AIPS++ N-Dimensional Array Classes

A. G. WILLIS[1], M. P. HEALEY[1], AND B. E. GLENDENNING[2]

[1]*Dominion Radio Astrophysical Observatory, Penticton, B.C., V2A 6K3, Canada*
[2]*National Radio Astronomy Observatory, Charlottesville, VA 22903*

ABSTRACT

This article describes a set of C++ classes developed for the AIPS++ project. These classes handle arrays having an arbitrary number of dimensions. We give an overview of the methods available in these classes and show some simple examples of their use. Finally we describe the use of these classes to develop a radio astronomy application and discuss some of the performance issues that must be considered when these classes are used.

1 INTRODUCTION

Seven radio astronomy observatories that operate aperture synthesis radio telescopes have joined forces to develop an object-oriented data processing system called AIPS++. The seven observatories are the National Radio Astronomy Observatory (NRAO), based in Charlottesville, VA, the Netherlands Foundation for Research in Astronomy (NFRA), the Australia Telescope National Facility (ATNF), the Nuffield Radio Astronomy Laboratory (NRAL) at Jodrell Bank, England, The Giant Metre Wavelength Telescope in India (GMRT), the Berkeley - Illinois - Maryland Array (BIMA), and the Dominion Radio Astrophysical Observatory (DRAO) in Canada. This is truly a worldwide project!

AIPS++ is an acronym for Astronomical Information Processing System (incremented by one). It is designed to be a replacement for the original AIPS system developed by the NRAO in the early 1980s. The original AIPS was written in Fortran 66 so that it would be portable to almost any computer with a Fortran compiler. Portability was important because a primary goal of AIPS was to ensure that astronomers who observed at NRAO's Very Large Array (VLA) radio telescope could take data back to their home institutions and reduce the data on a local computer.

Although AIPS has proved to be a very successful data reduction system installed at some 200 sites worldwide, it is showing its age. System maintenance is difficult, and the development of new algorithms is painful. About 2 years ago, the decision was made to replace the original AIPS with a modern object-oriented system written in C++.

2 WHY DEVELOP A LIBRARY TO HANDLE N-DIMENSIONAL ARRAYS?

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 pic-

Received April 1993
Revised July 1993

DRAO is part of the Herzberg Institute of Astrophysics, National Research Council of Canada

Scientific Programming, Vol. 2, pp. 239–246 (1993)
CCC 1058-9244/94/040239-08

ture 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 an N-dimensional array, the actual number of dimensions being defined by the application programmer. Because 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. At the moment most operations (arithmetic, logical operations, . . .) are actually performed in the base array classes and only obvious specializations such as indexing or extracting the diagonal of a matrix, are implemented in the inherited classes.

An additional advantage of this approach is that we can create methods in other classes that define the generic array class as an input or output parameter, but then use these methods with Vector, Matrix, or Cube objects, without having to overload the method.

The AIPS++ array classes will implement the mathematical functionality required for radio astronomical applications (image processing and the like). At the moment the classes are at a very early stage of development; they are only a few man-months old. Some of the discussion in this document will change as the library matures and becomes more tuned.

The library is fully templated. Originally the templates were based upon the Texas Instrument "COOL" preprocessor [1], although the classes are currently being converted to "ARM" [2] style templates as they are now widely available. With ARM templates it is much easier to specialize operations for certain types, and optimizations to (e.g., BLAS) will be made more frequently in the derived types.

The array classes use reference counting to implement array sections ("slicing" in Rogue Wave) and return by value. At the moment the copy constructor uses reference semantics although this may change because it violates the "principle of least astonishment." Copy-on-write semantics are not supported.

Note that from the viewpoint of the applications programmer array indexing in AIPS++ is done in Fortran columnwise order. Also, we want it to be possible to map F90 on to the AIPS++ arrays, so that F90 machines can do the actual arithmetic. A concrete example where this affected things was that conformance rules were changed so that only the shape, not the origin, was considered as in F90.

At the time we started this project we were not aware of any other N-dimensional array classes that were implemented with templates. Our understanding is that Rogue Wave's latest math.h++ library has similar features. However, the AIPS++ package will eventually be made freely available under the conditions of the GNU General Public License to any astronomical institution (or any one else for that matter) that requests a copy of the package. Because many small astronomical institutions are unable to afford commercial software, all components of AIPS++ must be self-contained and not rely on calls to commercial software packages.

3 ARRAY EXAMPLES

The easiest way to introduce the AIPS++ Array classes is to give some examples.

3.1 Declaration of 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. 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`).

```
// An IPosition is a zero-based
// vector used for indexing arrays of
// arbitrary dimension.
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 five elements on its first axis and six 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 five elements on its first axis and six 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 copy constructor

```
Array<T>::Array<T>(const Array<T>&)
```

The array b, however, is not a copy of a; it is actually a reference.

3.2 Indexing

Indexing is achieved using operator() and IPosition. For example, given a four-dimensional array a, you could index a certain element using a four-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);
                  //save a (1,2,3,4)
a (index) = 0.0;
            //set a(1,2,3,4) to 0.
```

One advantage of the derived classes Vector, Matrix, and Cube is that we may index them using integers, without need of IPositions:

```
Matrix<float> m(5,5);
...
m(1,2) = 5.5; //set element (1,2) to 5.5
```

4 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();        //"dimension" gets 3.
uInt num_els = a.nelements();    //"num_els" is 8388608 (1024*1024*8)
IPosition o, s, e;
o = a.origin();              //"o" is (0,0,0)
s = a.shape();               //"s" is (1024, 1024, 8);
e = a.end();                 //"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 along as they have the same shape.

4.1 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` of unknown dimension. For example, given an array of dimension 2 or higher, you can use a `VectorIterator` to iterate it one Vector at a time:

```
IPosition shape(2);
shape(0) = 10;  shape(1) = 8;
Array<float>  m(shape);
VectorIterator<float> iter(m);
// Construct a VectorIterator for "m".

while(!iter.pastEnd()) {
   // iter.vector() returns a
   //reference to a 10 element
   //vector, actually a
   // column of m.
   iter.vector()(4) = 0.0;
   iter.next();
}
```

Given a three (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.
}
```

Another way to iterate an object is using the class `IPositionIterator`. 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`. 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). 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 the future, iterators will allow access in arbitrary order, not just "bottom to top."

5 A GENERAL PURPOSE METHOD USING ARRAYS

To describe the use of Array methods in an actual application we will discuss 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 that increases with distance from the image center 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)
//
// 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 now a three element vector that will hold the current
   //values of loop counters i, j, and k.

   Vector<Int> grid(dimension);  //"dimension" 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 discussion is based on the initial AIPS++ library. As the library develops and is made more efficient, many of these details likely will 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);

   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
   }
}
```

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, because 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 × 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 three times as long! Clearly, this performance hit is not acceptable.

Notice that the line index = position.pos(); is also executed over one million times for our test array. Although pos() is an inlined function, the compiler we have been using does not seem to have given us the performance we require. Is there some way around this? There is, but it is a little tricky. First, the ArrayPositionIterator::pos() function does not actually return on IPosition object, but a constant reference to an IPosition object. Its prototype is:

```
const IPosition
 &ArrayPositionIterator::pos() const;
```

Note: An IPosition is an n-element vector of positive numbers. If the IPosition index has the value (0,0), and image is a matrix or two-dimensional array, then image(index) returns the value at image(0, 0). This means that the function pos() returns a reference to, or alias for, some IPosition that is (in this case) a private member of the class ArrayPositionIterator. The first const keyword indicates that this reference may not be used as an l-value, i.e., this is illegal:

```
IPosition I;
ArrayPositionIterator iterator(shape,
                               origin, step);
...
Iterator.pos() = I; //Error, can't
          assign to const reference!
```

Without the const modifier, the above code would be legal and correct (assuming that I is the correct dimension). The second const keyword simply says that the function pos() does not modify the ArrayPositionIterator with which it is associated. In other words, if we make the declaration:

```
const ArrayPositionIterator iterator
                (shape, origin, step);
```

then the call

```
iterator.pos()
```

is legal and does not modify the constant object iterator. A call to a nonconst function, such as next(), is illegal for the const object. Armed with this understanding of the function pos(), we can make the following improvement to our code:

```
void
GridTool::conv_correct(Array<float> &image)
{
   int i, Size;
   ArrayPositionIterator Position(image.shape(), image.origin(), 0);

   Size = image.nelements();
   const IPosition& index = Position.pos();
   for (i=0; i<Size; i++) {
      image(index *= grid_corr(index);        //perform correction
      Position.next();                        //advance iterator
   }
}
```

Now, what is happening is that the IPosition object referenced by the return value of the call to Position.pos() is also referenced by the const IPosition& index. So, we can move the call to the pos() function outside the while loop-the calls to Position.next() update the IPosition referred to by the call to Position-.pos(), and hence also the IPosition referred

to by index. So, next time around, image(index) is the next element of image. The above code gets us down to about 2X seconds to process a 1024 × 1024 array. Things are getting better but...

The next logical step is to try to reduce or eliminate calls to ArrayPositionIterator::next(). To do that let us use a VectorIterator. This is somewhat like an ArrayPositionIterator, 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 us 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--avoid calls to next().
      }
      image_iter.net();
   }
}
```

Because the i loop is counting the correct number of elements for a column, we do not need to worry about index(0)++ giving us an illegal index. This code finally gets us to about X seconds to process a 1024 × 1024 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.

REFERENCES

[1] Texas Instruments Inc., *C++ Object-Oriented Library User's Manual.* Austin, TX: Information Technology Group, 1990.

[2] M. A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual.* Reading, MA, Addison-Wesley, 1990.