aips++ prototype

aips++ *development group, Charlottesville* [1]

August 28, 1992

[1]Sanjay Bhatnagar, Mark Calabretta, Brian Glendenning, Lloyd Higgs, Bob Hjellming, Mark Holdaway, Friso Olnon (editor), Bob Payne, Bob Sault, Dave Shone, Mark Stupar, Peter Teuben

Contents

Chapter 1

Overview

1.1 Why a prototype?

In early February a group met in Green Bank. This group consisted of six of the **aips++** development group from Charlottesville and six others (from NRAO, NFRA and NRAL). The goal of this group was to analyse the process of calibration and imaging astronomical data, in particular radio-astronomical data. Most participants were satisfied with the results from that meeting, and their report is summarized in [?]. One of the participants has very strong disagreements with [?], his disagreements and a proposal to start afresh are outlined in [?].

After the Green Bank meeting and followup discussion in Charlottesville (including Andrew Klein, our OO/C++ consultant) it was decided to write a small prototype for the following reasons:

- To attempt to see if there was anything grossly wrong with the "Green Bank" model (it is fair to point out that a small prototype may inadequately push the boundaries of the model).
- For the implementation group to get some hands-on experience with programming radio-astronomical applications in C++. Although the group was supposed to arrive in Charlottesville familiar with C++, the experience was largely with small personal "toy" programs.
- To see how well the individual groups could communicate and provide services for one another. This is critical if a distributed project like **aips++** is to succeed.

1.2 What prototype?

The intention of the prototype was to build a system that was "broad and shallow" rather than one that was "narrow but deep". The consensus was that it was most appropriate to attempt in some small way many parts of the ultimate system (also, as a practical matter it is easier to get a lot of people working on a wide problem).

The chosen problem was to apply an antenna based calibration to a UV dataset, image it, and display it using a simple "keyword=value" based command line interface. Details of the prototype are explained more fully in the following chapters and appendices.

1.3 How the prototype was built

The prototype was finished in just under three weeks and consisted of about 15,000 lines of code and comments. We built our prototype on top of the CIC container classes and with the GNU String class.

The team was split up into groups to implement the prototype. The *Raw Data* group[1] was responsible for forming/calibrating and imaging UV data, the *Image* group[2] was responsible for creating image classes and simplified coordinate systems, the *User Interface* group[3] was responsible for creating a parameter passing mechanism, the *Fundamental Libraries* group[4] provided some support with support classes, and the *Organization* group[5] was responsible for creating appropriate makesfiles and directory structures.

The group was fairly enthusiastic about the prototype. In fact, doing it was enough fun so that it was hard to stop work on it when it reached its predefined limits. The initial few days of design were extremely useful, especially the exercises with the CRC (Class - Responsibility - Collaborator) cards. The implementation went pretty well; there was some time wasted getting the ObjectCenter (Saber) programming environment to work properly with the preprocessor that is used by CIC to emulate C++ parametric types. A somewhat surprising result was that the software environment — makefiles, source code checkin systems, etc. — was extremely important, even for this small prototype with people in only two offices.

1.4 Some preliminary conclusions

The user interface group should probably have been given a larger problem, perhaps a simple image arithmetic program. The groups were only moderately successful at communicating with each other. While it is desirable to keep communications costs down, we likely erred too strongly on the side of working independently.

Some flaws in the Green Bank model were found and discussed during the implementation of the prototype. The consensus of some internal email [?] was that some decoupling of entities was required, undoubtedly by associating objects explicitly inside a database. This should also be attractive for end-users of the system.

[1] Dave Shone, Sanjay Bhatnagar, Bob Hjellming, Mark Holdaway, and Bob Sault
[2] Lloyd Higgs, Mark Calabretta, and Brian Glendenning
[3] Friso Olnon and Peter Teuben
[4] Bob Payne and Mark Stupar
[5] Mark Calabretta

Chapter 2

UV data and imaging

2.1 Class descriptions

The problem specified for the UV data and imaging group is essentially as specified for the Green Bank meeting; *i.e.* the manipulation of (for the prototype) visibility data, with particular regard to calibration, and the formation of images from data. Thus the classes which have been defined and implemented closely resemble those described in the Green Bank report, but given the very limited amount of time available for this prototype, the design and implementation of these is often rather different from what we envisage for a full-scale system based on the Green Bank analysis. In particular, the classes do not exhibit the full functionality specified in the Green Bank report, and their relationships have been somewhat simplified. Nevertheless, the overall scheme is basically the same as that decribed in the Green Bank report, although it is by no means a meaningul test of that proposal.

A number of issues have been ignored and/or circumvented. In particular, persistence is neither required nor implemented in the prototype.

The fundamental classes are described below, and the way in which these differ from the corresponding classes in the Green Bank report are indicated. More detailed descriptions of the classes are given in the header files.

YegSet and IntYegSet - These are the representations of the data themselves. The full functionality of YegSet as described in the Green Bank report is not implemented. We have chosen to deal with data in bulk, as YegSets, rather than as individual Yegs. In addition, the selection and sorting operations have not been implemented, although these are still regarded as essential to a real system.

Telescope - As described in the Green bank report, this is essentially a crude means of associating various kinds of objects which are related in some way. In our small prototype, this functionality is largely redundant, since we can maintain the relationships in a "hard-wired" form, and the use of this is not really explored.

Telescope Model - This is a model of all attributes of the observing telescope which are required in calibration/self-calibration of YegSets, together with methods for updating the model *e.g.*, determining gain solutions and applying corrections to the data on

the basis of the new attributes. In the prototype, this models a simple interferometer using only complex gains for each receptor.

ImagingModel/IntImagingModel - This is an imaging model for interferometer data. It performs a Fourier transform on a YegSet to produce an image which represents the sky brightness distribution.

In addition to the principal classes, a number of subsidiary classes have also been developed from an analysis of the individual classes described above. Briefly, these are:

RVector - Real vector;

DVector - Double vector;

CVector - Complex vector;

AssArray - Associative array;

GainTable - Complex table;

Whilst these classes have been developed specifically to serve the requirements of the application-related classes, they are clearly likely to be generally useful. The Vector classes used to implement YegSets, could also be used to implement the GainTable in a way which might be more efficient for operations such as applying complex gains to a YegSet.

2.2 Design issues, problems and lessons

A number of simplifications to the Green Bank model have been mentioned, and in most cases we do not envisage that these would be present in the ultimate design. A number of problems which arise if these simplifications are not made will be discussed shortly, and these will have to be addressed. However, one change which is likely to remain is that we may wish to work with YegSets (rather than individual Yegs) in many most cases. In the prototype, we have assumed that all Yegs in a YegSet are associated with the same Telescope, TelescopeModel and ImagingModel, and we believe it may be more convenient (as well as efficient) to make this assumption. We should still be able to cope with data from multiple telescopes, but we should deal with them as sets of YegSets.

One of the most important problems arose out of design exercise which immediately preceeded the implementation of the prototype. The Green Bank scheme requires many kinds of objects to be associated together, and proposed that this be implemented by using pointers or references in one object to refer to another. It is often the case that such associations may involve classes derived from those for which the pointers are defined, and may require a cast down to the derived type in order to make use of methods which are not present in the base class. This problem appears to be quite common, and it is almost certainly the case that any scheme other than the Green Bank proposal, which has the degree of flexibility we require, will also suffer from this problem. Put simply, relying solely on the static typing of C++ may restrict the kind of dynamic association of different kinds of objects which is likely to be essential.

This presents no problem in the simplified prototype, but must be addressed in a generally extensible system. Whilst it would be inappropriate to discuss solutions to this problem in

this report, we should say that it is not insurmountable, and need not affect the fundamental classes and their interrelationships. The most obvious difference might be the need to maintain associations between objects using some entity external to the objects, rather than by using pointers or references within the objects themselves. This external entity might be a simple table containing "handles" for the objects to be associated, or might be a more complicated database-like system, possibly part of the Project system proposed in the Green Bank report.

The prototyping exercise has confirmed that under some circumstances, encapsulation may be an obstacle to efficiency. For example, if we wish to perform some arbitrary operation on all the elements of a vector or multi-dimensional array, this is best performed with direct access to the array itself. It seems unlikely that it will always be possible to implement such operations as methods of the class, and it might be argued that allowing applications programmers to routinely modify such fundamental classes as vectors and images is worse than introducing methods which present an array of raw data to the application. The latter might be regarded as a violation of encapsulation, but could be performed with some degree of control by the class itself. The appropriate solution to this problem is not clear; there are a number of possibilities, most of which will have important consequences for the ultimate design of our classes.

The analysis and design of high level application-oriented classes leads to new requirements which are best implemented as a lower layer of classes. The Vector and GainTable classes are examples of these; with a little modification, they could serve a variety of requirements from many high-level classes. It is clear that there is considerable scope for reuse of objects at this level. This will require a great deal of interaction between designers of high level classes in order to specify requirements for utility classes which can be used commonly. However, this is likely to result in a class library which is better tailored to the needs of the system, as compared to one which is built out of classes which have not been designed with a particular need in mind, and thus often turn-out to be a poor fit to the requirements.

Chapter 3

Image handling

3.1 Basic approach

The development of classes for image handling for the prototype **aips++** exercise was based on a few basic postulates:

- image data values would be restricted to "float", although it was recognized that many types of images would eventually be required (complex, double, int, ...).
- dimensionality would be restricted to 2, although the code was written so that it could, without a great deal of effort, be expanded to more dimensions.
- rudimentary capability for keeping track of pixel coordinates in an astronomical frame of reference should be built in, but not developed in any depth.
- rudimentary capability for keeping a history attached to an image should be built in, but not developed in depth.
- the prototype classes should demonstrate some capabilities of dynamic binding.
- some attempt should be made to assess the usefulness of the CIC image classes.

In the following, the design of the set of prototype IMAGE classes will be outlined, and some of the member functions for the various classes will be briefly described.

3.2 Types of images

One of the first assumptions made in the image area was that the concept of an Image class would be restricted to sets of data values in which each value has coordinates that can be mapped onto a grid or n-cube, i.e. the coordinates can be represented by integers. Data having random coordinate values therefore do not fall into the class that is defined by "Image". Within the bounds of this definition of "Image", it was apparent that there were needed (at least) three sub-classes of "Image": images in which a full matrix of data values exist, images for which a list of pixel values and coordinates exist, and images that are

defined by an analytical function. In the following, these will be referred to as "FilledImage", "ListImage", and "ModelImage". These will be described in considerable detail after the generic attributes of the base "Image" class are presented.

3.3 Coordinate systems

One of the fundamental attributes of an Image is a coordinate system description that allows the coordinates of any given pixel to be specified (in some user defined representation). For the two-dimensional case treated in the prototype, a basic system of grid coordinates, aligned with the "rows and columns" of an image, and which can be uniquely related to some "physical" system of coordinates, has been assumed. Grid positions in this system are called ImPixel coordinates, with a coordinate specification being given by an ImPixelCoord object. Coordinate values in this system have been treated as "float" values, even though image pixels are located at integral coordinate values. (More will be said about this later). ImPixel coordinates are defined to increase from left to right across an image (first value) and from the bottom to the top of an image (second value).

The class "CoordSys" was devised to specify the relationship between ImPixel coordinates and some physical coordinate system, and also to specify the representation in which the user wishes to have image coordinates expressed. For example, an image may have intrinsic physical coordinates such as (l,m) in an interferometric image, but the user would like to access the image in Galactic coordinates. The coordinate system selected by the user is termed Image coordinates, and coordinates in this system are given by an ImageCoord object.

The attributes for a CoordSys objects, for this exercise, consisted of a set of parameters that can convert ImPixel coordinates to some intrinsic or "native" physical coordinate system, assuming that the two components of the native coordinates are orthogonal and separable (the standard AIPS convention). The characteristics of this native coordinate system are specified in an object of the "CoordSysType" class. For this exercise, this is defined by a name, an epoch, and a set of four parameters. Similarly, the user-specified Image coordinate system is defined by a CoordSysType object. Conversions between the native physical coordinate system and the user's Image coordinate system have not been implemented in the prototype, but the parameters provided in the CoordSysType objects should suffice for such methods.

Thus a CoordSys object allows the following coordinate conversion, and its inverse:

```
ImPixelCoord --> ''Native Coords" --> ImageCoord
```

where the "native" coordinate system is just a useful intermediary.

A third (as defined by separate classes) coordinate system is the internal coordinate system used within a given image. At first glance, it would seem logical to use ImPixel coordinates. However, there are reasons to introduce a separate system, in order to minimize changes required to CoordSys objects as images are manipulated. For example, if an image is defined by a "window" which moves in "astronomical space" and ImPixel coordinates are used for internal image coordinates, the parameters in the attached CoordSys object must be changed for each window location. Similarly, if a new image is created by taking a sub-image (perhaps every n'th pixel within a window area), the derived image must have a new CoordSys object different from that of the parent image. Although all of this is possible, it

seemed simpler to introduce a third coordinate system and one image attribute that links this system to ImPixel coordinates. This limits the proliferation of CoordSys objects. The new system is referred to as Pixel coordinates, and a set of coordinate values is given by a PixelCoord object. For the prototype exercise, Pixel coordinates have their origin (0,0) at the top-left corner of the image and the first value increases from left to right, and the second from top to bottom.

If a sub-image is extracted from an image by selecting every m'th pixel in the x direction and every n'th in the y direction, in the resulting image each increment in Pixel coordinates will no longer correspond to a unit increment in ImPixel coordinates. An image attribute (object of the ImPixStep class) records this relation. Conversions between Pixel coordinates and ImPixel coordinates are the responsibility of Images.

3.4 Image history

Images must carry with them some record of processing history. For the prototype, a class HistFile was implemented, consisting only of a linked list of strings (using CIC classes). Simple methods for listing entries in the files, and for inserting entries, are provided. This is an area that would require much more development for a practical system.

3.5 Image base class and derived classes

The abstract base class Image contains data members which provide linkages to a history file and to a coordinate system, descriptors giving the type of file and data units, and several parameters describing the relationship between internal Pixel coordinates and ImPixel coordinates. Regions of interest (regions within which, and only within which, certain image operations are to be performed) have not been incorporated in detail in the prototype. One region of interest has been included, but in a practical system, a list of regions of interest is probably required. Aside from the parameters that define the Pixel coordinate - ImPixel coordinate relationship, another useful parameter that has been introduced is the image "center". This is user-definable but defaults to Pixel coordinates of (m/2, n/2-1) where the dimensions are [m,n].

The major member functions in the Image class deal with the following functional requirements:

- setting a pixel in the image, and retrieving a given pixel from an image.
- conversions between Pixel coordinates and ImPixel (or Image) coordinates, and vice-versa.
- checking whether one image is "conformant" with another, i.e. an add operation can be performed on them without a chance of adding "apples and oranges".
- adding (with weighting) two images to produce a third, either in the UNION or INTERSECTION sense.
- scaling an image and adding it to another.

- adding entries to the image history, either individual entries or by copying the history file of another image.
- adjusting an image's reference (TLC) position in ImPixel coordinates so that the center of the image falls on a given ImPixel coordinate.
- finding the maximum and minimum pixel values in an image.

The three derived classes of images are "FilledImage", "ListImage" and "ModelImage". The data members of these classes are not generally of interest to the users of the class, so nothing more will be said of them here. (See the header files for details). It should be stated, however, that the CIC class libraries and templates have been used in the prototype to implement linked lists and arrays. The major member functions of these classes implement the following functionality (in addition to that presented by the base Image class).

FilledImage Class

- display image as a gray-scale using PGPLOT.
- provide access to the data storage for efficient mathematical operations. Although this violates encapsulation to some extent, it may well be required in any realistic system.
- allow a sub-image to be extracted from an image, creating a new image, where there is considerable flexibility in the selection of the sub-image, i.e. every i'th pixel in x and every j'th pixel in y. It is this operation which demonstrates the utility of having both Pixel and ImPixel coordinates, since the choice of i and j completely restructures the relationship of Pixel coordinates and ImPixel coordinates, but the extracted sub-image and its parent can and do still have the identical CoordSys object.

ListImage Class

- be able to "clone" itself, copying all attributes but zeroing the list of pixels. This functionality might be required for FilledImages also but was only implemented as a test for ListImages.
- various methods of adding or retrieving pixels in the list (by matching Pixel coordinates, by serial number in the list, etc.). Special methods were introduced to make use of the list iterator provided by CIC.
- the ability for the "dimensions" of a ListImage to grow as new pixels, with new Pixel coordinates, are added to the list.
- the ability to merge pixels in the list which may have the same Pixel coordinates (useful for a list of CLEAN components). In a real system, this might be accompanied by a sort operation.

ModelImage Class

- to allow the flexible specification of an analytical model of an image.
- to provide a flexible means of updating the parameters of the model.

3.6 Image operations using dynamic binding

Some experience in applications of dynamic binding has been obtained. Aside from the general pixel access routines, which are implemented as virtual base class functions, the combined and scaled-add methods have been the best examples of this. A statement of the type:

```
Image C = 5 * Image A - 7 * Image B  (logical code)
```

works correctly regardless of the image types (except that C cannot be a ModelImage). The prototype has given some insight into the practicality (not always) of such methods, and the requirements for implementing them.

3.7 What has been learned from the prototype

The prototype has provided a lot of experience in using C++ and C++ tools, but has also, we feel, indicated that the general framework adopted for image handling is probably not too different from what one would like for a practical system. The use of Pixel and ImPixel coordinate systems has raised some questions of overhead (and possible user confusion) but has provided great flexibility. Part of the overhead arises because Pixel coordinates have two aspects: as indexes in image data arrays (where they must be integers), and as computed counterparts of Image (or ImPixel) coordinates (where they must usually be floating numbers). A practical system may have to introduce a better way of meeting both these requirements, if it can be done with lower overhead. Certainly, several methods that now take Pixel coordinate arguments or return Pixel coordinates will have to be overloaded to also take/return ImPixel coordinates. It is possible that an ImPixel class (data value plus ImPixel coordinates) will have to join the current Pixel class. If the dual-coordinate system is to be used successfully, the client must be able to perform all image operations without ever bothering with Pixel coordinates! The prototype has not been overly successful in testing the usefulness of the CIC image classes, but only because multi-dimensional CIC arrays of floating numbers, and associated display methods, are unavailable at this moment.

3.8 What was missing from the prototype

The prototype lacks dimensionality greater than two, generality with regard to types of pixels, and no capabilities in the areas of regions of interest and image errror data, amongst other things. The latter of these needs careful analysis before the optimum implementation can be designed.

Chapter 4

User Interface

During the prototype stage a basic command line user interface was build, with which tasks have been constructed. Some work was spend in showing that both the AIPS interpreter and the Graphical User Interface (GUI) are plug-in compatible user interfaces. For example, a functional GUI for Khoros[1] is available for demo purposes. The AIPS shell interpreter can be thought of in terms of the Miriad[2] shell interpreter.

4.1 Astronomers vs. Programmers

The basic (command line) user interface is a series of "*keyword=value*" pairs, which we call **program parameters**[3].

The `class Param` (see `Param.h`) implements one single such **parameter**. In addition to a name and a value, a parameter has a variety of other attributes, such as a one-line help string (useful when being prompted etc.), a type, a range and optional units. All of these are character strings; parsing and error checking is done at a different level. The programmer however will never interact with a parameter through it's class interface. This is done with the `class Input`, which is some kind of container of `Param`'s, with a variety of user interface attributes (help-level, message/debug-level etc).

Although the programmer must supply the user interface with a number of predefined **program parameters**, the user interface itself will create a small number of **system parameters** (help=, debug=). The purpose of these is to tell the task how to communicate with the user and it's environment, and give the user control over these items. For example, the user may want to be prompted, with error recovery, and see (debug) messages above a certain threshold level.

For the benefit of the Programmer, the user interface also defines a number of standard parameters ("templates"), which can be copied and bound to a program parameter.

Parameter names are to be found by minimum match, if so requested by the user.

Most programs are probably happy with a simple set of parameters, like a linear list. We

[1] (c) University of New Mexico
[2] (c) BIMA
[3] The name **parameter** and **keyword** are sometimes used both

have discussed hierarchical keywords and in Section ?? a few thoughts are expressed.

All input as well as output is controlled by the user interface. The Astronomer has a varying degree of control over how and where input and output occurs. In the command line interface system control occurs through a small number of system parameters, which can be preset by environment variables, supplied as if they were parameters on the command line, or both.

For example, a interactive UNIX shell session may look like:

```
1% setenv DEBUG 1
2% setenv HELP prompt,aipsenv
3% prog key1=val1 key3=val3
4% prog val1 val2 key4=val4 key5=val5 debug=0
5% unsetenv HELP DEBUG
6% prog help=pane > prog.pane
```

After having preset the DEBUG and HELP modes in commands **1%** and **2%**, commands **3%** and **4%** will act accordingly: the user is prompted, and parameter default values are restored and saved from an AIPS environment file before and after invocation. In addition, in command **4%** the user decided not to see any messages. Command **6%** gives an example of the self-describing mode of programs, where a pane description file for Khoros has been constructed.

4.2 Programmers: Where is my `main`?

No, we don't want you to use `main(int argc, char **argv)` anywhere in your code. Instead, use `aips_input()`, `aips_main()` and `aips_output()`.

To summary, your section of code could then look something like:

```
//aips++
//  Hypothetical Silly Interactive Contour Plotter
//

#include <Main.h>        // Standard declarations needed for an AIPS++ main program
#include <SillyImage.h>

aips_input(Input &inputs) // Definition of the allowed Program Parameters
{
  inputs.Version("19-mar-92 PJT");
  inputs.Usage("Hypothetical Silly Interactive Contour Plotter");

  inputs.Create(              "in",      "",      "Input file",        "InFile",  "r!");
  inputs.Create(              "levels",  "",      "Contour levels",    "RealArray");
  inputs.StdCreate("lstyle",  "lstyle",  "solid", "My Contour line type");
  inputs.StdCreate("lwidth");
  inputs.Create(              "annotate","full",  "What annotation?",  "String",  "full|brief|none|publica
  inputs.StdCreate("device");
}

aips_main(Input &inputs) // Computation box - this could be spawned to various machines
{
    String    dname     = inputs.GetString("device");
    Device device(dname);

    do {

        File      f         = inputs.GetFile("in");
        RealArray contours  = inputs.GetRealArray("levels");
        String    lstyle    = inputs.GetString("lstyle");
        Int       lwidth    = inputs.GetInt("lwidth");

        contours.Sort();       // Make sure this array is sorted

        if(contours.Count() > 20) cwarning << "A lot of contours buddy\n"
        if(countour.Count() == 0) break;

        cdebug.Level(1);
        cdebug << "Plotting " << contours.count() << " contours\n"
               << Level(2) << contours << "\n";

        SillyImageContour(f.name(),contours.Count(),contours.Value(),
                          lstyle, lwidth, dname);

    } while (inputs.More());

    device.Close();
}
```

Comments:

- In **aips_input**, the **program parameters** are defined through the **Create** member function. In addition, a **Version** and **Usage** string should be supplied to the user interface.
- The **aips_input** routine could be automatally made by a code generator from a description section encoded in the source code of the program itself, much like Mark Calabretta's proposal discussed last fall. The advantage of this is that we can generate more elaborate online context and level dependant help. It should not be too hard to create readable documents in page description languages like man, latex or texinfo. The Andrew Toolkit, which has been considered too, is a different story.
- A number of standard **ostream**'s (**cwarning**, **cerror** and **cdebug**) are to be provided for[4], acting much like **cerr**; they handle warning messages, fatal error messages and a (Astronomer controlable message level) debug output. After a fatal error the program will exit gracefully. A specified number of fatal errors can be overridden by a system parameter (error=). The Programmer can also define a cleanup function, say **aips_cleanup**, which is called before the program really quits. Even a recover function could be supplied with which Programmers can recover from a known localized fatal error.
- Alternatively, variable argument (**<stdarg.h>**) versions of the above output could be made available under the names **error**, **warning** and **debug**:

```
#include <stdarg.h>

void error(char *fmt ...);
void warning(char *fmt ...);
void debug(int level, char *fmt ...);
```

- The **aips_main** function acts as a replacement for where C/C++ programmers commonly define their **main**. A true **main(int argc, char **argv)** is present in the AIPS library (See **Main.C**), and gets automatically linked in when you **#include <Main.h>**.
- An **Output** object has not been defined yet.
-

[4]Not present in this prototype

4.3 Heirarchical parameters

A hierarchical parameter would be set using the format

```
key.class1.class2.class3=value
```

(e.g. "*xaxis.grid.style=dotted*") we will use a notation where the hierarchical level is given by a the appropriate number of dots that the keyname starts with. To start with an example, a somewhat elaborate program which would clearly benefit from hierarchical keywords

```
<Key>          <StdKey>      <Type>      <Range>
====           ========      ======      =======

in             infile        InFile      r|w|w!|rw|.....
.region        xyzselect     String
contour                      bool        t|f
.levels                      RealArray   sort($0,$N)
.style         lstyle        String      solid|dotted|dashed|....
.thickness     lwidth        int         0:5
.color         color         String      cyan|red|green|0x134|....
greyscale                    bool        t|f
.minmax                      Real[2]     $1<$2
.gamma                       Real        >=0
.invert                      bool        t|f
.colormap      colormap      InFile      bw|rainbow|..
xaxis
.ticks                       Real[2]
.grid                        Real
..style        lstyle
..thickness    lwidth
.label                       String
..font         font          InFile      (calcomp|helvetica|roman)(10,12,15,20)
yaxis
.ticks                       Real[2]
.grid                        Real
..style        lstyle
..thickness    lwidth
.label                       String
..font         font          InFile      (helvetica|roman)(10,12,15,20)
annotate                     String      none|brief|full|publication
```

Comments/Problems:

- The order in which keywords are "created"[5] is still important, not only to properly define their hierarchy, but foremost to allow shortcuts with nameless specification of parameters on the command line. E.g. "`ccdplot ngc1365u 'box(10,10,20,20)' t 10:20:2 grey=t ann=full`" would be interpreted as `in=ngc1565u` etc. Obviously once a parameter was named, all subsequent ones need to be too (assuming the command line is parsed left to right).
- `Range` must contain a boolean expression, where `$0` is the name of an array, `$N` the number of elements, `$1`, `$2`, `$3`, ... `$($N)` the array elements, `&` and `|` the boolean operators, : to denote an implied do-loop (with optional second : followed by the stride). A fairly rich syntax will be made available.

[5] See `Input::Create()`

- `File` could be the same as a `String` but could also be usefull class (InFile and OutFile) in itself, with name, file pointer? and appropriate wildcard expansion of the string into the full filename.
- `xaxis,yaxis`: these two keywords are clearly related. In prompt mode it would be annoying if the Astronomer sat through the whole `xaxis` family, and then wants to do the `yaxis` tree with the defaults now inherited from the `xaxis` tree. (perhaps only the label name would be different (though the most appropriate default would be the one from the image header, if available). The programmer must leave the defaults in `yaxis` blank, and take the `xaxis` equivalent if none supplied in the `yaxis` equivalent.

4.4 Terminology/Glossary

program Executable within the Unix environment, that has the AIPS user interface.

task – same as above?

parameter Has a name, value, help and all that other good stuff. They come as **program parameters** and **system parameters**, though a third kind, the **standard parameters**[6] are internally defined by the user interface. Programmers can bind **standard parameters** to `program parameters` at compile time.

keyword The name of a parameter.

default The value of a parameter as defined by `aips_input`, though possibly overriden by previous settings of the Astronomer if the user interface was told to (aipsenv file, commandline)

[6] The name **template parameters** is perhaps more appropriate, but confusing in the C++ environment

Chapter 5

System management

With the dissemination of two papers, *System management for* **aips++** - *Part 1: organization and distribution* and *Part 2: activation, generation, and verification* the design of **aips++** system management is now well advanced. The final part of the trilogy *Part 3: networking* is scheduled for release by Jun/30, and will cover the area of network services.

Implementation of the system design has been driven by necessity. Creation of the empty **aips++** directory tree was a trivial operation, belying a great deal of thought which had been put into its design.

Code management has, in the first instance, been implemented by using *RCS*. Each of the code directories has an *RCS* repository attached to it. Plain-text copies of the code are kept in the code area itself. Programmers can create their own "shadow" representation of the **aips++** code directory tree by using the *mktree* utility which creates symbolic links to the `~aips++` *RCS* directories. Programmers then appear to have their own private workspace with a window into the master *RCS* repositories, and can check code in and out of the *RCS* repository as though it were their own. This mechanism has served us extremely well.

A generic GNU makefile works together with the *RCS* mechanism described above to compile classes, class test programs, and the kernel library, and also has several other functions. With a dozen programmers contributing 1000 lines of code per day on average, the system has grown in complexity at an accelerating rate, and the makefile is now proving to be indispensible. The makefile allows programmers to compile code without having to check it out of *RCS* and thereby minimizes the number of files that need to be present in their private workspace, with the consequent possibility that these may be "stale". It uses the search path mechanisms which are part of GNU *make*, searching for files first in the programmer's own directory, then in the standard **aips++** directories. However, although the makefile is logically correct, it is not particularly efficient in shirking unnecessary work. In particular, it recompiles a class implementation file if any header file has changed. A later generation should be able to do better.

aips++ programmers may now define their **aips++** "environment" by means of an `aipsinit.[c]sh` script. This redefines the `PATH` (and `MANPATH`) environment variables, appending the **aips++** binary (and man page) directories to it. It also defines a single environment variable, `AIPSPATH`, which contains five space-separated character strings which define the root of the **aips++** directory tree, the host architecture, the **aips++** version, the local site name, and

the local host name. This information is fundamental and must be known in order to access the **aipsrc** databases.

The **aipsrc** databases have been implemented via a **C**-program called *getrc*. It looks for device and other definitions in a format similar to that of the *.Xdefaults* database used by **X-windows**. In resolving a reference it searches the following sequence of **aipsrc** files:

```
~/.aipsrc
$AIPS/$ARCH/$VERS/$SITE/$HOST/aipsrc
$AIPS/$ARCH/$VERS/$SITE/aipsrc
$AIPS/$ARCH/$VERS//aipsrc
```

The last of these files contains default values, and the other three allow these to be overridden on a user-, host-, and site-specific basis.

The first use to which the **aipsrc** mechanism has been put is that of a simple, and easily configurable set of printer utilities, including a utility to print class header and implementation files in a compact and convenient form.

Appendix A

Public interfaces

A.1 CoordSys.h

HEADER FILE DESCRIPTION

Header file for CoordSys class

ENVIRONMENT

```
#define COORDSYS_H

#include <iostream.h>
#include "ImPixelCoord.h"
#include "ImageCoord.h"
#include "CoordSysType.h"
```

CLASS DESCRIPTION

Class: CoordSys This class defines the coordinate system associated with an image. The coordinate system specification consists of two parts. The first relates ImPixelCoords to a "native" image coordinate system. For example, the "native " coordinate system for a VLA observed image would be l and m in the SIN system. The second part of the specification defines the coordinate system which the user wants to use for ImageCoords. For example, the user may want to use Galactic coordinates as ImageCoords.

CLASS SUMMARY

```
class CoordSys
{
public:
        void SetRefPix(ImPixelCoord);
        ImPixelCoord GetRefPix() const;
        void SetRefCoord(ImageCoord ic);
        ImageCoord GetRefCoord() const;
        void SetDeltaCoord(ImageCoord ic);
        ImageCoord GetDeltaCoord() const;
        void SetRotAng(double);
        double GetRotAng() const;
        void SetNtvCrd(CoordSysType ct);
        CoordSysType GetNtvCrd() const;
        void SetImageCrd(CoordSysType ct);
        CoordSysType GetImageCrd() const;
        ImageCoord GetImageCoord(ImPixelCoord) const;
        ImPixelCoord GetImPixelCoord(ImageCoord) const;
        CoordSys();
        CoordSys(ImPixelCoord rp, ImageCoord rc, ImageCoord d, double a,
                CoordSysType n, CoordSysType i);
        ~CoordSys();
```

```
};
ostream& operator << (ostream& os, const CoordSys& cs);
```

MEMBER FUNCTIONS

Set the coordinates of the reference ImPixel for coordinate system transformations to native ImageCoords.

```
void SetRefPix(ImPixelCoord);
```

Return the reference ImPixel for coordinate transformations

```
ImPixelCoord GetRefPix() const;
```

Set the reference native ImageCoord value at the reference pixel (presumably a value in radians for sky coordinates)

```
void SetRefCoord(ImageCoord ic);
```

Return the reference native ImageCoord value at the reference pixel (presumably a value in radians for sky coordinates)

```
ImageCoord GetRefCoord() const;
```

Set the increment native ImageCoord value, per ImPixelCoord increment (presumably a value in radians for sky coordinates)

```
void SetDeltaCoord(ImageCoord ic);
```

Return the increment native ImageCoord value, per ImPixelCoord increment (presumably a value in radians for sky coordinates)

```
ImageCoord GetDeltaCoord() const;
```

Set the rotation angle (radians)

```
void SetRotAng(double);
```

Get the rotation angle (radians)

```
double GetRotAng() const;
```

Set the type of the native coordinate system (SIN, NCP, etc.)

```
void SetNtvCrd(CoordSysType ct);
```

Return the type of the native coordinate system (SIN, NCP, etc.)

```
CoordSysType GetNtvCrd() const;
```

Set the type of the desired Image Coordinate system (SIN, NCP, etc.)

```
void SetImageCrd(CoordSysType ct);
```

Return the type of the Image Coordinate system (SIN, NCP, etc.)

```
CoordSysType GetImageCrd() const;
```

Convert ImPixel Coordinates to Image Coordinates (converting from system "ntvcrd" to "imagecrd").

```
ImageCoord GetImageCoord(ImPixelCoord) const;
```

Convert Image Coordinates to ImPixel Coordinates (converting from system "imagecrd" to "ntvcrd" in the process).

```
ImPixelCoord GetImPixelCoord(ImageCoord) const;

// constructors and destructor
CoordSys();
CoordSys(ImPixelCoord rp, ImageCoord rc, ImageCoord d, double a,
        CoordSysType n, CoordSysType i);
~CoordSys();
```

NON-MEMBER FUNCTIONS

```
ostream& operator << (ostream& os, const CoordSys& cs);
```

A.2 CoordSysType.h

HEADER FILE DESCRIPTION

Header file for CoordSysType class

ENVIRONMENT

```
#define COORDSYSTYPE_H

#include <iostream.h>
#include <K_String.h>
```

CLASS DESCRIPTION

Class: CoordSysType This class defines the type of a coordinate system (e.g. SIN, NCP, RaDec, etc.), and associated parameters. Currently the type is defined by an arbitrary character string, but in future might be defined as one of a number of pre-determined descriptors.

CLASS SUMMARY

```
class CoordSysType
{
public:
        void SetCoordSysID(K_String& );
        K_String GetCoordSysID() const;
        void SetEpoch(float);
        float GetEpoch() const;
        void SetParam(int, double);
        double GetParam(int) const;
        CoordSysType();
        CoordSysType(K_String , float = 2000, double = 0.0, double = 0.0,
                double = 0.0, double = 0.0);
        ~CoordSysType();
};
ostream& operator << (ostream& os, const CoordSysType& cst);
int operator == (const CoordSysType& ct1, const CoordSysType& ct2);
```

MEMBER FUNCTIONS

Set the coordinate-system descriptor

```
        void SetCoordSysID(K_String& );
```

Return the coordinate-system descriptor

```
K_String GetCoordSysID() const;
```

Set the epoch

```
void SetEpoch(float);
```

Get the epoch

```
float GetEpoch() const;
```

Set parameter i (0 ¡= i ¡= 3)

```
void SetParam(int, double);
```

Get parameter i (0 ¡= i ¡= 3)

```
double GetParam(int) const;
```

```
CoordSysType();
CoordSysType(K_String , float = 2000, double = 0.0, double = 0.0,
        double = 0.0, double = 0.0);
~CoordSysType();
```

NON-MEMBER FUNCTIONS

Print out a CoordSysType object

```
ostream& operator << (ostream& os, const CoordSysType& cst);
```

Compare two CoordSysTypes for equality

```
int operator == (const CoordSysType& ct1, const CoordSysType& ct2);
```

A.3 FilledImage.h

HEADER FILE DESCRIPTION

This file contains the interface to the FilledImage class.

ENVIRONMENT

```
#define FILLED_IMAGE_H

#include "Array2d.h"
#include "Image.h"
#include <assert.h>
#include <iostream.h>
#include "ListImage.h"
#include "ModelImage.h"

DECLARE_ONCE Array2d<float>;
DECLARE_ONCE Array2d<double>;

class ModelImage;
class ListImage;
```

CLASS DESCRIPTION

The FilledImage class is a two dimensional image (of float) class with access via the various coordinate type.

CLASS SUMMARY

```
class FilledImage  : public Image
{
public:
    FilledImage();
    FilledImage(int m, int n, float v=0.0);
    FilledImage(const FilledImage& src);
    FilledImage(const ListImage& src, float);
    FilledImage(const ModelImage& src);
    FilledImage &operator=(const FilledImage& src);
    FilledImage &operator=(const ModelImage& src);
    int GetNumEl() const;
    Array2d<double> GetStorage();
    void SetStorage(const Array2d<double> &);
    void Scale(float);
    void Fill(float);
    int Extrema(Pixel &maxpix, Pixel &minpix) const;
    void SetPixel(PixelCoord, float);
```

```
    float &operator() (int,int);
    float GetPixel(PixelCoord) const;
    int GetImPixelVal(const ImPixelCoord&, float&) const;
    virtual int UCombine (float, const Image&, float, const Image&, float);
    virtual int XCombine (float, const Image&, float, const Image&, float);
    virtual int ScaledAdd (float, const Image&);
    FilledImage SubImage(PixelCoord &, ImPixStep &, ImageDim &) const;
    void Display();
    void Write(Char *File);
    ~FilledImage();
    friend ostream &operator<<(ostream &os, const FilledImage &im);
};
```

MEMBER FUNCTIONS

Default constructor makes 0 sized image.

```
    FilledImage();
```

Make a FilledImage of a given size (possibly with a set value)

```
    FilledImage(int m, int n, float v=0.0);
```

Copy constructors

```
    FilledImage(const FilledImage& src);
```

Copy constructor (from ListImage) - needs "fill" value

```
    FilledImage(const ListImage& src, float);
```

Copy constructor (from ModelImage)

```
    FilledImage(const ModelImage& src);
```

Assignment operators

```
    FilledImage &operator=(const FilledImage& src);
```

The following will fail if the existing FilledImage does not have enough storage allocated!!

```
    FilledImage &operator=(const ModelImage& src);
```

Accessors

```
    int GetNumEl() const;
```

Put all the values in memory, which this function allocates

```
Array2d<double> GetStorage();
```

Fill the image with the values in memory, optionally delete that memory

```
void SetStorage(const Array2d<double> &);
```

Data operations.

```
void Scale(float);
void Fill(float);
int Extrema(Pixel &maxpix, Pixel &minpix) const;
void SetPixel(PixelCoord, float);
```

Set/get pixel values by direct indexing

```
float &operator() (int,int);
```

If PixelCoord is non-integral or lies outside image range, method exits.

```
float GetPixel(PixelCoord) const;
```

The following returns FALSE if corresponding PixelCoord is non- integral or lies outside image.

```
int GetImPixelVal(const ImPixelCoord&, float&) const;
```

Union/Intersection operators

```
virtual int UCombine (float, const Image&, float, const Image&, float);
virtual int XCombine (float, const Image&, float, const Image&, float);
virtual int ScaledAdd (float, const Image&);
```

Extract sub-image with the TLC specified (as a Pixel coordinate), the step in pixels specified (by an ImPixStep object), and desired dimension specified. Exits if no image is possible (e.g. TLC doesn't lie within the parent image). Desired dimension will shrink to get maximum sub-image allowed.

```
FilledImage SubImage(PixelCoord &, ImPixStep &, ImageDim &) const;
```

Display Function (uses PGPLOT to present a grey-scale representation of the image)

```
void Display();
```

Write Function (ASCII)

```
void Write(Char *File);
```

Destructor.

```
~FilledImage();
```

Output an image's attributes and contents. Note that a CoordSys must be defined.

```
friend ostream &operator<<(ostream &os, const FilledImage &im);
```

```
        double mydecl=50.0);     // Source declination (degrees).
```

Return the Telescope object associated with this observation.

```
    Telescope& Filler::GetTelescope();
```

A.5 GainTable.h

HEADER FILE DESCRIPTION

This file contains definitions for the GainTable class.

ENVIRONMENT

```
#define GAINTABLE_H

#include <assert.h>
#include <math.h>
#include <complex.h>
```

CLASS DESCRIPTION

GainTable class

CLASS SUMMARY

```
class GainTable
{
public:
    GainTable(int, int);
    ~GainTable();
    complex& Gain(int, int);
    void SetGain(int, int, complex);
    int Nants();
    int Nentries();
};
```

MEMBER FUNCTIONS

Constructor for GainTable specifying: number of receptors; number of entries;

```
    GainTable(int, int);
```

Destructor.

```
    ~GainTable();
```

Return a reference to a gain field in the table specifying: entry/tuple number; receptor number;

```
    complex& Gain(int, int);
```

Set a Gain value specifying entry/tuple number; receptor number; value

```
void SetGain(int, int, complex);
```

Return number of collumns (NANTS)

```
int Nants();
```

Return number of rows (Entries)

```
int Nentries();
```

A.6 GridTool.h

HEADER FILE DESCRIPTION

ENVIRONMENT

```
#define GRIDTOOL_H

#include <IntYegSet.h>
#include <DummyImage.h>

typedef double  DFunction(double);
double SincExp(double);
```

CLASS DESCRIPTION

CLASS SUMMARY

```
class GridTool
{
public:
   GridTool(DFunction *CF=SincExp,int w=1,int n=6);
   ~GridTool();
   void SetConvFunc(DFunction *CF);
   void SetWeightingScheme(int w);
   void SetSupportSize(int i);
   virtual int Resample(const IntYegSet &, DummyImage &);
   virtual int Resample(const DummyImage &, IntYegSet &);
   virtual int Resample(const IntYegSet &, DummyImage &, DummyImage &);
};
```

MEMBER FUNCTIONS

Constructor

```
   GridTool(DFunction *CF=SincExp,int w=1,int n=6);
```

Destructor

```
   ~GridTool();
```

Services

```
   void SetConvFunc(DFunction *CF);
   void SetWeightingScheme(int w);
   void SetSupportSize(int i);
```

The Gridding and De-gridding services. The approprate algorithm is invoked by the ordering of the argumensts.

```
Resample(IntYegSet, DummyImage) ==> griddiding from VIS -> Image
Resample(DummyImage, IntYegSet) ==> de-gridding from Image -> VIS

  virtual int Resample(const IntYegSet &, DummyImage &);
  virtual int Resample(const DummyImage &, IntYegSet &);
```

Gives grid for PSF as well

```
  virtual int Resample(const IntYegSet &, DummyImage &, DummyImage &);
```

A.7 HistFile.h

HEADER FILE DESCRIPTION

Header file for HistFile class

ENVIRONMENT

```
#define HISTFILE_H

#include "K_DList.h"
#include "K_String.h"
#include <iostream.h>

DECLARE_ONCE K_DList<K_String>;
```

CLASS DESCRIPTION

Class HistFile defines a linked list of CIC strings, which holds an account of the history associated with an image.

CLASS SUMMARY

```
class HistFile
{
public:
        HistFile();
        ~HistFile();
        HistFile(const HistFile& src);
        HistFile& operator = (const HistFile &);
        void AddRecord(K_String);
        void AddFile(const HistFile &);
        int NumRecs();
};
ostream& operator << (ostream&, HistFile&);
```

MEMBER FUNCTIONS

Constructor and desctructor

```
        HistFile();
        ~HistFile();
```

Copy constructor

```
        HistFile(const HistFile& src);
```

Assignment operator

```
HistFile& operator = (const HistFile &);
```

Add a record

```
void AddRecord(K_String);
```

Add the contents of another history file

```
void AddFile(const HistFile &);
```

Get number of records

```
int NumRecs();
```

Print on given stream records i to j. The value of i must be positive and i must be ¡ j. If no specified records exist, nothing is output!

NON-MEMBER FUNCTIONS

Print out a HistFile object

```
ostream& operator << (ostream&, HistFile&);
```

A.8 ImPixStep.h

HEADER FILE DESCRIPTION

ENVIRONMENT

```
#define IMPIXSTEP_H

#include <iostream.h>
#include <K_Array.h>
#include <assert.h>
#include "PixelCoord.h"
#include "ImPixelCoord.h"

DECLARE_ONCE K_Array<int>;    // For the TI preprocessor
```

CLASS DESCRIPTION

Number of ImPixel's per pixel. Important for derivation of sub-images from an image. For a system where ImPixels increase left to right, and bottom to top; and a system such as CIC where image pixels increase left to right and top to bottom, ImPixStep is + in the x coordinate and - in the y coordinate. ImPixStep is also a useful object for integer vector operations involving Pixel or ImPixel coordinates.

INVARIANT

Pixel steps must be non zero, although both positive and negative are acceptable.

CLASS SUMMARY

```
class ImPixStep
{
public:
    ImPixStep();
    ImPixStep(int dx,int dy);
    ImPixStep(const K_Array<int> &);
    virtual ~ImPixStep();
    virtual int Ok() const;
    void SetImPixStep(int i, int m);
    void SetImPixStep(const K_Array<int> &);
    int GetImPixStep(int i) const;
    K_Array<int> GetImPixSet() const;
};
ImPixelCoord operator * (const PixelCoord &, const ImPixStep &);
ImPixelCoord operator * (const ImPixStep &, const PixelCoord &);
PixelCoord operator / (const ImPixelCoord&, const ImPixStep&);
ImPixStep operator * (const ImPixStep&, const ImPixStep&);
int operator == (const ImPixStep&, const ImPixStep& );
ostream& operator<< (ostream&, ImPixStep);
```

MEMBER FUNCTIONS

Default constructor sets to steps of 1

```
ImPixStep();
```

Construct with given steps (non-zero)

```
ImPixStep(int dx,int dy);
```

Construct with given steps given in an array (non-zero)

```
ImPixStep(const K_Array<int> &);
```

Destructor does nothing presently

```
virtual ~ImPixStep();
```

Returns one if the state is acceptable (non-zero steps)

```
virtual int Ok() const;
```

Set's the ith step to m (must be non-zero)

```
void SetImPixStep(int i, int m);
```

Sets the steps to the vector elements (must be non zero and the correct size.

```
void SetImPixStep(const K_Array<int> &);
```

Get the ith step

```
int GetImPixStep(int i) const;
```

Get all steps into a vector;

```
K_Array<int> GetImPixSet() const;
```

NON-MEMBER FUNCTIONS

Arithmetic operators

```
ImPixelCoord operator * (const PixelCoord &, const ImPixStep &);
ImPixelCoord operator * (const ImPixStep &, const PixelCoord &);
PixelCoord operator / (const ImPixelCoord&, const ImPixStep&);
ImPixStep operator * (const ImPixStep&, const ImPixStep&);
```

Compare two ImPixSteps for equality

```
int operator == (const ImPixStep&, const ImPixStep& );
```

Print a PixelCoord

```
ostream& operator<< (ostream&, ImPixStep);
```

A.9 ImPixelCoord.h

HEADER FILE DESCRIPTION

Header file for ImPixelCoord class

ENVIRONMENT

```
#define A_IMPIXELCOORD_H

#include <assert.h>
#include <iostream.h>
#include "ImageCoord.h"
```

CLASS DESCRIPTION

This class defines the n-dimensional (currently 2-dimensional) coordinates of an image, in the pixel frame of reference used by a CoordSys object. The x axis increases left to right and the y axis from bottom to top. Although usually only integral coordinate values are used with reference to an image, they are stored as floats for generality. (Coordinate conversions from astronomical coordinates can return non-integral values!)

CLASS SUMMARY

```
class ImPixelCoord
{
public:
        void SetImPixCoord(int i, float val);
        void SetImPixCoord(float*);
        float GetImPixCoord(int i) const;
        float* GetImPixCoord() const;
        ImPixelCoord operator += (const ImPixelCoord &);
        ImPixelCoord operator -= (const ImPixelCoord &);
        ImPixelCoord();
        ImPixelCoord(float i, float j);
        ~ImPixelCoord();
};
ostream& operator << (ostream& os, const ImPixelCoord ic);
ImPixelCoord operator + (const ImPixelCoord&, const ImPixelCoord&);
ImPixelCoord operator - (const ImPixelCoord&, const ImPixelCoord&);
ImPixelCoord operator * (const ImPixelCoord&, const float);
ImPixelCoord operator * (const float, const ImPixelCoord&);
ImageCoord operator * (const ImPixelCoord&, const ImageCoord&);
ImageCoord operator * (const ImageCoord&, const ImPixelCoord&);
ImPixelCoord operator / (const ImageCoord&, const ImageCoord&);
```

MEMBER FUNCTIONS

Set the i-axis coordinate to value: val. The value of i must be greater than 0 and less than 3 (currently)

```
Preconditions:
          valid axis: 0 < i  < 3

Postconditions:
          invalid input parameters: assertion exit

  void SetImPixCoord(int i, float val);
```

Set the coordinate values to the array of float values given by the float pointer.

```
Preconditions:
          valid number of values: 2

Postconditions:
          invalid input parameters: assertion exit

  void SetImPixCoord(float*);
```

Get the value of the i'th axis coordinate. The value i must be greater than 0 and less than 2.

```
Preconditions:<
          valid axis: 0 < i  < 3

Postconditions:
          true

  float GetImPixCoord(int i) const;
```

Return the coordinate values by a pointer

```
Preconditions:
          true

Postconditions:
          true

  float* GetImPixCoord() const;
```

Add an ImPixelCoord

```
  ImPixelCoord operator += (const ImPixelCoord &);
```

Subtract an ImPixelCoord

```
ImPixelCoord operator -= (const ImPixelCoord &);


ImPixelCoord();
ImPixelCoord(float i, float j);
~ImPixelCoord();
```

NON-MEMBER FUNCTIONS

Print an ImPixelCoord

```
ostream& operator << (ostream& os, const ImPixelCoord ic);
```

Add two ImPixelCoords

```
ImPixelCoord operator + (const ImPixelCoord&, const ImPixelCoord&);
```

Subtract two ImPixelCoords

```
ImPixelCoord operator - (const ImPixelCoord&, const ImPixelCoord&);
```

Multiply an ImPixelCoord by a float

```
ImPixelCoord operator * (const ImPixelCoord&, const float);
```

Multiply a float by an ImPixelCoord

```
ImPixelCoord operator * (const float, const ImPixelCoord&);
```

Multiply an ImPixelCoord by a ImageCoord

```
ImageCoord operator * (const ImPixelCoord&, const ImageCoord&);
```

Multiply a ImageCoord by an ImPixelCoord

```
ImageCoord operator * (const ImageCoord&, const ImPixelCoord&);
```

Divide two ImageCoords (a coordinate value by a coordinate increment)

```
ImPixelCoord operator / (const ImageCoord&, const ImageCoord&);
```

A.10 Image.h

HEADER FILE DESCRIPTION

aips++ header

CLASS DESCRIPTION

Abstract base class which stores an image.

ENVIRONMENT

```
#define A_IMAGE_H

#include <assert.h>
#include <iostream.h>
#include "K_String.h"
#include "CoordSys.h"
#include "ImPixelCoord.h"
#include "PixelCoord.h"
#include "ImageUnits.h"
#include "ImPixStep.h"
#include "HistFile.h"
#include "ImageDim.h"
#include "Pixel.h"
```

CLASS SUMMARY

```
class Image
{
public:
   Image(int m =0, int n = 0);
   ImageDim GetDim() const;
   virtual int GetNumEl() const = 0;
   K_String GetImageType() const;
   void SetImageType(K_String s);
   void SetCenPix(PixelCoord p);
   PixelCoord GetCenPix() const;
   void SetImCornPix(ImPixelCoord p);
   ImPixelCoord GetImCornPix() const;
   void SetPixStep(ImPixStep r);
   ImPixStep GetPixStep() const;
   void SetDataType(ImageUnits iu);
   ImageUnits GetDataType() const;
   void SetCoordSys(CoordSys * pcs);
   CoordSys * GetCoordSys() const;
```

```
   void SetRegBLC(ImPixelCoord blc);
   ImPixelCoord GetRegBLC() const;
   void SetRegTRC(ImPixelCoord trc);
   ImPixelCoord GetRegTRC() const;
   HistFile * GetHistPointer() const;
   void CenterImage(ImPixelCoord);
   int ConformsWith(const Image&) const;
   ImPixelCoord GetImPixCoord(PixelCoord) const;
   ImageCoord GetImageCoord(ImPixelCoord) const;
   ImageCoord GetImageCoord(PixelCoord) const;
   ImPixelCoord GetImPixCoord(ImageCoord) const;
   PixelCoord GetPixelCoord(ImPixelCoord) const;
   PixelCoord GetPixelCoord(ImageCoord) const;
   void AddHistory(K_String);
   void AddHistory(const Image&);
   void ListHistory() const;
   virtual void Scale(float) = 0;
   virtual void Fill(float) = 0;
   virtual int Extrema(Pixel&, Pixel&) const = 0;
   Pixel Maximum() const;
   Pixel Minimum() const;
   virtual void SetPixel(PixelCoord, float) = 0;
   virtual float GetPixel(PixelCoord) const = 0;
   virtual int GetImPixelVal(const ImPixelCoord&, float&) const = 0;
   virtual int UCombine (float, const Image&, float, const Image&, float) = 0;
   virtual int XCombine (float, const Image&, float, const Image&, float) = 0;
   virtual int ScaledAdd(float, const Image&) = 0;
   virtual  ~Image();

protected:
   ImageDim dim;
   K_String pImtype;
   ImPixelCoord regblc, regtrc;
   PixelCoord cenpix;
   ImPixelCoord imcornpix;
   ImPixStep pixstep;
   ImageUnits  datatype;
   CoordSys *pCsys;
   HistFile *pHist;

   int CombTwoImages(int, const Image&, const Image&, float&, float&,
float&, float&);
   int CombOneImage(const Image&, float&, float&, float&, float&);
   void UpdateCenter();
};
```

MEMBER FUNCTIONS

Constructor ("fill with value" version must be in derived class). (Probably need more constructors.)

```
Image(int m =0, int n = 0);

// Accessors.
```

The following accessor returns the image dimensions. If the dimensions of an image are [m, n], then PixelCoords run from 0 to m-1 in the X dimension and from 0 to n-1 in the Y direction. Image dimensions can be set only by the constructor, or modified by adding pixels to a list image, or by an X/UCombine method. The default dimensions from the constructor are [0,0].

```
ImageDim GetDim() const;
```

Return the number of pixels in the image

```
virtual int GetNumEl() const = 0;
```

Return/set the image descriptor (a string)

```
K_String GetImageType() const;
void SetImageType(K_String s);
```

Set/return the specification of the "image center" in PixelCoord. N.B. The default "center" position is (m/2, n/2-1) in Pixel coordinates, i.e. displaced to Top Right of geometric image center.

```
void SetCenPix(PixelCoord p);
PixelCoord GetCenPix() const;
```

Set/return the ImPixelCoords of the TLC of the image (the origin (0,0) of PixelCoord).

```
void SetImCornPix(ImPixelCoord p);
ImPixelCoord GetImCornPix() const;
```

Set/return the ImPixStep for the image == the number of ImPixels per Pixel. Since the TLC is the origin for PixelCoords (CIC convention), the Y value of ImPixStep is negative. Step values must be integral.

```
void SetPixStep(ImPixStep r);
ImPixStep GetPixStep() const;
```

Set/return the type of data units for the image

```
void SetDataType(ImageUnits iu);
ImageUnits GetDataType() const;
```

Set/retrieve the "coordinate system" for the image. This defines the relationship between ImPixelCoord and some selected system of ImageCoords. The definition of the precise system of ImageCoords to be used is defined within the CoordSys object. When images are cloned, sub-images are formed, assignments made to new image objects, copied, etc., the same CoordSys object is referenced.

```
void SetCoordSys(CoordSys * pcs);
CoordSys * GetCoordSys() const;
```

Set/retrieve a "region of interest" for the image. This feature is defined by a BLC in ImPixelCoord and a TRC in ImPixelCoord. Implementation of a feature like this is not currently in place. Probably one would need a list of regions of interest.

```
void SetRegBLC(ImPixelCoord blc);
ImPixelCoord GetRegBLC() const;

void SetRegTRC(ImPixelCoord trc);
ImPixelCoord GetRegTRC() const;
```

Get pointer to history file

```
HistFile * GetHistPointer() const;
```

Method to shift "center" of image to a specified ImPixelCoord. The reference value (imcornpix) of the TLC of the image is modified.

```
void CenterImage(ImPixelCoord);
```

Checks whether another Image conforms with the current Image, i.e. can be used with it in a "combine" or "scaledAdd" operation. The two images must have the same "pCsys", the same "pixstep", the same "dataunit", and their "imcornpix" must be consistent with"pixstep". Returns TRUE if they are conformant, FALSE otherwise.

```
int ConformsWith(const Image&) const;
```

Coordinate conversion - forward (various coordinate systems) Note that the first may return non-integral values of ImPixelCoord.

```
ImPixelCoord GetImPixCoord(PixelCoord) const;
ImageCoord GetImageCoord(ImPixelCoord) const;
ImageCoord GetImageCoord(PixelCoord) const;
```

Coordinate conversion - reverse (various coordinate systems) Note that these may return non-integral values of ImPixelCoords or PixelCoord (even though the latter must be integral to refer to image data elements).

```
ImPixelCoord GetImPixCoord(ImageCoord) const;
PixelCoord GetPixelCoord(ImPixelCoord) const;
PixelCoord GetPixelCoord(ImageCoord) const;
```

History maintenance. Add comments (string), another Image's history, or print out file

```
void AddHistory(K_String);
void AddHistory(const Image&);
void ListHistory() const;
```

Scale all pixel values in the image by a scaling factor.

```
virtual void Scale(float) = 0;
```

Replace all pixel values in the image by a new value.

```
virtual void Fill(float) = 0;
```

Return the maximum and minimum Pixel value (by returning Pixels) in the image. Returns FALSE if image is empty, otherwise TRUE.

```
virtual int Extrema(Pixel&, Pixel&) const = 0;
```

Return the maximum and minimum Pixel values (by returning Pixels) in the image. Exits if image is empty.

```
Pixel Maximum() const;
Pixel Minimum() const;
```

Inserts a pixel value into the image with given PixelCoord.

```
virtual void SetPixel(PixelCoord, float) = 0;
```

Returns the data value of a pixel having a given PixelCoord. For a ListImage, it will return the value of the FIRST matching pixel in the list. If no matching pixel exists in the image, the method exits.

```
virtual float GetPixel(PixelCoord) const = 0;
```

Returns the data value of a Pixel having a specified ImPixel coordinate. This function may not always be able to find a matching pixel, in which case it returns FALSE. (There is no guarantee that an integral ImPixel coordinate will be an integral Pixel coordinate). In the case of a "ListImage", it will return the FIRST matching pixel found in the list.

```
virtual int GetImPixelVal(const ImPixelCoord&, float&) const = 0;

// Union image operators.
```

General UNION combine: Image C (*this) = factor1*Image A + factor2*Image B . Image C can be Filled or ListImage type, while Images A and B can be any image type. Images A and B must be conformant: same ImageUnits, same CoordSys, same ImPixStep and their "imcornpix" reference corners must be separated by an integral number of ImPixSteps. Image C will be made to be conformant with images A and B, and its history file (if any) will be repaced by that of A followed by B. The dimensions of C, and its reference corner, will be set to represent the true UNION of images A and B. Output pixels in C will be set to a "fill" value (only if C is a FilledImage) if there are no corresponding pixels in A or B. Otherwise, the output pixels in C will be = factor1*A (if pixel exists only in A), = factor2*B (if pixel exists only in B), = factor1*A + factor2*B (if pixel exists in both). The method returns FALSE if the images are non-conformant.

```
virtual int UCombine (float, const Image&, float, const Image&, float) = 0;

// Intersection image operators.
```

General INTERSECTION combine: Image C (*this) = factor1*Image A + factor2*Image B . Image C can be Filled or ListImage type, while Images A and B can be any image type. Images A and B must be conformant: same ImageUnits, same CoordSys, same ImPixStep and their "imcornpix" reference corners must be separated by an integral number of ImPixSteps. Image C will be made to be conformant with images A and B, and its history file (if any) will be repaced by that of A followed by B. The dimensions of C, and its reference corner, will be set to represent the true INTERSECTION of images A and B. Output pixels in C will be set to a "fill" value (only if C is a FilledImage) if there is no corresponding pixels in both A and B. (This can only happen if A or B is a ListImage.) Otherwise, the output pixels in C will be = factor1*A + factor2*B. The method returns FALSE if the images are non-conformant, or if there is no INTERSECTION.

```
virtual int XCombine (float, const Image&, float, const Image&, float) = 0;
```

General scaled-add: Image A (*this) = Image A + factor*Image B. Image A can be Filled or ListImage type, while B can be any image type. Images A and B must be conformant: same ImageUnits, same CoordSys, same ImPixStep and their "imcornpix" reference corners must be separated by an integral number of ImPixSteps. The history file of Image B will be added to that of Image A. The scaled-add operation will take place only for pixels in A that have corresponsding pixels in B. The method returns FALSE if the images are non-conformant.

```
virtual int ScaledAdd(float, const Image&) = 0;


// Destructor.
virtual  ~Image();
```

PROTECTED DATA MEMBERS

Dimensions of image [x,y]

```
ImageDim dim;
```

Image type identifier.

```
K_String pImtype;
```

The ImPixel coordinates of the BLC and TLC of a region of interest.

```
ImPixelCoord regblc, regtrc;
```

Image "centre", in pixel coordinates. Whenever the image dimensions are updated (at construction time or otherwise) this is set to the default value of (m/2, n/2-1) where the dimensions are [m,n]. The "CenterImage()" method uses this parameter.

```
PixelCoord cenpix;
```

ImPixel coordinates of Pixel coordinate (0,0), the top-left corner. This defines the position of the image with respect to ImPixel (and hence Image) coordinates. The default value of this is (1,n) when the dimensions are [m,n], i.e. the BLC is (1,1).

```
ImPixelCoord imcornpix;
```

Number of ImPixels per Pixel. This defines the ratio of ImPixel coordinates to Pixel coordinates, and the default value is (1,-1), corresponding to the CIC convention of having Pixel coordinate (0,0) at the TLC. When a new image is created by taking a sub-image of an existing image, this may change.

```
ImPixStep pixstep;
```

Data units for the pixel values in the image.

```
ImageUnits  datatype;
```

Associated coordinate system. Note that a coordinate system must be explicitly assigned (currently no constructor does this!) by inserting a pointer to it here.

```
CoordSys *pCsys;
```

Associated history (pointer to Hist File object). This is usually assigned automatically by the constructor, and the destructor removes the storage.

```
HistFile *pHist;
```

PROTECTED MEMBER FUNCTIONS

Methods for combining images. These are solely for the use of the derived classes in implementing dynamically bound combine and scaled add functions. Essentially they compute the "union" or "intersection" areas in terms of x and y values of ImPixel coordinates.

```
    int CombTwoImages(int, const Image&, const Image&, float&, float&,
float&, float&);
    int CombOneImage(const Image&, float&, float&, float&, float&);
```

Update the "centre pixel". This method is for the use of the base and derived classes, to update the "center" definition whenever the dimensions are changed.

```
void UpdateCenter();
```

A.11 ImageCoord.h

HEADER FILE DESCRIPTION

aips++

ENVIRONMENT

```
#define A_IMAGECOORD_H

#include <iostream.h>
```

CLASS DESCRIPTION

Class ImageCoord is used to define astronomical coordinates, or increments in astronomical coordinates. The units are usually assumed to be radians, but could have other units for specialized coordinate systems

CLASS SUMMARY

```
class ImageCoord
{
public:
        ImageCoord();
        ImageCoord(double i, double j);
        void SetImageCoord(int i, double m);
        void SetImageCoord(double *);
        double GetImageCoord(int i) const;
        double* GetImageCoord() const;
        ~ImageCoord();
};
ostream& operator<<(ostream&, ImageCoord);
ImageCoord operator + (const ImageCoord&, const ImageCoord&);
ImageCoord operator - (const ImageCoord&, const ImageCoord&);
ImageCoord operator * (const ImageCoord&, const float);
ImageCoord operator * (const float, const ImageCoord&);
```

MEMBER FUNCTIONS

Default constructor.

```
        ImageCoord();
```

Constructor which sets coordinates.

```
        ImageCoord(double i, double j);
```

Set i'th coordinate to m.

```
void SetImageCoord(int i, double m);
```

Set coordinates to values.

```
void SetImageCoord(double *);
```

Return i'th coordinate.

```
double GetImageCoord(int i) const;
```

Return coordinates.

```
double* GetImageCoord() const;
```

Destructor.

```
~ImageCoord();
```

NON-MEMBER FUNCTIONS

Print an ImageCoord.

```
ostream& operator<<(ostream&, ImageCoord);
```

Add two ImageCoords

```
ImageCoord operator + (const ImageCoord&, const ImageCoord&);
```

Subtract two ImageCoords

```
ImageCoord operator - (const ImageCoord&, const ImageCoord&);
```

Multiply an ImageCoord by a float

```
ImageCoord operator * (const ImageCoord&, const float);
```

Multiply a float by an ImageCoord

```
ImageCoord operator * (const float, const ImageCoord&);
```

A.12 ImageDim.h

HEADER FILE DESCRIPTION

Header file for ImageDim class

ENVIRONMENT

```
#define IMAGEDIM_H

#include <iostream.h>
```

CLASS DESCRIPTION

Class ImageDim defines the (vector) dimensions of an image. Currently it is defined only for two-dimensional images, i.e. there are two vector (int) elements.

CLASS SUMMARY

```
class ImageDim
{
public:
        void SetDim(int i, int val);
        void SetDim(int*);
        int GetDim(int i) const;
        int* GetDim() const;
        ImageDim();
        ImageDim(int i, int j);
        ~ImageDim();
};
ostream& operator << (ostream& os, const ImageDim id);
```

MEMBER FUNCTIONS

Set the i'th dimension to value: val. The value of i must be greater than 0 and less than 3 (currently)

```
        Preconditions:
                valid axis: 0 < i  < 3
                valid dimension value: 0 <= val

        Postconditions:
                invalid input parameters: assertion exit


         void SetDim(int i, int val);
```

Set the dimensions to the values given by an int pointer. There must be two valid values, and each must be positive.

```
Preconditions:
          valid number of values: 2
          valid dimension value: 0 <= val

Postconditions:
          invalid input parameters: assertion exit

 void SetDim(int*);
```

Get the dimension of the i'th axis. The value i must be greater than 0 and less than 2.

```
Preconditions:
          valid axis: 0 < i  < 3

Postconditions:
          true

 int GetDim(int i) const;
```

Return the dimensions

```
Preconditions:
          true

Postconditions:
          true

 int* GetDim() const;


 ImageDim();
 ImageDim(int i, int j);
 ~ImageDim();
```

NON-MEMBER FUNCTIONS

```
ostream& operator << (ostream& os, const ImageDim id);
```

A.13 ImageUnits.h

HEADER FILE DESCRIPTION

Header file for ImageUnits class

ENVIRONMENT

```
#define IMAGEUNITS_H

#include <iostream.h>
#include <K_String.h>
```

CLASS DESCRIPTION

Class ImageUnits defines the pixel units for an image. Currently this is defined by any arbitrary character string, but in future might be idefined as one of a number of predetermined descriptors, unit scaling factors, etc.

CLASS SUMMARY

```
class ImageUnits
{
public:
        void SetImUnits(K_String& );
        K_String GetImUnits() const;
        ImageUnits();
        ImageUnits(K_String s);
        ~ImageUnits();
};
ostream& operator << (ostream& os, const ImageUnits& iun);
int operator == (const ImageUnits& iu1, const ImageUnits& iu2);
```

MEMBER FUNCTIONS

Set the image units descriptor

```
        void SetImUnits(K_String& );
```

Return the image units descriptor

```
        K_String GetImUnits() const;

        ImageUnits();
        ImageUnits(K_String s);
        ~ImageUnits();
```

NON-MEMBER FUNCTIONS

Print out an ImageUnits object

```
ostream& operator << (ostream& os, const ImageUnits& iun);
```

Compare two ImageUnits for equality

```
int operator == (const ImageUnits& iu1, const ImageUnits& iu2);
```

A.14 ImagingModel.h

HEADER FILE DESCRIPTION

Uses DummyImage, must be replaced by Image

ENVIRONMENT

```
#define A_IMAGINGMODEL_H

#include "YegSet.h"
#include "DummyImage.h"
```

CLASS DESCRIPTION

CLASS SUMMARY

```
class ImagingModel
{
public:
   ImagingModel();
   ~ImagingModel();
   virtual int InvertYeg(YegSet &, DummyImage &);
   virtual int PredictYeg(YegSet &, DummyImage &);
};
```

MEMBER FUNCTIONS

```
   //Constructor
   ImagingModel();

   //Destructor
   ~ImagingModel();

   //Services
   virtual int InvertYeg(YegSet &, DummyImage &);
   virtual int PredictYeg(YegSet &, DummyImage &);
};
```

A.15 Input.h

HEADER FILE DESCRIPTION

header file for class Input

ENVIRONMENT

```
#define A_INPUT_H

#include <iostream.h>
#include <stdarg.h>
#include "Param.h"
#include <K_DList.h>                    // CIC

DECLARE_ONCE K_DList<Param>;            // Template kludge (CIC)
DECLARE_ONCE K_DLinkable<Param>;        // Template kludge (CIC)
```

CLASS DESCRIPTION

Class Input is: A linked list of parameters (defined by the helper class "Param") with various user interface attributes.

Part of an example of a traditional "key=value + help" command-line user interface.

CLASS SUMMARY

```
class Input
{
public:
     Input();
    ~Input();
     void Create(String key, String value, String help);
     void StdCreate(String stdkey, String key, String value, String help);
     void Close();
     double GetDouble(String key);
     int GetInt(String key);
     String GetString(String key);
     bool GetBool(String key);
     int Count();
     bool Debug(int l);
     bool Put(String key, String value);
     bool Put(String keyval);
};
extern Input inputs;
void error(char *fmt ...);              // The aips stdarg family
void warning(char *fmt ...);
```

```
void debug(int l, char *fmt ...);
```

MEMBER FUNCTIONS

The default constructor is the only one! It enables the creation of parameters. It puts the program in no-prompt mode unless environment variable HELP is defined with value "prompt". The output debug level is set according to the value of the environment variable DEBUG. The maximum number of error messages to be outputed is set according to the value of the environment variable ERROR.

```
Input();
```

Destructor.

```
~Input();
```

```
                                // parameter creation
```

Create a new parameter, either from scratch or looking it up from an internal list of templates.

The function also checks whether parameters can still be created, and whether key is unique for the program.

The value, help and remaining arguments are all optional.

```
void Create(String key, String value, String help);
void StdCreate(String stdkey, String key, String value, String help);
```

Disable the creation of parameters. Highly recommended, but not required?

```
void Close();
```

```
                                // query functions
```

Get the double value of the parameter (or 0.0 if unknown key). If the program is in prompt mode, ask the user for the value.

```
double GetDouble(String key);
```

Get the int value of the parameter (or 0 if unknown key). If the program is in prompt mode, ask the user for the value.

```
int GetInt(String key);
```

Get the string-type value of the parameter (or "" if unknown key). If the program is in prompt mode, ask the user for the value.

```
String GetString(String key);
```

Get the boolean value of the parameter (or FALSE if unknown key). If the program is in prompt mode, ask the user for the value.

```
bool GetBool(String key);
```

Get the number of parameters of this program

```
int Count();
```

See if the current debug level is thresholded

```
bool Debug(int l);
```

```
                                    // modify function
```

Set the value for a named parameter. Return FALSE if key is an unknown parameter name. The default value is "".

The function can also be called with a single argument of the form 'key=value', where key is a valid new parameter name, and where value may be empty (the '=' is required though). In this case a new parameter will be created (provided that creation is still allowed).

```
bool Put(String key, String value);
bool Put(String keyval);
```

NON-MEMBER FUNCTIONS

```
void error(char *fmt ...);          // The aips stdarg family
void warning(char *fmt ...);
void debug(int l, char *fmt ...);
```

A.16 IntYegSet.h

HEADER FILE DESCRIPTION

This file contains the definitions for the IntYegSet class.

ENVIRONMENT

```
#define INTYEGSET_H

#include "YegSet.h"
```

CLASS DESCRIPTION

The IntYegSet class provides services to return telescope, data, weights and coordinates.

CLASS SUMMARY

```
class IntYegSet:public YegSet
{
public:
    IntYegSet(const YegSet&);
    DVector u() const;
    DVector v() const;
    DVector w() const;
    DVector time() const;
    DVector r1() const;
    DVector r2() const;
    virtual void ShowYegSet (int i1, int i2);
};
```

MEMBER FUNCTIONS

Construct an IntYegSet from an YegSet. Default copy constructor should also work OK.

```
    IntYegSet(const YegSet&);
```

Return u,v,w coordinates, time and receptor numbers.

```
    DVector u() const;      // u in wavelengths.
    DVector v() const;      // v in wavelengths.
    DVector w() const;      // w in wavelengths.
    DVector time() const;   // time in seconds.
    DVector r1() const;     // Number of ant1.
    DVector r2() const;     // Number of ant2.
```

Display

```
virtual void ShowYegSet (int i1, int i2);
```

A.17 ListImage.h

HEADER FILE DESCRIPTION

Header file for ListImage class

ENVIRONMENT

```
#define LISTIMAGE_H

#include <iostream.h>
#include "Image.h"
#include "K_DList.h"
#include "Pixel.h"
#include "ModelImage.h"
#include "FilledImage.h"

DECLARE_ONCE K_DList<Pixel>;

class FilledImage;        // Needed for proper declaration order
class ModelImage;
```

CLASS DESCRIPTION

The ListImage class: pixels (values plus coordinates) are stored as a linked list.

CLASS SUMMARY

```
class ListImage : public Image
{
public:
        ListImage(int m = 0, int n = 0);
        ~ListImage();
        ListImage(const ListImage& src);
        ListImage(const ModelImage& src);
        ListImage(const FilledImage& src);
        ListImage & operator = (const ListImage&);
        ListImage & operator = (const ModelImage&);
        ListImage & operator = (const FilledImage&);
        int GetNumEl() const;
        void Scale(float);
        void Fill(float);
        void FillPixel(int, float);
        int Extrema(Pixel &, Pixel &) const;
        void SetPixel(PixelCoord, float);
        int PixelExists(int) const;
        int PixelExists(PixelCoord) const;
```

```
        float GetPixel(PixelCoord) const;
        int GetPixel(PixelCoord, float&) const;
        void DeletePixel(int);
        Pixel GetPixel(int) const;
        void SetToPix(int) const;
        Pixel GetNxtPixel() const;
        int GetImPixelVal(const ImPixelCoord&, float&) const;
        int GetFirstNeg() const;
        ListImage CloneEmpty() const;
        void SortMerge(int);
 friend ostream& operator << (ostream&, ListImage&);
        int ScaledAdd(float, const Image&);
        int UCombine(float, const Image&, float, const Image&, float);
        int XCombine(float, const Image&, float, const Image&, float);
};
end{verbatim}

\subsection*{MEMBER FUNCTIONS}

           Constructor and destructor
\begin{verbatim}
        ListImage(int m = 0, int n = 0);
        ~ListImage();
```

Copy constructors

```
        ListImage(const ListImage& src);
        ListImage(const ModelImage& src);
        ListImage(const FilledImage& src);
```

Assignment operators

```
        ListImage & operator = (const ListImage&);
        ListImage & operator = (const ModelImage&);
        ListImage & operator = (const FilledImage&);
```

Return number of elements

```
        int GetNumEl() const;
```

Scale all pixel values by a scaling factor

```
        void Scale(float);
```

Replace values of all existing pixels by a new value.

```
        void Fill(float);
```

Fill the value of the n'th pixel with a new number. n must be a valid serial pixel number (0 ¡ n ¡= number of elements)

```
void FillPixel(int, float);
```

Return maximum and minimum pixels in image. Returns FALSE if empty image.

```
int Extrema(Pixel &, Pixel &) const;
```

Set a pixel (add a pixel to the list)

```
void SetPixel(PixelCoord, float);
```

Check that pixel of serial number i exists.

```
int PixelExists(int) const;
```

Check that a pixel having given Pixel Coords exists.

```
int PixelExists(PixelCoord) const;
```

Retrieve the value of a pixel (scan through list). The value of the FIRST pixel with matching coordinates is returned. If there is no matching pixel, or if the image is empty, the method exits.

```
float GetPixel(PixelCoord) const;
```

Retrieve the value of a pixel (scan through list). The value of the FIRST pixel with matching coordinates is returned. The method returns FALSE if the image is empty or no matching pixel is found, otherwise TRUE.

```
int GetPixel(PixelCoord, float&) const;
```

Delete the n'th pixel in the list. The serial pixel number n must be ¿ 0 and ¡= the number of pixels.

```
void DeletePixel(int);
```

Retrieve the n'th pixel in the list. The serial pixel number n must be ¿ 0 and ¡= the number of pixels.

```
Pixel GetPixel(int) const;
```

Prepare to extract pixels having serial number n (and greater). This is a faster access method than the preceding. The number n must be in range 0 ¡ n ¡= number of pixels in image.

```
void SetToPix(int) const;
```

Get next pixel. Must be preceded by invocation of itself, or SetToPix(). Exits if one has run off end of pixel list!

```
Pixel GetNxtPixel() const;
```

Return value of FIRST pixel having ImPixelCoords. If there is no matching pixel, it returns FALSE, otherwise TRUE.

```
int GetImPixelVal(const ImPixelCoord&, float&) const;
```

Get serial pixel number of first negative pixel. Returns zero for empty image or no negative pixels.

```
int GetFirstNeg() const;
```

"Clone" a new ListImage that has an empty list but all other attributes of the current image

```
ListImage CloneEmpty() const;
```

Merge pixels with corresponding coordinates (i = 0), and merge and then sort into decreasing order of pixel value (i = 1). The latter is not yet implemented.

```
void SortMerge(int);
```

Print out a ListImage object. Note that a CoordSys must be defined!

```
friend ostream& operator << (ostream&, ListImage&);
```

Union and intersection combine and scaled-add. All methods used to implement these must be virtual methods!

```
int ScaledAdd(float, const Image&);
int UCombine(float, const Image&, float, const Image&, float);
int XCombine(float, const Image&, float, const Image&, float);
```

A.18 ModelImage.h

HEADER FILE DESCRIPTION

aips++ header

ENVIRONMENT

```
#define A_MODELIMAGE_H

#include <assert.h>
#include <iostream.h>
#include "Image.h"

typedef float (*pFunc)(float *,ImPixelCoord);
```

CLASS DESCRIPTION

Class: ModelImage

Image with pixel values defined by an analytical function.

An example of how to use ModelImages follows:

If one wished to have a 128 x 128 ModelImage of a Gaussian function centered on ImPixel coordinates (64,64) and having width paramters (5,5) and unity amplitude, the following is the procedure.

```
float Gaussian(float   parms, ImPixelCoord imp);
    {
        float xim = imp.GetImPixCoord(1);
        float yim = imp.GetImPixCoord(2);
        float dx = (xim - parms[1])/parms[3];
        float dy = (yim - parms[2])/parms[4];
        double val = parms[0]*exp(-dx*dx - dy*dy);
        return (float)val;
     }

 float gparms[5] = {1,64,64,5,5};


 ModelImage mim(128,128);
 // Attach a coordinate system here!
 mim.SetFunction(Gaussian);
 mim.SetParmList(gparms);
         etc.
```

NOTE: If one wished to have the Gaussian centred at given Pixel coordinates, one must (before any image operation) set gparms[1:2] to the ImPixel coordinates corresponding to

the Pixel coordinates. This requires a call to the Image method that makes this conversion.

CLASS SUMMARY

```
class ModelImage  : public Image
{
public:
   ModelImage ( int m = 0, int n = 0, pFunc f = 0, float *p = 0,
                float sc = 1.0);
   ModelImage(const ModelImage& src);
   ModelImage &operator=(const ModelImage& src);
   int GetNumEl() const;
   void SetParmList(float * pp);
   void SetFunction(pFunc fp);
   void Scale(float scl);
   void Fill(float);
   int Extrema(Pixel &maxpix, Pixel &minpix) const;
   void SetPixel(PixelCoord, float);
   float GetPixel(PixelCoord) const;
   int GetImPixelVal(const ImPixelCoord&, float&) const;
   int UCombine (float, const Image&, float, const Image&, float);
   int ScaledAdd(float, const Image&);
   int XCombine(float, const Image&, float, const Image&, float);
 friend ostream& operator << (ostream&, ModelImage&);
   ~ModelImage();
};
```

MEMBER FUNCTIONS

Make a ModelImage.

```
   ModelImage ( int m = 0, int n = 0, pFunc f = 0, float *p = 0,
        float sc = 1.0);
```

Copy constructor

```
   ModelImage(const ModelImage& src);
```

Asignment operator

```
   ModelImage &operator=(const ModelImage& src);
```

Accessors

```
   int GetNumEl() const;
   void SetParmList(float * pp);
   void SetFunction(pFunc fp);
```

Data operations.

```
void Scale(float scl);
void Fill(float);
```

Find extrema. Returns FALSE if empty image.

```
int Extrema(Pixel &maxpix, Pixel &minpix) const;
void SetPixel(PixelCoord, float);
```

Return "pixel" value. It will return a model image value, even for non-integral values of PixelCoord.

```
float GetPixel(PixelCoord) const;
```

Get pixel value, given ImPixel coordinates. Returns FALSE if the resulting PixelCoord is non-integral.

```
int GetImPixelVal(const ImPixelCoord&, float&) const;
```

Union image operator.

```
int UCombine (float, const Image&, float, const Image&, float);
```

Scaled Add and Intersection image operator.

```
int ScaledAdd(float, const Image&);
int XCombine(float, const Image&, float, const Image&, float);
```

Output a ModelImage object. Note that a CoordSys must be defined!

```
friend ostream& operator << (ostream&, ModelImage&);
```

Destructor.

```
~ModelImage();
```

A.19 Param.h

HEADER FILE DESCRIPTION

header file for class Param

ENVIRONMENT

```
#define A_PARAM_H

#include <iostream.h>
#include "String.h"              // Meant to be the GNU String !!!!
#include <string.h>              // things like strlen() and such

enum bool {FALSE, TRUE};
```

CLASS DESCRIPTION

Class Param ...

Part of an example of a traditional 'key=value + help' command-line user interface.

CLASS SUMMARY

```
class Param
{
public:
    Param();
    Param(String, String, String);
    Param(const Param&);
    ~Param();
    bool   operator == (const Param&) const;
    double GetDouble(bool do_prompt=FALSE);
    int    GetInt(bool do_prompt=FALSE);
    String GetString(bool do_prompt=FALSE);
    bool   GetBool(bool do_prompt=FALSE);
    String Get();
    String GetHelp();
    String GetKey();
    String KeyVal() const;
    bool   Put(const String a_value);
    void   Print(ostream& out) const;
};
 ostream& operator<< (ostream& os, const Param& p);
```

MEMBER FUNCTIONS

default constructor

```
Param();
```

normal constructor with optional value and help strings

```
Param(String, String, String);
```

copy constructor

```
Param(const Param&);
```

destructor

```
~Param();
```

```
                                    // operator functions:  =, == and <<
```

copy-assignment operator

```
Param& operator= (const Param&);
```

comparison operator

```
bool operator== (const Param&) const;
```

```
                                    // query functions
```

get a double parameter value; prompt if switch is TRUE

```
double GetDouble(bool do_prompt=FALSE);
```

get an int parameter value; prompt if switch is TRUE

```
int GetInt(bool do_prompt=FALSE);
```

get a string-type parameter value; prompt if switch is TRUE

```
String GetString(bool do_prompt=FALSE);
```

get a boolean parameter value; prompt if switch is TRUE

```
bool GetBool(bool do_prompt=FALSE);
```

get parameter value as a string

```
    String Get();
```

get parameter help string

```
    String GetHelp();
```

get parameter name

```
    String GetKey();
```

get the string 'key = value' for the parameter

```
    String KeyVal() const;
                                    // modify function
```

set new parameter value; return FALSE if invalid value

```
    bool Put(const String a_value);
                                    // Output function
    void Print(ostream& out) const;
```

NON-MEMBER FUNCTIONS

output operator

```
    inline ostream& operator<< (ostream& os, const Param& p)
```

A.20 Pixel.h

HEADER FILE DESCRIPTION

Class which stores a "float" pixel value and pixel coordinates.

ENVIRONMENT

```
#define A_PIXEL_H

#include <iostream.h>
#include "PixelCoord.h"
```

CLASS DESCRIPTION

CLASS SUMMARY

```
class Pixel
{
public:
        Pixel();
        Pixel(float v, float x, float y);
        void SetPixel(float v, PixelCoord pc);
        void SetPixel(float v);
        void SetPixel(PixelCoord c);
        float GetPixelValue() const;
        PixelCoord GetPixelCoord() const;
        ~Pixel();
};
ostream& operator<<(ostream&, Pixel);
int operator == (const Pixel&, const Pixel&);
```

MEMBER FUNCTIONS

Default constructor.

```
        Pixel();
```

Constructor which sets coordinates.

```
        Pixel(float v, float x, float y);
```

Set pixel coordinate and value.

```
        void SetPixel(float v, PixelCoord pc);
```

Set pixel value.

```
void SetPixel(float v);
```

Set pixel coordinate.

```
void SetPixel(PixelCoord c);
```

Get pixel value.

```
float GetPixelValue() const;
```

Get pixel coordinate.

```
PixelCoord GetPixelCoord() const;
```

Destructor.

```
~Pixel();
```

NON-MEMBER FUNCTIONS

Print a Pixel.

```
ostream& operator<<(ostream&, Pixel);
```

Compare two pixels for equality

```
int operator == (const Pixel&, const Pixel&);
```

A.21 PixelCoord.h

HEADER FILE DESCRIPTION

The PixelCoord class is used for holding positions in the "storage" coordinates. They may have fractional values, although in general use they are restricted to integers (indexes along the coordinate axes. If the dimensions of an image are m,n, the PixelCoords range from 0 to m-1, and 0 to n-1. The valid operations essentially consist of setting and getting values. Dimensionality is implicitly 2. PixelCoords must always be positive when being used within an image but can have negative values otherwise. Therefore there is trap for negative values.

It would probably be very useful to define coordinate arithmetic and comparisons on this type. (¡,¿,-,= etc).

ENVIRONMENT

```
#define PIXELCOORD_H

#include <iostream.h>
#include <assert.h>
```

CLASS DESCRIPTION

CLASS SUMMARY

```
class PixelCoord
{
public:
    PixelCoord();
    PixelCoord(float ri, float rj);
    virtual ~PixelCoord();
    virtual int Ok() const;
    void SetPixelCoord(int i, float v);
    void SetPixelCoord(float *fp);
    float GetPixelCoord(int i) const;
    float* GetPixelCoord() const;
};
ostream& operator<< (ostream&, PixelCoord);
PixelCoord operator + (const PixelCoord&, const PixelCoord&);
PixelCoord operator * (const PixelCoord&, const float);
PixelCoord operator * (const float, const PixelCoord&);
PixelCoord operator - (const PixelCoord&, const PixelCoord&);
int operator == (const PixelCoord&, const PixelCoord&);
```

MEMBER FUNCTIONS

The default constructor sets the positions to 0.0

```
PixelCoord();
```

Construct at a given position

```
PixelCoord(float ri, float rj);
```

Destructor currently does nothing

```
virtual ~PixelCoord() {}
```

Checks the state of this object

```
virtual int Ok() const;              // This object always OK
```

Set the location on the i'th axis. Range checks i.

```
void SetPixelCoord(int i, float v);
```

Sets the positions on all axes. Should be a vector type so we could check the length of the vector.

```
void SetPixelCoord(float *fp);
```

Gets the value on the i'th axis

```
float GetPixelCoord(int i) const;
```

Returns the value on all axes (should be vector)

```
float* GetPixelCoord() const;
```

NON-MEMBER FUNCTIONS

Print a PixelCoord

```
ostream& operator<< (ostream&, PixelCoord);
```

Add two PixelCoords

```
PixelCoord operator + (const PixelCoord&, const PixelCoord&);
```

Multiply a PixelCoord by a float

```
PixelCoord operator * (const PixelCoord&, const float);
```

Multiply a float by a PixelCoord

```
PixelCoord operator * (const float, const PixelCoord&);
```

Subtract two PixelCoords

```
PixelCoord operator - (const PixelCoord&, const PixelCoord&);
```

Compare two PixelCoords for equality

```
int operator == (const PixelCoord&, const PixelCoord&);
```

A.22 TelModel.h

HEADER FILE DESCRIPTION

This file contains definitions for the TelModel class.

ENVIRONMENT

```
#define TELMODEL_H

#include <assert.h>
#include <math.h>
#include "String.h"
#include <complex.h>
#include "GainTable.h"
#include "YegSet.h"
#include "Telescope.h"
```

CLASS DESCRIPTION

TelModel class.

CLASS SUMMARY

```
class TelModel
{
public:
    TelModel(int, float, float, float);
    ~TelModel();
    void Update(YegSet&, YegSet&);
    YegSet& Apply(const YegSet&);
    complex InterpGain(int, float);
};
```

MEMBER FUNCTIONS

Constructor for TelescopeModel specifying: number of receptors; time of first entry; time of last entry; time interval between entries.

```
    TelModel(int, float, float, float);
```

Destructor.

```
    ~TelModel();
```

Update this model given a pair (measured and predicted) of YegSets. This model solves for and updates its internal state. The YegSets remain unchanged.

```
void Update(YegSet&, YegSet&);
```

Create a new, corrected, YegSet and associated Telescope from an existing YegSet by applying this model. The preexisting YegSet remains unchanged. A copy of this model is attached to the new telescope as its default model.

```
YegSet& Apply(const YegSet&);
```

Return an interpolated gain given: a receptor number; time. This simple performs a crude linear interpolation between the two nearest entries in the gain table.

```
complex InterpGain(int, float);
```

A.23 Telescope.h

HEADER FILE DESCRIPTION

This file contains the defintions for the Telescope class.

CLASS DESCRIPTION

The Telescope class is mainly a book-keeping class, being a convenient place to hang together pointers to objects that are used by others.

ENVIRONMENT

```
#define TELESCOPE_H

#include <String.h>
#include <YegSet.h>
#include <TelModel.h>
#include <ImagingModel.h>
```

CLASS SUMMARY

```
class Telescope{
public:
    Telescope(const String&);
    Telescope(const TelModel&);
    Telescope(const String&, const TelModel&);
    void SetName(const String& myName);
    const String Name();
    void SetYegs(YegSet& myYegs);
    YegSet& Yegs();
    double Coord(const String&) const;
    void SetTelMod(TelModel& myModel);
    TelModel& TelMod() const;
    void SetImMod(ImagingModel& myModel);
    ImagingModel& ImMod() const;
};
```

MEMBER FUNCTIONS

Construct the telescope, just giving the telescope name. In this case, the default telescope model is a null.

```
    Telescope(const String&);
```

Construct the telescope, giving just the default telescope model. In this case, the telescope name is "Unknown-Telescope" !!

```
Telescope(const TelModel&);
```

Construct the telescope giving both the telescope name and the default model.

```
Telescope(const String&, const TelModel&);
```

Set and return the name of the telescope.

```
void SetName(const String& myName);
const String Name();
```

Set and return the YegSet associated with this telescope.

```
void SetYegs(YegSet& myYegs);
YegSet& Yegs();
```

Inquire about coordinates of this observation. e.g. freq.

```
double Coord(const String&) const;
```

Set and return the telescope model associated with this telescope.

```
void SetTelMod(TelModel& myModel);
TelModel& TelMod() const;
```

Set and return the imaging model associated with this telescope.

```
void SetImMod(ImagingModel& myModel);
ImagingModel& ImMod() const;
```

A.24 XYZ.h

HEADER FILE DESCRIPTION

This file contains the definitions for the XYZ position class.

ENVIRONMENT

```
#define XYZ_H

#include "Vector.h"
#include <assert.h>
```

CLASS DESCRIPTION

The XYZ position class is used to store the location of various antennas. Positions are assumed to be in an equatorial system. That is, Y axis points east, Z axis is parallel to the pole, and the X axis to defined to make a right-handed orthogonal system. Units are assumed to be nanoseconds.

CLASS SUMMARY

```
class XYZ
{
public:
    XYZ(int n, const double* myx, const double* myy, const double* myz);
    int Dim() const;
    double X(int i) const;
    double Y(int i) const;
    double Z(int i) const;
};
```

MEMBER FUNCTIONS

Construct the antenna locations from arrays of the positions. Input are the antenna locations and the number of antenna.

```
    XYZ(int n,                  // Number of antennas.
      const double* myx,        // Antenna locations in X.
      const double* myy,        // Antenna locations in Y.
      const double* myz)        // Antenna locations in Z.
```

Function to return the number of antenna.

```
    int Dim() const;
```

Functions to return the locations of the various antenna.

```
double X(int i) const;
double Y(int i) const;
double Z(int i) const;
```

A.25 YegRep.h

HEADER FILE DESCRIPTION

This file contains the definitions for the YegRep class.

ENVIRONMENT

```
#define YEGREP_H

#include "Vector.h"
#include "AssArray.h"

declare(AssArray,DVector)

class Telescope;
```

CLASS DESCRIPTION

The YegRep class provides services to return telescope, data, weights and coordinates.

CLASS SUMMARY

```
class YegRep
{
public:
    YegRep(const Telescope&,CVector);
    YegRep(const YegRep&);
    void SetData(CVector);
    void SetWeight(RVector);
    void SetCoord(const String&,DVector);
    const Telescope& tel() const;
    int NumYegs() const;
    CVector Data() const;
    RVector Weight() const;
    DVector Coord(const String&) const;
    int CoordPresent(const String& name) const;
    int RefInc();
    int RefDec();
};
```

MEMBER FUNCTIONS

Construct the YegRep from a telescope and a vector of complex data.

```
    YegRep(const Telescope&,CVector);
```

The copy constructor: Make a true copy of a YegRep.

```
YegRep(const YegRep&);
```

Overwrite the data.

```
void SetData(CVector);
```

Initialise the weights.

```
void SetWeight(RVector);
```

Initialise a coordinate.

```
void SetCoord(const String&,DVector);
```

Return a reference to the associated telescope.

```
const Telescope& tel() const;
```

Return the number of yegs.

```
int NumYegs() const;
```

Return the vector giving the data.

```
CVector Data() const;
```

Return a single precision vector of the weights.

```
RVector Weight() const;
```

Return a double precision vector of a particular coordinate.

```
DVector Coord(const String&) const;
```

Check if a particular coordinate is available.

```
int CoordPresent(const String& name) const;
```

Allow access to the reference counting mechanism.

```
int RefInc();
int RefDec();
```

A.26 YegSet.h

HEADER FILE DESCRIPTION

This file contains the definitions for the YegSet class.

ENVIRONMENT

```
#define YEGSET_H

#include "Vector.h"
#include "String.h"
#include "YegRep.h"

class Telescope;
```

CLASS DESCRIPTION

The YegSet class provides services to return telescope, data, weights and coordinates.

CLASS SUMMARY

```
class YegSet
{
  friend class IntYegSet;
public:
    YegSet(const Telescope& tel,CVector data);
    YegSet(const YegSet& ys);
    virtual ~YegSet();
    void SetData(CVector cv);
    void SetWeight(RVector rv);
    void SetCoord(const String& name,DVector dv);
    const Telescope& tel() const;
    int NumYegs() const;
    CVector Data() const;
    RVector Weight() const;
    DVector Coord(const String& name) const;
    virtual void ShowYegSet (int i1, int i2);
    int CoordPresent(const String& name) const;
};
```

MEMBER FUNCTIONS

The only constructor.

```
    YegSet(const Telescope& tel,CVector data);
```

Constructed from another YegSet.

```
YegSet(const YegSet& ys);
```

The destuctor.

```
virtual ~YegSet();
```

Overwrite the data.

```
void SetData(CVector cv);
```

Initialise the weights.

```
void SetWeight(RVector rv);
```

Initialise a coordinate.

```
void SetCoord(const String& name,DVector dv);
```

Return a reference to the associated telescope.

```
const Telescope& tel() const;
```

Return the number of yegs.

```
int NumYegs() const;
```

Return the vector giving the data.

```
CVector Data() const;
```

Return a single precision vector of the weights.

```
RVector Weight() const;
```

Return a double precision vector of a particular coordinate.

```
DVector Coord(const String& name) const;
```

Virtual Display Function

```
virtual void ShowYegSet (int i1, int i2);
```

Check if a particular coordinate is available.

```
int CoordPresent(const String& name) const;
```

Appendix B

Details

This chapter is just a bag of tricks. The items may not belong in this report, but their inclusion at least ensures that they do not get lost.

B.1 Object-state checking

Brian Glendenning

It is useful, especially when the code is being debugged, to be able to check the "state" of an object (in Eiffel this is called checking the invariants). It is useful to write a member function `int Ok()` that performs such a check and returns `0` if not successful, `1` otherwise (or boolean values). This function should be virtual for obvious reasons, and a subclass that redefines `Ok()` should probably within it also refer to its parent classes `Ok()`, e.g., refer to `Parent::Ok()` to ensure that the parent class is also consistent.

Conceptually you want to check the state of the object after construction, and at entry to every member function. However this state checking could be computationally expensive (e.g., checking all the pointers in a complicated data structure) so it must be possible to compile it out, preferably on a class by class basis, for the production system. A way of doing this is to use the macro `OK` rather than the function `Ok()` (obviously the latter is still available). Then the macro `OK` can be defined to be nothing, or can be defined to be something like `assert(Ok())` (the assert should be replaced with the exception handling mechanism when available). Every `.h` file for classes which use the `Ok()` mechanism should have OK default to something, e.g. if you wanted to check state by default:

```
#ifndef OK
#define OK assert(Ok())     // Use exceptions when available
#endif /* OK */
```

This can be overridden at compile time (probably in the makefile).

B.2 Efficient indexing in vectors

Bob Sault

The three vector classes RVector, DVector and CVector (for real, double precision and complex) have methods defined on them to do all the normal arithmetic operations (addition, subtraction, multiplcation, division), and functions (sin, cos, etc), as well as interconversion. They use a reference counting scheme, and a copy-on-modify policy to reduce the about of copying.

The implementation of the indexing operator which returns a data element of the array (i.e operator []) is noteworthy. Convention dictates that this should act as either an lvalue or an rvalue. Having this operator always return a reference to a data-element would have disadvantages. As the operator would not know whether it is being used as an lvalue or an rvalue, it would have to assume the worse – an lvalue – which would significantly reduce the effectiveness of the reference counting and copy-on-modify policy. The way used to avoid this loss is to have two overloaded versions of the indexing operator, defined (for the real vector class) as

```
float operator[](int) const;
```

and

```
float& operator[](int);
```

The first version can only appear as an rvalue, and the compiler will use this in preference whenever the vector is of const type. The second version can appear as both an lvalue and an rvalue, and the compiler will use this for non-const vectors. Provided the programmer uses the const declaration wherever possible, the effectiveness of the reference counting, etc is maintained. For a multiply-referenced, non-const vector, then a copy will have to be made sooner or later anyway, so it does not matter what this is initiated by the indexing operator (even in instances where it is used as an rvalue).

Coplien discusses an alternate scheme (pp. 50-52) which would be applicable here. Coplien's scheme is quite a bit more expensive – a very undesirable characteristic in such a common operation as indexing. It also has the disadvantage that the indexing operator would always be a non-const method.

Bibliography

[1] *Calibration, Imaging and Datasystems for AIPS++* — Report of the meeting held at Green Bank, West Virginia 3d – 14th February, 1992, Ed. D.L. Shone and T.J. Cornwell.

[2] *A Rational Plan for AIPS++*, Chris Flatters, March 12 1992.

[3] *A Delayed Reaction to Green Bank*, B.E. Glendenning; *Comments on Brian's note*, R.?. Sault; *++GreenBank - Part 1*, D.L. Shone, March 1992.