Users Guide to Image Coordinate Systems

Mark Holdaway
Sanjay Bhatnagar
14 March 1992

The following is our evaluation of the use and design of the Image class and its underlying coordinate systems. We are writing from the perspective of programmers who wish to use the Image class as a service. While the Image class and its coordinate structure looks very pleasing and seems to obey encapsulation, its use in application programs is not easy, conceptual encapsulation is broken, and optimization of routines such as DFTs and FFTs breaks the encapsulation.

1 The Coordinate System is Cumbersome

Any application programmer must know a lot about the underlying conceptual structure of the Image and its three coordinate systems. For example, if I want to set the parameters associated with a coordinate system, I must write something like

```
nx = outImage.GetDim(1); ny = outImage.GetDim(2);

CoordSys *myCoords = outImage.GetCoordSys();

ImageCoord myDelta = myCoords.GetDeltaCoord();
double xcell = myDelta.GetImageCoord(1);
double ycell = myDelta.GetImageCoord(2);

ImPixelCoord myRefPix = myCoords.GetRefPix();
double xrefpix = myRefPix.GetImPixCoord(1);
double yrefpix = myRefPix.GetImPixCoord(2);
                .
                .
```

while in the system SDE the analogous operation is implemented with a single line of code:

```
CALL CRDPUT (IMAGE, NAX, TYPE, NAXIS, RVAL, RPIX, DELT, ROTA)
```

the higher level coordinate information to determine the pixel location. One suggested solution to this is to use the methods of the image class to access the data and to indicate the astronomical (Image) coordinates of that point. This, however, leads to very inefficient code for highly vectorizable operations on regular orthogonal coordinate systems.

4 Recomendations

We feel that there has not been strong communication between the (u, v) group and the Image group. The Image group has created an independent system of code which is logically consistent but does not mesh very well with the requirements of the (u, v) data group. The (u, v) group developed its code with a DummyImage class which was very simple and supplied us with everything we needed. In retrospect, the (u, v) group should have asked the Image group to supply certain data members and functions in a preliminary image class so the Image class and one of its primary clients could evolve together.

At this point, we feel strongly that the coordinate systems and Image classes need to be thought about in terms of ease of use, minimization of confusion, and the capability of writing efficient code without breaking the encapsulation of the Image class. We do not like the ImPixel coordinate system, but we are open to solutions which use the ImPixel system if our objections can be met.

OBSRA and OBSDEC are required. Perhaps even a vector of OBSRA and OBSDEC. We should keep our eyes open for a better way of doing this.

There seems to be no axis type data member of CoordSys. One cannot assume a spectral line cube will always be in X, Y, F order. For that matter, we will probably stick things onto coordinate axes that none of us have thought of yet.

The coordinates of an image and the coordinates of a Fourier Transform of that image will be related to each other. The coordinate system must know about this. The parameters nx and ny are not considered to be part of the coordinate system. However, when the transform image is considered, the CELLSIZE of the transform image is equal to 1.0 / (CELLSIZE of the image * NX). In this way, NX and NY get a little insestuous with the coordinate system, and perhaps should be considered as part of the coordinates. Needs some thought.

Also, a piece of thought which is in the YEG domain: what astronomical coordinates are needed for the YEGS? The OBSRA, OBSDEC, reference RA, reference DEC, and coordinate projection type. We YEGGERS need to see what parts of the Image coordinate system we can steal for ourselves, and we need to make the interface between the YEG coordinate system and the UV grid, which should be considered to be just another image, except with different coordinate axis types ("UU—SIN", "VV—SIN"...).

Consider a MEM based program in which you want to take the current iteration's model (128 x 128) and convolve it with the PSF. An efficient way of doing this is to take the psf (256 x 256) and do an FFT to make the transfer function (256 x 256). Then, when we want to convolve the (128 x 128) image, we just do a (128 x 128) —¿ (256 x 256) FFT, multiply by the transfer function, then back-FFT (256 x 256) —¿ (128 x 128). In doing these FFT's, it is useful to store the dimensions and coordinate systems of the previous image in the current (FFT'd) image...it aids in going back.

3 Efficiency vs Incapsulation

Access of the data in the Image class is not consistent with optimized code. Currently, the Image data can be accessed one pixel at a time or by a pointer to a copy of the image (the copy can be read back into the image). DFTs and FFTs should not have to access the data one pixel at a time. However, if the pointer-to- array-copy access is chosen, we mo longer have access to the power of the Image class and its methods: we might as well be writing in FORTRAN.

There is a subtle interaction between the problem of data access and the problem with coordinate confusion. Remember, the data are stored in the order of Pixel coordinates. However, the values returned for `xrefpix` and `xcell` in the example code above refer to the ImPixel coordinate system and its relationship to the Image coordinate system. Great confusion results when one tries to access the data efficiently through the pointer-to-copy and then tries to use

2 Conceptually Complicated and Uncleanly Divided

The AIPS++ coordinate information *access* might be simplified soon, but the application programmer still must know about and understand Image Coordinates, Image Pixel Coordinates, and Pixel Coordinates. While these three levels of abstraction facilitate some operations and enable the applications programmer to consider any corner of an image to be `0, 0`, they also lead to a great deal of confusion. The primary confusion arises from the fact that the data are firmly connected to the Pixel coordinate system and the fact that there are operations one can do on the levels of any of the three coordinate systems. For example, in implementing a CLEAN algorithm, one gets the maximum value of an array with

```
Pixel maxpix = myImage.Maximum();
```

To designate the center of the image to be the pixel where the maximum was reported, one has to do

```
PixelCoord myPixCoord = maxpix.GetPixelCoord()
myImage.SetCenPix(myPixCoord);
```

If, for some reason, one has to shift the center of the image, one needs to get the ImPixelCoord out in the code as well. The net effect of this is that the Pixel, PixelCoord and ImPixelCoord are all visibile at the application level throught out the code. While there might be good reasons to encapsulate functionality into the three coordinate systems, the concepts are not encapsulated and the programmer must be thinking about them all simultaneously and must remember which functions belong to which coordinate systems. A bare minimum of two coordinate systems (data storage coordinates and a mapping of the data storage coordinates onto the sky) are required. The programmer should have, as much as possible, an independent and complete set of permisible operations in each coordinate system's interface (or the Image's interface). As it stands now, it is impossible to think of the Image as a set of values on a regular grid defined by "one" coordinate system. While the Image and its coordinate system must be flexible enough to deal with difficult cases such as non-linear or non-orthogonal coordinate systems, this generality should not overburden the simple coordinate geometries.

We also note that the Image class has 38 methods in it, and more classes than we wish to count. These will obviously grow.

2.1 Coordinate Wish List

Here are a number of things which I find useful about SDE and some random thoughts about the current coordinate systems.