using the class `Slice`. In order to use this class, you must include the appropriate `AIPS++` header file, `#include <aips/Slice.h>`. The class `Slice` is used to define a range on a certain axis. It can be called like `Slice(start,length,increment)`. The default increment is one. Since vectors have only one axis, they are sliced this way:

```
Vector<int> v(15);
v = 0;
v(Slice(0, 5)) = 3;
v(Slice(5, 5)) = 2;
cout<<"the vector is "<<v<<endl;
```

This gives the result:

```
the vector is [3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0]
```

For a `Matrix`, you would use syntax like:

```
Matrix<int> mint(10,10);
mint = 0;
mint(Slice(1,5,2),Slice(1,5,2)) = 2.0;
cout<<"the matrix is "<<mint<<endl;
```

This operation leaves the first row and column of matrix `mint` set to zero, but assigns the value 2 to every other element in every other row of the remaining part of the matrix, as the resulting output indicates:

```
the matrix is Axis Lengths: [10, 10]  (NB: Matrix in Row/Column order)
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0
 0, 2, 0, 2, 0, 2, 0, 2, 0, 2
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
 0, 2, 0, 2, 0, 2, 0, 2, 0, 2
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
 0, 2, 0, 2, 0, 2, 0, 2, 0, 2
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
 0, 2, 0, 2, 0, 2, 0, 2, 0, 2
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
 0, 2, 0, 2, 0, 2, 0, 2, 0, 2]
```

The same idea logically extends to cubes.

2.5 Inquiry

Functions are provided for finding the properties of these array classes. One may wish to determine the size, origin, shape, and end of one of these objects:

```
Matrix<float> m;
...
n = m.nelements();           //function returns the number of elements in "m".
Int o_x, o_y, shape_x, shape_y, end_x, end_y;
m.origin(o_x, o_y);          //Place origin in integers o_x and o_y
m.shape(shape_x, shape_y);   //Get the shape of "m".
m.end(end_x, end_y);         //get the last element of "m".

//Now, this assigns "0.0" to the first element of "m".
m(o_x, o_y) = 0.0;
```

For more information about these classes, refer to the AIPS++ header files `Vector.h`, `Matrix.h`, `Cube.h`, and `Array.h`, and the corresponding source files.

2.6 Changing Shape

The `resize()` function may be used with integers for the `Vector`, `Matrix`, and `Cube` classes:

```
Matrix<Int> m(5,5);
m.resize(2, 2, 1, 1);  // "m" is now a 2x2 matrix with origin (1,1)
m.resize(4,3);         // "m" is now a 4x3 matrix with origin (0,0)
```

The `resize()` function is defined similarly for `Vectors` and `Cubes`.

2.7 Math

Functions are provided to support traditional linear algebra with the `Vector` and `Matrix` classes. These include, for `Vectors`: dotProduct, crossProduct, and norm; and for `Matrices`: multiplication, transpose, conjugate, and adjoint. Other linear algebra functions are currently under development. See the section "Vector and Matrix Math" in the AIPS++ Programming Manual for details.

Also, simple element-by-element arithmetic functions (a += b, etc.) are provided, along with comparison operators, and transcendental functions (sin, cos, sqrt, ...). To use these, you must `#include<aips/ArrayMath.h>`. They are illustrated in more detail in the section describing class `Array`.

2.8 Operations specific to matrices:

Operations are provided to permit access to rows, columns, and diagonals of matrices. For example:

```
Matrix<Int> m(4,4);
m = 2;            //set all elements of m to 2.
m.row(1) = 4;     //set row 1 of "m" to 4.

// result:
// | 2, 2, 2, 2 |
// | 4, 4, 4, 4 |
// | 2, 2, 2, 2 |
// | 2, 2, 2, 2 |

m.column(2) = 3;  //set column 2 of "m" to 3.

// result:
//      | 2, 2, 3, 2 |
//      | 4, 4, 3, 4 |
//      | 2, 2, 3, 2 |
//      | 2, 2, 3, 2 |

Vector<Int> a;
a = m.column(2);  //"a" is now a four element vector, all elements == 3.
```

The function `diagonal()`, like `row()` and `column()`, returns a reference, and so may be used as an lvalue. Here is an example program to illustrate the use of the `diagonal()` function.

```
#include <aips/Matrix.h>
#include <iostream.h>

main()
{
    Matrix<Int> m(4,4);
    for (Int i=-3; i <=3; i++) {
        m.diagonal(i) = i;
        cout << m.diagonal(i) << endl;
    }
    cout << endl << m << endl;
    return 0;
}
```

Here is the result, running from `diagonal(-3)` to `diagonal(3)`, and the resulting matrix:

```
[-3]
[-2, -2]
[-1, -1, -1]
[0, 0, 0, 0]
[1, 1, 1]
[2, 2]
[3]

Axis Lengths: [4, 4]  (NB: Matrix in Row/Column order)
[0, 1, 2, 3
 -1, 0, 1, 2
 -2, -1, 0, 1
 -3, -2, -1, 0]
```

2.9 Operations specific to cubes

You may access particular matrices of a cube with the "xyPlane(Int)" function. For example:

```
Cube<float> c(5,4,3);
c = 2.0;
Matrix<float> m = c.xyPlane(0); //"m" is a reference to the front face of "c"
```

3 n-dimensional Arrays

3.1 Introduction–the class IPosition

In order to use arrays, you must understand the class `IPosition`. An `IPosition` is a vector that is used to index arrays, and always has its origin at zero. For example, if you wish to index a 3-dimensional array, you can use a three-element `IPosition`:

```
IPosition index(3);
//Assume "a" is a previously defined array
```

```
index = 5;                    //index is the vector  <5, 5, 5>.
a(index) = 0;                 // set a(5,5,5) to 0
```

We are now ready to explore the class `Array`.

3.2 Declaration of Arrays

The first thing to note is that you must have the line `#include <aips/Array.h>` in your source code to use arrays.

The array classes are templated. So when you use an array, you must specify what type of data it will hold. To declare a floating point array, use `Array<float>`, to declare an array of integers, use `Array<Int>`, etc. There are four constructors for class Array. When you declare an array, you invoke one of these constructors, depending on the parameters you use in the declaration. Here are examples of each:

```
Array<float> a;
```

This example invokes the constructor `Array<T>::Array<T>()`, and produces an array with no elements (where `T` in this case is `float`).

```
IPosition shape(2), origin(2);
shape(0) = 5;
shape(1) = 6;
origin(0) = 10;
origin(1) = 15;
Array<float> a(shape, origin);
```

Here we invoke the constructor `Array<T>::Array<T>(const IPosition&, const IPosition&)`. The first `IPosition` defines the shape of the array, in this case it is two dimensional, with 5 elements on its first axis and 6 on its second. The second `IPosition` defines the origin of the array, in this case (10,15).

```
IPosition shape(2);
shape(0) = 5;
shape(1) = 6;
Array<float> a(shape);
```

This example invokes the constructor `Array<T>::Array<T>(const IPosition&)`. This makes a two dimensional array, with 5 elements on its first axis and 6 on its second. By default, its origin is (0,0).

```
IPosition shape(1);
shape(0) = 10;
Array<Int> a(shape);    //one dimensional array with 10 elements
Array<Int> b(a);        //Array<Int> b = a; is identical...
```

This invokes the constructor `Array<T>::Array<T>(const Array<T>&)`. This is called the "copy constructor", since it creates a new array based on an existing one. The array `b`, however, is not a copy of `a`; it is actually a reference. In other words, `b` is just another name for `a`. This can have dangerous side-effects. One should avoid writing functions that take arrays as value parameters, as these parameters are actually references to the external argument sent by the caller:

```
void func(Array<float> a)
{
```

```
    a *= 2.0;
}

void main(void)
{
    IPosition shape(3); shape = 5;
    Array<float> b(shape);  // "b" is a 5x5x5 array
    func(b);                // multiplies all elements of "b" by 2.0!
}
```

This happens because when `func(Array a)` is invoked, the `Array` class copy constructor is invoked to create the temporary array object `a`. Since this constructor makes a reference, `a` becomes a reference to `b`. A compelling reason NOT to use pass-by-value, as in `func`, above, is that you cannot pass a class derived from `Array` into func–it will not operate on a `Matrix` for example. Using pass-by-reference or a pointer will allow you to pass an object of a derived class. To assure the outer parameter is untouched you should use a `const` reference:

```
void func(const Array<float> &a)
{
    a *= 2.0;   // illegal, since a is const and can't be modified...
}
```

Or, you can use the `unique()` function to assure that `a` is not a reference:

```
void func(Array<float> a)
{
    a.unique();
    a *= 2.0;
    ...
}
```

If you wish to modify `a` in the function, do not pass by value and rely on the reference semantics of the copy constructor. Pass it by reference instead:

```
void func(Array<float> &a)
{
    a *= 2;
}

void main(void)
{
    IPosition shape(3); shape = 5;
    Array<float> b(shape);  // "b" is a 5x5x5 array
    func(b);                // multiplies all elements of "b" by 2.0!
}
```

3.3 Initialization

The simplest type of initialization is the assignment of all elements of an array to a certain value:

```
IPosition shape(3);
shape(0) = shape(1) = shape(2) = 5; //could be written "shape = 5;"
```

```
Array<float> c(shape);
c = 0.0;                              //Sets all elements to 0.0.
```

You may also initialize from an existing object:

```
IPosition shape(1);
shape = 6;
Array<float> v(shape);
v=1.0;                  //Initializes all elements of "v" to 1.0.
Array<float> w;         //"w" is a zero-length Array<float>
w = v;                  //use Array assignment operator.
```

The array w now is a 6 element Array<float> with each element 1.0. Note that the assignment operator, =, uses copy semantics. So vectors w and v are independent–modifying one does not change the other.

You may initialize one array from another as is done here with w:

```
IPosition shape(1);
shape = 5;
Array<Complex> v(shape); //"v" is a 5 element vector of complex numbers
v = Complex(5.0, 3.0);   //Assign all elements of "v" to 5 + 3i.
Array<Complex> w = v;    //Uses copy constructor.
                         //Array<Complex> w(v); would have the same effect.
```

The vector w is actually a reference to the vector v. In other words, the assignment

```
w(3) = Complex(0,0);
```

also sets v(3) to 0 + 0i. Conversely, changing v also changes w.

3.4 Indexing

Note that from the viewpoint of the applications programmer array indexing in AIPS++ is done in FORTRAN columnwise order.

Indexing is achieved using operator() and IPosition. For example, given a 4-dimensional array a, you could index a certain element using a 4-element IPosition:

```
Array<Int> a(shape); //assume shape is a 4-element IPosition
...
IPosition index(4);
Index(0) = 1;
Index(1) = 2;
Index(2) = 3;
Index(3) = 4;

Int saved_value = a(index);     //saved_value = a(1,2,3,4)
a(index) = 0.0;                 //set a(1,2,3,4) to 0.
```

3.5 Output

You may print the contents of an array to your screen using the operator <<. If you have an array a defined, place this line in your code to write a to stdout:

```
cout << a;
```

Here is the output that is produced if a is a four-dimensional array with origin (0,0,0,0) and axis lengths 4, 3, 2, and 3 respectively:

```
[0, 0, 0, 0][0, 1, 2, 3]
[0, 1, 0, 0][4, 5, 6, 7]
[0, 2, 0, 0][8, 9, 10, 11]
[0, 0, 1, 0][12, 13, 14, 15]
[0, 1, 1, 0][16, 17, 18, 19]
[0, 2, 1, 0][20, 21, 22, 23]
[0, 0, 0, 1][24, 25, 26, 27]
[0, 1, 0, 1][28, 29, 30, 31]
[0, 2, 0, 1][32, 33, 34, 35]
[0, 0, 1, 1][36, 37, 38, 39]
[0, 1, 1, 1][40, 41, 42, 43]
[0, 2, 1, 1][44, 45, 46, 47]
[0, 0, 0, 2][48, 49, 50, 51]
[0, 1, 0, 2][52, 53, 54, 55]
[0, 2, 0, 2][56, 57, 58, 59]
[0, 0, 1, 2][60, 61, 62, 63]
[0, 1, 1, 2][64, 65, 66, 67]
[0, 2, 1, 2][68, 69, 70, 71]
```

The output from left to right, top to bottom is in memory order. The first dimension varies the fastest, followed by the second, and so on down to the last dimension. The elements in the above example are labeled to reflect this.

3.6 Copy and Reference

The function `copy()` is used to produce a copy of an array. The `reference()` function is used to make one array reference another.

```
IPosition shape(1);
shape = 10;
Array<Int> a(shape); a = 5;

Array<Int> b = a.copy();   //"b" is an independent copy of "a".
Array<Int> c;
c.reference(a);            //"c" is a reference to "a".
```

3.7 Slicing

A slice is reference to a portion of an array. Since the slice is itself an `Array`, it may be used in the same way as any `Array`. If, however, it us used on the left hand side of the assignment operator, it modifies the original array that it was built from. A slice is defined using `IPositions`. You define the start index of the slice, the end index of the slice, and an optional increment on each axis.

```
IPosition shape(3);
shape = 10;
Array<float> a(shape);        //"a" is a 10x10x10 array
```

```
Array<float> b;
IPosition Start(3), End(3), Increment(3);
Start(0) = 2; Start(1) = 3; Start(2) = 1;
End(0)   = 4; End(1)   = 8; End(2) = 1;
Increment(0) = 1; Increment(1) = 2; Increment(2) = 1;
b = a(Start, End, Increment);
```

The array `b` is a copy of these elements of `a`: (2, 3, 1), (3, 3, 1), (4, 3, 1), (2, 5, 1), (3, 5, 1), (4, 5, 1), (2, 7, 1), (3, 7, 1), (4, 7, 1). You may also change `a` by assigning to a slice:

```
a(Start, End, Increment) = 0.0;
```

The array `a` is modified; the elements listed in the previous example are set to zero.

3.8 Inquiry

Often it is necessary to ask an Array about its properties. For example, a function may wish to know how many elements there are in the array or what its dimension is. There are several array functions to provide this information. Examples:

```
IPosition shape(3);
shape(0) = 1024;
shape(1) = 1024;
shape(2) = 8;
Array<float> a(shape);

Int dimension = a.ndim();       //ndim gives number of dimensions so
//"dimension" is 3
uInt num_els = a.nelements();   //nelements gives number of elements so
//"num_els" is 8388608 (1024*1024*8)
IPosition o, s, e;
o = a.origin();             //the beginning index "o" is (0,0,0)
s = a.shape();              //the array shape "s" is (1024, 1024, 8);
e = a.end();                //the end index "e" is (1023, 1023, 7);
```

Another inquiry function is `conform()`, which tells whether two arrays are identical in shape:

```
if(a.conform(b)) {
   cout << "a and b are the same shape. " << endl;
} else {
   cout << "a and b are not the same shape." << endl;
}
```

Note that `conform` will return true for two arrays that do not have the same origin, as long as they have the same shape. Also, any scalar conforms with any `Array`. The scalar is considered to represent an `Array` of the same shape as the `Array` it is being tested against; all elements are considered to be the same value as the scalar.

An array with no elements also conforms if used on the left-hand side of an assignment:

```
Array<float> a;
a = b;       //works if 'b' is previously defined...
```

3.9 Changing Shape

Arrays can be re-sized or have their shape changed dynamically. The `resize()` function is used to redefine the shape and origin of an `Array`:

```
IPosition shape(3), origin(3);
shape(0) = 5; shape(1) = 3; shape(2) = 4;
origin = 1;
an_array.resize(shape, origin);
another_array.resize(shape);
```

The array `an_array` is changed to a 5x3x4 array, with origin at (1, 1, 1). The array `another_array` also becomes a 5x3x4 array, but its origin defaults to (0, 0, 0). Copy semantics are used, so there are no other references to an array that has just been resized. You should consider the contents of the newly resized array to be undefined.

If you wish to use reference semantics, the `reform()` function is used. This function changes the shape and origin of the array, but requires that the new array has the same number of elements as the original array. Also, the original array must have increments of one on each axis (eg it may not be a slice with increments). Example:

```
IPosition shape(2);
shape = 5;
Array<Int> a(shape); a = 1; // "a" is a 5x5 array with all elements set to 1.

IPosition newShape(1), newOrigin(1), index(1);
newShape(0) = 25;
newOrigin = 0;
Array<Int> b = a.reform(newShape, newOrigin);
index(0) = 5;
b(index) = 5;   // also changes a(5,0)...
```

The array `a` is still a 5x5 two-dimensional array. Using `b`, however, you may access `a` as though it was a 25 element one-dimensional array.

Another function that creates a reference to an array is the `nonDegenerate()` function. This function removes all degenerate axes– those that have a length of one. This is useful for forcing conformance after slicing:

```
IPosition matrix_shape(2), cube_shape(3);
matrix_shape = 5; cube_shape = 5;
Array<float> m(matrix_shape), c(cube_shape);
m = 3.0; c = 0.1415926536;

IPosition start(3), end(3);
start = 0;
end(0) = 4; end(1) = 4; end(2) = 0;

m += c(start, end).nonDegenerate();
```

Adds the "front face" of the cube `c` to the matrix `m`.

3.10 Raw Storage

Occasionally it is necessary to directly access an array's storage. The functions `getStorage` and `putStorage` are used for this purpose. `getStorage` returns a pointer to an array's storage, and `putStorage` is used to replace an array's storage. Use of these functions should generally be avoided, except where necessary and then only if you're a black belt c++ programmer. One case where direct access to the storage is necessary occurs if you want to pass array data to a FORTRAN subroutine. For example:

```
IPosition shape(2);
shape = 10;
Array<float> a(shape);    // a is a 10x10 array
a = 0.0;
Bool delete_it;
float *a_data = a.getStorage(delete_it);
fortran_func(a_data);     // call a FORTRAN subroutine...
a.putStorage(a_data, delete_it);
```

The array `a` now reflects any changes made to `a_data` by the FORTRAN subroutine. The delete_it flag is set by the call to `getStorage`; sometimes this function makes a copy of the array's storage and returns a pointer to that, depending on considerations such as whether the array has increments on any of its axes. Sometimes the returned pointer points directly to the array's storage. The `delete_it` flag is set to `true` if the storage pointed to by `a_data` is a copy of `a`'s storage, `false` if `a_data` points directly to `a`'s storage. When the call `a.putStorage(a_data, delete_it)` is made, nothing is done if `delete_it` is `false`. If `delete_it` is `true`, then putStorage first copies the data pointed to by `a_data` into the array `a`, then deletes the storage pointed to by `a_data`. If you do not wish to modify the array, (eg if it's `const`), instead of using the `putStorage` method to clean up after use of the array's storage, you must use the `freeStorage` method. For example, if array `a` is `const`, you might have:

```
const float *a_data = a.getStorage(delete_it);
do_something(a_data);   //call a function which uses array a's data
a.freeStorage(a_data, delete_it);
```

The `freeStorage` method prevents memory leaks by freeing up the memory pointed to by the `a_data` pointer if the boolean variable `delete_it` is true.

3.11 Iteration

Special iterator classes are provided to allow iteration of arrays by a certain dimension. This is most useful when dealing with an object of the base class `Array`. To use iterators, you must `#include <aips/ArrayIter.h>`. For example, given a one (or more) dimensional array, you can use a `VectorIterator` to iterate it one vector at a time. The following example shows the use of a `VectorIterator`. It also illustrates the use of the `pastEnd` method, which checks whether the iterator has reached the end of the array, and the use of the `next` method, which moves the iterator by one vector length.

```
IPosition shape(2);
shape(0) = 5;  shape(1) = 4;
Array<float>  m(shape);
VectorIterator<float> iter(m); // Construct a VectorIterator for "m".
```

```
m = 2.0;
cout << m << endl;
while(!iter.pastEnd()) {    // reached the end of the array?
   iter.vector()(4) = 0.0;  // iter.vector() returns a reference to a
                            //  5 element vector, actually a column of m,
                            //  so set last element of each column to 0.0
   iter.next();             // move the iterator
}
cout << m << endl;
```

Here is the output from the above example:

```
Axis Lengths: [5, 4]  (NB: Matrix in Row/Column order)
[2, 2, 2, 2
 2, 2, 2, 2
 2, 2, 2, 2
 2, 2, 2, 2
 2, 2, 2, 2]

Axis Lengths: [5, 4]  (NB: Matrix in Row/Column order)
[2, 2, 2, 2
 2, 2, 2, 2
 2, 2, 2, 2
 2, 2, 2, 2
 0, 0, 0, 0]
```

Given a two (or more) dimensional array, you may iterate it a matrix at a time:

```
IPosition shape(3);
shape(0) = 5; shape(1) = 4; shape(2) = 3;
Array<Int>  c(shape);
MatrixIterator<Int>     iter(c);  // construct a MatrixIterator for "c"

while(!iter.pastEnd()) {
   iter.matrix().row(1) = 5.0;    // set row 1 of each matrix to 5.0.
   iter.next();                   // advance the iterator.
}
cout << c << endl;
```

Here is the output:

```
Ndim=3 Axis Lengths: [5, 4, 3]
[0, 0, 0][0, 5, 0, 0, 0]
[0, 1, 0][0, 5, 0, 0, 0]
[0, 2, 0][0, 5, 0, 0, 0]
[0, 3, 0][0, 5, 0, 0, 0]
[0, 0, 1][0, 5, 0, 0, 0]
[0, 1, 1][0, 5, 0, 0, 0]
[0, 2, 1][0, 5, 0, 0, 0]
[0, 3, 1][0, 5, 0, 0, 0]
[0, 0, 2][0, 5, 0, 0, 0]
[0, 1, 2][0, 5, 0, 0, 0]
[0, 2, 2][0, 5, 0, 0, 0]
[0, 3, 2][0, 5, 0, 0, 0]
```

Another way to iterate an object is using the class `ArrayPositionIterator`. Instead of returning a reference to a vector or matrix within the object that is being iterated, this type of iterator returns the index of an element of the object, in the form of an `IPosition`. Here is an example to illustrate:

```
Matrix<float> m(20, 10);
m = 1.0; //set all elements to 1.0
ArrayPositionIterator element_iter(m.shape(), m.origin(), 0);
ArrayPositionIterator vector_iter (m.shape(), m.origin(), 1);
```

The last parameter of the previous two declarations tells the iterator what dimension to iterate by. The `pos()` function is used to get a reference to the current `IPosition` of the iteration:

```
int sum = 0;
while(!element_iter.pastEnd()) {
   sum += m(element_iter.pos());
   element_iter.next();
}
```

The above code sums all the elements in the matrix `m`. After the loop has been completed, `sum` will have a value of 200. Another example:

```
int sum = 0;
while(!vector_iter.pastEnd()) {
   sum += m(vector_iter.pos()); //use vector_iter instead of elem_iter
   vector_iter.next();
}
```

This code sums all of the elements (0, 0), (0, 1), (0, 2),..., (0, 8), (0, 9), so after the loop has been completed, `sum` will have a value of 10. Note that the `ArrayPositionIterator` is not actually associated with the array it is iterating; It is essentially a server that returns subsequent indices for any array of the shape and origin provided in its constructor.

In future, iterators will allow access in arbitrary order, not just "bottom to top."

4 Array Mathematics

The AIPS++ array classes come with a variety of mathematical functions and arithmetic operators that can be directly applied to the entire contents of an array on an element by element basis. Other functions, such as the `mean` function, allow the programmer to extract information about the data in an array.

4.1 Simple Arithmetic

To use most of the arithmetic functions, you must `#include <aips/ArrayMath.h>`. The first thing to note is that these operations are simple element-by-element operations; true linear algebra is not implemented for general arrays. For example, two-dimensional multiplication is defined as "for all pairs (x,y), take element (x,y) of the first array, multiply it by element (x,y) of the second array, and place the result in (x,y) of the resulting array." This means that the array operands must be the same size (following conformance rules) for these operations to work. Examples:

```
IPosition shape(1);
shape = 5;
Array<float> v(shape);
v = 1.0;
Array<float> w(shape);
w = 2.0;
v += w;                  //All elements of v are now 3.0.
w -= (float)3.0;         //All elements of w are now -1.0.
Array<float> x;
x = v + w;               //All elements of x are now 2.0

IPosition shapem(2);
shapem(0) = 4; shapem(1) = 5;
Array<float> m(shapem), n(shapem);
m = 2.0;
n = 3.0;
Array<float> o;
o = m * n;               //All elements of "o" are 6.0.
o += (float)4.0;         //All elements of "o" are now 10.0.
```

Note that in the previous example we had explicitly cast 4.0 to a float in the statement `o += (float)4.0;`. This is inconvenient and is "fixed" in the upcoming language standard. However the old rules were that global functions (like -=) needed an exact in typematch, whereas member functions (operator=) do type conversions.

4.2 Transcendental and Other Functions

The following transcendental functions can be applied to essentially all numeric types: `cos`, `cosh`, `exp`, `log`, `log10`, `pow`, `sin`, `sinh`, `sqrt`. The meaning of most of them is self-explanatory. They are applied to the members of an array on an element by element basis. Here are some examples.

```
Matrix<float> show(5,5);
show=0.0;
show(1,1) = 3.1415927;
show (2,2) = show(1,1) * (0.5);
show (3,3) = show(2,2) * (0.5);
Matrix<float> result = cos(show);
cout<<"cosines are = "<<result<<endl;
```

Running this code produces the following result:

```
cosines are = Axis Lengths: [5, 5]  (NB: Matrix in Row/Column order)
[1, 1, 1, 1, 1
 1, -1, 1, 1, 1
 1, 1, -4.37114e-08, 1, 1
 1, 1, 1, 0.707107, 1
 1, 1, 1, 1, 1]
```

The `pow` function takes two array arguments (the arrays must be conformant) and raises each element in the first array(left argument) to the power of the value of the corresponding element in the second array (right argument). For example

```
Matrix<float> pow_a(4,4);
pow_a=0.0;
pow_a(0,1) = 3.0;
pow_a (2,2) = 4.5;
pow_a (3,3) = 2.0;

Matrix<float> pow_b(4,4);
pow_b=0.0;
pow_b(0,0) = 5.5;
pow_b(0,1) = 5.0;
pow_b (2,2) = 2.5;
pow_b (3,3) = 0.5;
Matrix<float> result1 = pow(pow_a, pow_b);
cout<<"result of pow = "<<result1<<endl;
```

produces the result

```
result of pow = Axis Lengths: [4, 4]  (NB: Matrix in Row/Column order)
[0, 243, 1, 1
 1, 1, 1, 1
 1, 1, 42.9567, 1
 1, 1, 1, 1.41421]
```

Although the following transcendental functions are templated, they may not make sense for all numeric types: `acos`, `asin`, `atan`, `atan2`, `ceil`, `abs`, `floor`, `fmod`, `tan`, `tanh`, and `pow`. The meaning of most of these is, again, self-explanatory. They are applied to the members of an array on an element by element basis. Here, `pow` is overloaded so that you can raise each element of an array (the left argument) to a scalar power (the second argument); the scalar behaves as if it were a conformant array all of whose elements had the value of the scalar. Here are some examples.

The `floor` method returns the largest floating-point integer value not greater than the input parameter. Here is an example of using `floor`.

```
Matrix<float> example(4,4);
example = 0.0;
example(0,0) = -5.74;
example(1,1) = 6.35;
example(1,0) = -2.18;
example(3,1) = 3.95;
Matrix<float> result_floor = floor(example);
cout<<"result of floor = "<<result_floor<<endl;
```

This produces the result

```
result of floor = Axis Lengths: [4, 4]  (NB: Matrix in Row/Column order)
[-6, 0, 0, 0
 -3, 6, 0, 0
 0, 0, 0, 0
 0, 3, 0, 0]
```

The `ceil` method returns the smallest floating-point integer value not less than the input parameter. Here is an example of using `ceil`.

```
Matrix<float> example(4,4);
example = 0.0;
example(0,0) = -5.74;
example(1,1) = 6.35;
example(1,0) = -2.18;
example(3,1) = 3.95;
Matrix<float> result_ceil = ceil(example);
cout<<"result of ceil = "<<result_ceil<<endl;
```

This produces the result

```
result of ceil = Axis Lengths: [4, 4]  (NB: Matrix in Row/Column order)
[-5, 0, 0, 0
 -2, 7, 0, 0
 0, 0, 0, 0
 0, 4, 0, 0]
```

The `fmod` method computes the modulo floating-point remainder of x/y. The `fmod` method returns the value x-iy for some i such that if y is non-zero, the result has the same sign as x and magnitude less than the magnitude of y. The following code illustrates the result of using the `fmod` method.

```
Matrix<float> example1(4,4);
example1 = 0.0;
example1(0,0) = -5.0;
example1(1,1) = 6.5;
example1(1,0) = -2.0;
example1(3,1) = 3.2;
Matrix<float> example2(4,4);
example2 = 1.0;
example2(0,0) = 2.0;
example2(1,1) = 3.0;
example2(1,0) = -0.3;
example2(3,1) = 1.6;
Matrix<float> result_fmod = fmod(example1, example2);
cout<<"result of fmod = "<<result_fmod<<endl;
```

Compiling and running this code produces the result:

```
result of fmod = Axis Lengths: [4, 4]  (NB: Matrix in Row/Column order)
[-1, 0, 0, 0
 -0.2, 0.5, 0, 0
 0, 0, 0, 0
 0, 0, 0, 0]
```

4.3 Obtaining Global Information about Array Contents

Often we wish to obtain information about the global parameters of an array - what is the minimum or maximum value in an array? Where are these minima and maxima located? What is the rms, or standard deviation, of an array? And so on. The AIPS++ array classes come with the following pre-defined methods which allow us to answer such questions: `min`, `max`, `minMax`, `sum`, `product`, `mean`, `variance`, `stddev`, `avdev`, and `median`. See the `ArrayMath.h` header file for detailed information about each method.

For example, to find the minimum and maximum values in an array, you can use the `minMax` method. Here is an example:

```
Matrix<float> example3(4,4);
example3 = 1.0;
example3(0,0) = 2.0;
example3(1,1) = 3.0;
example3(1,0) = -0.3;
example3(3,1) = 1.6;
float array_min, array_max;
minMax(array_min, array_max, example3);
cout<<"the minimum = "<<array_min<<endl;
cout<<"the maximum = "<<array_max<<endl;
```

Running this code gives the expected result:

```
the minimum = -0.3
the maximum = 3
```

`minMax` is actually an overloaded function, and can also be used to return the locations of the minima and maxima, in addition to their values. For example:

```
Matrix<float> example3(4,4);
example3 = 1.0;
example3(0,0) = 2.0;
example3(1,1) = 3.0;
example3(1,0) = -0.3;
example3(3,1) = 1.6;
float array_min, array_max;
IPosition min_loc(2), max_loc(2);
minMax(array_min, array_max, min_loc, max_loc, example3);
cout<<"the minimum = "<<array_min<<endl;
cout<<"its location is = "<<min_loc<<endl;
cout<<"the maximum = "<<array_max<<endl;
cout<<"its location is = "<<max_loc<<endl;
```

Running this code gives the expected result:

```
the minimum = -0.3
its location is = [1, 0]
the maximum = 3
its location is = [1, 1]
```

Often radio astronomers wish to compute the mean and standard deviation of an array of numbers, usually to determine the noise characteristics of an image. AIPS++ provides the methods `mean` and `stddev` for obtaining these two parameters.

```
Matrix<float> example3(4,4);
example3 = 1.0;
example3(0,0) = 2.0;
example3(1,1) = 3.0;
example3(1,0) = -0.3;
example3(3,1) = 1.6;
example3(0,2) = -5.2;
example3(3,0) = 7.1;
float array_mean, array_stddev;
```

```
array_mean = mean(example3);
array_stddev = stddev(example3);
cout<<"the mean = "<<array_mean<<endl;
cout<<"the standard deviation = "<<array_stddev<<endl;
```

gives the result

```
the mean = 1.1375
the standard deviation = 2.34375
```

4.4 Conversions Between Real and Complex Arrays

AIPS++ provides a number of methods for getting the real and imaginary, or amplitude and phase, information out of an array of complex numbers. As you might expect these methods are named `real`, `imag`, `amplitude` and `phase`. The phase is computed in radians.

Here is an example of the use of these functions:

```
Matrix <Complex> example5(4,4);
example5 = Complex(5.0, 3.0);
Matrix<float> example6 = real(example5);
Matrix<float> example7 = imag(example5);
Matrix<float> example8 = amplitude(example5);
Matrix<float> example9 = phase(example5);
cout<<"the real matrix is "<<example6<<endl;
cout<<"the imaginary matrix is "<<example7<<endl;
cout<<"the amplitude matrix is "<<example8<<endl;
cout<<"the phase matrix is "<<example9<<endl;
```

Running this code produces the result:

```
the real matrix is Axis Lengths: [4, 4]  (NB: Matrix in Row/Column order)
[5, 5, 5, 5
 5, 5, 5, 5
 5, 5, 5, 5
 5, 5, 5, 5]

the imaginary matrix is Axis Lengths: [4, 4]  (NB: Matrix in Row/Column order)
[3, 3, 3, 3
 3, 3, 3, 3
 3, 3, 3, 3
 3, 3, 3, 3]

the amplitude matrix is Axis Lengths: [4, 4]  (NB: Matrix in Row/Column order)
[5.83095, 5.83095, 5.83095, 5.83095
 5.83095, 5.83095, 5.83095, 5.83095
 5.83095, 5.83095, 5.83095, 5.83095
 5.83095, 5.83095, 5.83095, 5.83095]

the phase matrix is Axis Lengths: [4, 4]  (NB: Matrix in Row/Column order)
[0.54042, 0.54042, 0.54042, 0.54042
 0.54042, 0.54042, 0.54042, 0.54042
 0.54042, 0.54042, 0.54042, 0.54042
```

```
 0.54042, 0.54042, 0.54042, 0.54042]
```

The `ComplexToReal` and `RealToComplex` methods are provided so that we may easily convert a complex array to an equivalent grouping of real numbers, and vice versa. These methods are useful because sometimes we wish to handle an array of complex numbers as if it were just an array of float or doubles. (This is sometimes a requirement for doing FFTs.) For example:

```
Matrix <Complex> example5(4,4);
example5 = Complex(5.0, 3.0);
Matrix<float> c_to_r = ComplexToReal(example5);
cout<<"here is a complex matrix "<<endl;
cout<<example5<<endl;
cout<<"the equivalent real matrix is "<<endl;
cout<<c_to_r<<endl;
```

Running this code produces the output:

```
here is a complex matrix
Axis Lengths: [4, 4]  (NB: Matrix in Row/Column order)
[(5, 3), (5, 3), (5, 3), (5, 3)
 (5, 3), (5, 3), (5, 3), (5, 3)
 (5, 3), (5, 3), (5, 3), (5, 3)
 (5, 3), (5, 3), (5, 3), (5, 3)]

the equivalent real matrix is
Axis Lengths: [8, 4]  (NB: Matrix in Row/Column order)
[5, 5, 5, 5
 3, 3, 3, 3
 5, 5, 5, 5
 3, 3, 3, 3
 5, 5, 5, 5
 3, 3, 3, 3
 5, 5, 5, 5
 3, 3, 3, 3]
```

5 Logical Operations on Arrays

The AIPS++ array classes allow you to perform logical comparisons between two arrays on an element by element basis, and then to set a boolean flag if the specified criteria are met for all elements of each array. The methods provided are `allLE`, `allLT`, `allGE`, `allGT`, `allEQ`, and `allNE`. These names are reminiscent of their FORTRAN counterparts. Each `all` method is called as `allXX(left, right)`, where `left` and `right` are conformant arrays. For example, if we called `allLE(left, right)`, and all elements of `left` are less than or equal to all elements of array `right` then `allLE` would return true. For example:

```
Matrix<float> b1(4,4);
b1 = 5.0;
Matrix<float> b2(4,4);
b2 = 2.0;
Bool compare = allLE(b1, b2);
cout<<"all elements of b1 are LE those of b2 = "<<compare<<endl;
```

This returns the result

```
all elements of b1 are LE those of b2 = 0
```

where 0 equates to false.

However, if we set `b1 = -5.0`

```
Matrix<float> b1(4,4);
b1 = -5.0;
Matrix<float> b2(4,4);
b2 = 2.0;
Bool compare = allLE(b1, b2);
cout<<"all elements of b1 are LE those of b2 = "<<compare<<endl;
```

we get the result

```
all elements of b1 are LE those of b2 = 1
```

where 1 equates to true.

These functions are all overloaded so that you can perform an element by element comparison between an array and a scalar; the scalar behaves as if it were a conformant array all of whose elements had the value of the scalar. The result is true only if the comparison is true for every element of the array. For example

```
Matrix<float> b3(4,4);
b3 = -5.0;
float test_val = -2.0;
Bool compare1 = allLE(b3, test_val);
cout<<"all elements of b3 are LE test_val = "<<compare1<<endl;
```

we get the result

```
all elements of b3 are LE test_val = 1
```

where 1 equates to true.

On the other hand you might want to compare two arrays on an element by element basis, and set a flag to true, if at least one element by element intercomparison meets your criteria. The `any` collection of logical methods is provided for this purpose. The methods provided are `anyLE`, `anyLT`, `anyGE`, `anyGT`, `anyEQ`, and `anyNE`. Again, each `any` method is called as `anyXX(left, right)`, where `left` and `right` are conformant arrays. For example, if we called `anyLE(left, right)`, and any element of `left` is less than or equal to its corresponding element in array `right` then `anyLE` would return true. For example:

```
Matrix<float> b6(4,4);
b6 = 5.0;
b6(0,0)=1.0;
Matrix<float> b7(4,4);
b7 = 2.0;
Bool compare3 = anyLE(b6, b7);
cout<<"at least one element of b1 is less than those of b2 = "<<compare3<<endl;
```

This returns the result

```
at least one element of b1 is less than those of b2 = 1
```

where 1 equates to true.

Like the `all` methods, the `any` methods are overloaded so that you can perform an element by element comparison between an array and a scalar; the scalar behaves as if it were a conformant array all of whose elements had the value of the scalar. The result is now true if the comparison is true for any element of the array. For example

```
Matrix<float> b3(4,4);
b3 = 5.0;
b3(2,2) = -9.0;
float test_val = 2.0;
Bool compare1 = anyLE(b3, test_val);
cout<<"at least one element of b3 is LE test_val = "<<compare1<<endl;
```

we get the result

```
at least one element of b3 is LE test_val = 1
```

where 1 equates to true.

6 A General Purpose Method using Arrays

We now discuss the use of the AIPS++ array classes to build an actual application. As we shall see, the array classes allow us to express radio astronomy algorithms in a very elegant way. However, we must be aware that processing speed may be traded for elegance!

We describe the development of the function `conv_correct()` from the AIPS++ class `GridTool`. Aperture synthesis radio telescopes collect data in the Fourier domain; this data must be convolved on to a regular grid before a FFT to the real image domain can be done. This convolution causes the resulting image to be attenuated by a factor which increases with distance from the image centre and which must be corrected for. Each element of the image must be multiplied by a correction factor that varies over the image. The image to be corrected might be a matrix or a cube.

We start with two definitions of this (overloaded) function: one that operates on matrices, and another that operates on cubes. Here is the function that operates on matrices:

```
void
GridTool::conv_correct(Matrix<float>& image)
//
// This function corrects an image for the attenuation
// caused by convolution in the fourier plane when the data were gridded.
//
// calling parameters:
// image - matrix of data containing the image to be corrected
//
{
   int rows = image.nrow();      // get the number of rows in "image"
   int cols = image.ncolumn();   // get the number of columns in "image"

   //"grid" is a two element vector that will hold the current values of
   //loop counters i and j.  This vector is passed as an argument to
   //the function "grid_corr()", which returns the correct value associated
   //with position (i, j) in "image".

   Vector<Int> grid(dimension);  //"dimension" is a GridTool private member
                                 // which has value 2 for a Matrix
```

```
    grid = 0;                        //zero all elements of the vector "grid"
    for (int j=0; j<cols; j++) {  //i and j iterate all elements of "image"
       grid(1) = j;
       for (int i = 0; i < rows; i++) {
          grid(0) = i;             //grid is now the vector <i, j>
          //Now, perform the necessary transformation on location (i, j)
          //of the matrix "image":
          image(i, j) = image(i, j) * grid_corr(grid);
       }
    }
}
```

Here is the same function that operates on cubes:

```
void
GridTool::conv_correct(Cube<float> &image)
//
// calling parameters:
// image - cube of data describing the image to be corrected
//
{
   int rows, cols, nz;
   //Get the number of rows, columns, and planes from the cube "image"
   image.shape(rows, cols, nz);

   //"grid" is a three element vector that will hold the current values of
   //loop counters i, j, and k.  This vector is passed as an argument to
   //the function "grid_corr()", which returns the correct value associated
   //with position (i, j, k) in "image".

   Vector<Int> grid(dimension);  //"dimension" is a GridTool private member.
                                 // which has value 3 for a Cube

   grid = 0;                         // zero all elements of "grid"
   for (int k = 0; k < nz; k++) {    // i, j, and k iterate all
                                     // elements of "image"
      grid(2) = k;
      for (int j = 0; j < cols; j++) {
         grid(1) = j;
         for (int i = 0; i < rows; i++) {
            grid(0) = i;
            // "grid" is now the vector <i, j, k>.
            // Perform the transformation
            // on location (i, j, k) of the cube "image":
            image(i, j, k) = image(i, j, k) * grid_corr(grid);
         }
      }
   }
}
```

Aside from the use of overloading, this is how this problem would be coded in any imperative programming language such as C or Fortran. Can we improve on this using object-oriented techniques and the AIPS++ library? First, these two functions are virtually identical. Also both the class `Matrix` and the class `Cube` inherit from the class `Array`. Therefore we can merge the two functions into the following one which uses the generic Array class.

```
void
GridTool::conv_correct(Array<Float> &image) {
...
}
```

However, how we go about doing this can have a significant impact on performance. (Note: the following development is based on the initial AIPS++ library. As the library develops and is made more efficient, some of these details may not apply.)

Here is a first attempt at the function, which uses class `ArrayPositionIterator`:

```
void
GridTool::conv_correct(Array<float> &image)
{
   //construct an ArrayPostionIterator to iterate "image":
   ArrayPositionIterator position(image.shape(), image.origin(), 0);


   while(!position.pastEnd()) {
      //perform correction
      image(position.pos()) *= grid_corr(position.pos());
      position.next();   //advance iterator.
   }
}
```

An `ArrayPositionIterator` is now used to iterate each of the elements in the array `image`.

We have succeeded in replacing the two functions conv_correct with a function that is shorter, more elegant, and in fact more powerful, since it can operate on arrays of any dimension. There is one problem though: let's say that our original function for the class `Matrix` took X seconds to process a 1024 x 1024 Matrix, which represents a fairly standard size of image we can expect to handle in AIPS++. Unfortunately our new "generic" function will take roughly twelve times as long! Clearly, this performance hit is not acceptable.

We must identify the inefficiencies and eliminate as many of them as we can. First, notice that the while loop must make a call to the function `ArrayPositionIterator::pastEnd()` for each iteration–with a 1024x1024 matrix this is over one million calls. Let's eliminate that first:

```
void
GridTool::conv_correct(Array<float> &image)
{
   //construct an ArrayPostionIterator to iterate "image":
   ArrayPositionIterator position(image.shape(), image.origin(), 0);

   IPosition index;
```

```
    int size = image.nelements(); // "Size" is the number of elements
                                  // in "image"
    for(int i=0; i<size; i++) {
       index = position.pos();           //get the current index values
       image(index) *= grid_corr(index); //perform correction
       Position.next();                  //advance iterator
    }
}
```

This simple change results in a vast improvement; the function now takes about 3X seconds to process a 1024x1024 array. Still, we would like to approach the speed of the original code, if possible.

The next logical step is to try to reduce or eliminate calls to `ArrayPositionIterator::next()`. To do that let's use a `VectorIterator`. This is somewhat like an `ArrayPostionIterator`, but it is associated with a specific array object. Recall that the method `VectorIterator::vector()` returns a const reference to the current vector of the iteration. Calls to `VectorIterator::next()` move on to the next vector of the object being iterated. Let's see if this can help us:

```
void
GridTool::conv_correct(Array<float>& image)
{
  VectorIterator<float> image_iter(image);
  Int start, end;
  image_iter.vector().origin(start); // start and end refer to the
                                     // starting index
  image_iter.vector().end(end);      // and last index of the vector
                                     // "image_iter.vector()".
  IPosition index(image.ndim());

  while (!image_iter.pastEnd()) {
     index = image_iter.pos();          //get the current IPosition.
     for(Int i=start; i <= end; i++) {  //iterate the current vector.
       image_iter.vector()(i) *= grid_corr(index);
       index(0)++;                      //advance the index manually
                                        // so as to avoid calls to next().
     }
     image_iter.next();
  }
}
```

Because the `i` loop is counting the correct number of elements for a column, we don't need to worry about `index(0)++` giving us an illegal index. This code finally gets us to about X seconds to process a 1024x1024 array. We have perhaps lost some readability during this process of refinement, but this code is still better than the code we started with, and now equally efficient. This technique of reducing an n-dimensional problem to a series of one or two dimensional problems using iterators has proved useful in several places in the AIPS++ library.