# AIPS++ Libraries Overview

James E. Horstkotte, NRAO

DRAFT 07 December 1994

## 1 Overview

This section provides an overview of the AIPS++ system. It is extracted, with very minor modifications, from the Overview chapter of the AIPS++ System manual, written by Mark Calabretta, the author of the AIPS++ code management and distribution system.

In the following discussion of the AIPS++ directory hierarchy we will assume that AIPS++ has been installed in directory `/aips++`. This is the preferred location, but in practice the AIPS++ "root" directory can reside anywhere. The root directory is generally referred to as `$AIPSROOT` and many other directories have standard variable names.

The major subdirectories of `/aips++` are `code` (`$AIPSCODE`), `docs` (`$AIPSDOCS`), and one or more architecture–specific subdirectories with names such as `sun4` and `ibmrs` which contain the AIPS++ system – that is, everything needed to run AIPS++ , including executables and sharable objects. These architecture–specific subdirectories are referred to collectively as `$AIPSARCH`.

The directory hierarchy beneath `/aips++/code`, or `$AIPSCODE`, consists of a collection of packages which are contained in separate subdirectories. The principle package is the `aips` package, which contains the source code for a class library required by all other packages. It also contains applications common to all areas of astronomical data processing. The contents of this library will be discussed in greater detail in the second part of this paper.

The `dish`, `synthesis`, and `vlbi` packages contain standard classes and applications common to single dish, aperture synthesis, and VLBI data processing tasks from all radio telescopes. Apart from a dependence of the `vlbi` package on the `synthesis` package, inclusion of these standard packages in an end–user installation is optional.

A `contrib` package contains source code contributed from the AIPS++ user community for redistribution with AIPS++ . If found to be generally useful, code from the `contrib` package may eventually be merged into one of the standard packages, but otherwise it is unsupported.

A `trial` package contains "trial" source code which must be shared by several developers during its development, but which is not ready for general release. When the code is ready, it is moved to another standard package.

Each AIPS++ consortium member is also entitled to maintain a package for data processing applications specific to their telescope(s). The sources for these classes and applications reside in the consortium–specific packages: `atnf`, `bima`, `drao`, `nfra`, `nral`, `nrao`, and `tifr`. These may or may not use the standard `dish`, `synthesis`, and `vlbi` packages, and their installation is also optional.

The standard AIPS++ packages (`aips`, `dish`, `synthesis`, and `vlbi`) contain an `implement` subdirectory which contains class header and implementation files, a `fortran` subdirectory which contains FORTRAN subroutines, a `test` subdirectory which contains test applications, and `scripts` and `data` subdirectories for package–related procedure files and system data such as standard colour–maps, or source catalogues. The `implement` directory can contain module subdirectories, which can in turn contain `test` directories. A particular module subdirectory contains files implementing classes and functions with related functionality, e.g. the `Tables` module in the `aips` package contains the files implementing the AIPS++ Table system. The `test` subdirectory of a module contains test applications for the module software. The applications for a particular package reside within the package subdirectory itself.

The substructure of the consortium–specific packages is left entirely to AIPS++ consortium members to determine.

There are a number of other subdirectories of `/aips++/code` which are unrelated to packages. The `install` subdirectory contains all of the utilities required to install and maintain AIPS++ . `doc` contains AIPS++ documentation sources, including the AIPS++ "specs", "memos", and "notes" series, and reference and design documentation in the corresponding subdirectories.

Also below `/aips++/code` is the `include` subdirectory which contains symbolic links to the `implement` subdirectory for each package. The purpose of these symlinks is to allow AIPS++ includes to be specified as "`#include <package/Header.h>`" by adding `-I/aips++/code/include` to the include path.

On AIPS++ consortium installations an additional `admin` subdirectory of `/aips++/code` contains files relating to the administration of the AIPS++ project.

## 2 Aips infrastructure library.

This section contains a more detailed description of the contents of the library for the `aips` package, describing the functionality provided. This will be organized from the most basic functionality such as run time type information (RTTI) and exceptions, through higher level utilities such as Arrays and Tables, to items of astronomical utility such as Measurement models, etc.

For the most part, the descriptions themselves have been extracted from the code and edited slightly.

- Basics
  - RTTI
  - Exceptions
- Program Interface
  - Program Input
  - Glish
  - AipsIO
  - FITS
- OS Interface
  - Environment Variables
  - Files
  - Dates and Times
  - Timing
  - Event Handling
  - X Wrappers
- Utilities
  - Compare

  - Sort
  - Fallible
  - PtrHolder
  - Assert
  - Counted Pointer
  - Dynamic Buffer
  - Sequence
  - Register
  - KeyWords
- GNU Utilities
  - Regular Expressions
  - Strings
- Math. Utilities
  - Constants
  - Hashing
  - Interpolation
  - Math. Functions
- GNU Math. Utilities
  - Random Numbers
- “Simple” Containers
  - Bit Vector
  - Blocks
  - General Iterator Support
  - Linked Lists
  - Stacks
  - Maps
- High Level “Containers”
  - Arrays

  - Tables
- High Level Utilities
  - Logging
  - Visualize
  - Gridding
  - Fourier Transforms
  - Deconvolution
  - Dummy Image
- Orphaned Code to be Picked Up
  - ObjectID
  - Catalogs
  - Measurement Model
  - Telescope Model
- New Code Soon to be Released
  - Queue
  - MaskedArray
  - Lattice
  - PagedArray
  - Image
  - Unit
  - Linear Least Squares Fitting
  - Single Dish
  - Graphics

## 2.1 Basics

### 2.1.1 RTTI

AIPS++ has implemented a runtime type information (RTTI) system. This system provides type information which is consistent through multiple executions of a program. This implementation goes beyond what will be provided as a standard part of C++ in the future because the C++ standard only guarantees that the type information will remain consistent throughout the current exection of a program.

The RTTI information is used by the exception mechanism. It is also used by AipsIO to identify the type of a datum written to or read from external storage.

#### 2.1.2 Exceptions.

AIPS++ has its own exception mechanism. Exceptions can be thrown and caught. Objects created on the heap are cleaned up after properly.

`AipsError` is the base class for all of the AIPS++ error classes. Because all of the errors have a common base class, any error can be caught with a single catch statement.

This class has a string which allows error messages to be propagated.

### 2.2 Program Interface

#### 2.2.1 Program Input

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

It is used to provide input to a program through program invocation arguments.

#### 2.2.2 Glish

This module contains AIPS++ wrapper classes for Glish values and events.

AIPS++ is using Glish as a user command line interpreter. Functionality is bound to the CLI by passing events back and forth between it and server programs, in this case written in C++. While it would have been possible to use the native Glish classes, it is somewhat tedious to do so in the context of AIPS++ ; c.f. the documentation for `GlishArray`.

The classes in the Glish module are intended to make it easy to construct values which are sent back and forth (typically via some type of IPC) between various processes.

The classes in this module are used to create values (array or record type) which are usually then transmitted to, or received from, some other process. The major external classes in this module are:

**GlishValue:** Base class for all Glish value objects. `GlishValue`s follow copy–on–write semantics, so assignment and passing and return by value are cheap.

**GlishArray:** A (possibly n–dimensional) array of glish values, all of which have the same type. This is also the class which is used to hold a single value.

**GlishRecord:** A “structure” of values, made up of named arrays and records (the latter makes it hierarchical).

**GlishSysEvent:** A named value which has been sent from an external source.

**GlishSysEventSource:** The object inside the program which emits the `GlishSysEvent` objects. It is also the object to which events from this executable are posted.

**GlishSysEventTarget:** Allows glish event handlers to be installed.

For a general introduction to Glish, see `ftp://ee.lbl.gov/glish/USENIX--93.ps.Z` .

### 2.2.3 AipsIO

`AipsIO` is a class designed to do IO for objects. It stores the data in canonical format using the routines `ToLocal` and `FromLocal`.

The output can be written into / read from a filebuf file. This file can be opened by `AipsIO` (by constructing with a filename) or the user can open the file himself and pass a file descriptor. The file can be opened by the constructor and closed by the destructor, but it can also be done “manually” using the open and close routine.

An object is written by writing all its data members. It will be preceeded by a header containing type and version. The IO can be done via the overloaded << and >> operators to write or read a single item (e.g. an int or an object). These operators are already defined for all built–in data types and for `Complex`, `DComplex`, `Char*`, `String` and `Bool`. The write is also defined for `SubString`.

There are also functions `put`, `get`, and `getnew` to write or read an array of values. These functions are defined for the same data types as << and >> (so one can write, for example, an array of `Strings`).

**AipsIO.put (nr, arr)** writes nr values from the given array.

**AipsIO.get (nr, arr)** reads nr values into the given user–supplied array.

**AipsIO.getnew (&nr, &arr)** reads the number of values written into an array allocated on the heap. It returns the nr of values read and a pointer to the array.

The data must be read back in the same order as it was written.

#### 2.2.4 FITS

This module contains classes for reading and writing FITS files. All of the usual FITS header data units are provided.

## 2.3 OS Interface

### 2.3.1 Environment Variables

The `EnvironmentVariables` class works with the definitions, names, and values of environment variables.

### 2.3.2 Files

AIPS++ provides classes for working with paths, files, directories, symbolic links, etc.

### 2.3.3 Dates and Times

The `Time` class handles date and times with modified Julian day number and calendar Time.

### 2.3.4 Timing

The `Timer` class provides an interface to system timing. It allows a C++ program to record the time between a reference point (mark) and now. This class uses the system `times(2)` interface to provide time resolution at either millisecond or microsecond granularity, depending upon operating system support and features. Since the time duration is stored in a 32–bit word, the maximum time period before rollover occurs is about 71 minutes.

Due to operating system dependencies, the accuracy of all member function results may not be as documented. For example, some operating systems do not support timers with microsecond resolution. In those cases, the values returned are provided to the nearest millisecond or other unit of time as appropriate. See the `Timer` header file for system–specific notes.

This `Timer` class is based on the TI COOL library `Timer` class.

### 2.3.5 Event Handling

`SysEvent` is the base class for all system events. It is simply a wrapper around the basic types of events which must be handled in the system, e.g. X events, Glish events, signals, etc. It provides a common interface to all events.

### 2.3.6 X wrappers

The `XSysEvent` class is a wrapper for X events. It provides a standard (minimal) interface to X events.

Often one needs to mix X events with other types of events. This class provides a common interface for X events, and all of the replated classes provide a system for handling most asynchronous types of events which occur.

## 2.4 Utilities

### 2.4.1 Compare

`Compare` is a templated function to compare two objects.

### 2.4.2 Sort

AIPS++ provides two classes for sorting.

The `Sort` class allows you to sort data on one or more keys in a mix of ascending and descending order. Optionally duplicates will be skipped. The data can be in one record or in separate records. The sort algorithm does not sort the records themselves, but returns an index to the records. A variety of sort algorithms are provided including insertion sort, quicksort, and heapsort.

The static member functions of class `GenSort` are highly optimized sort functions. They do an in–place sort of an array of values. The functions are templated, so they can in principle be used with any data type. However, if used with non-builtin data types, their class must contain the following functions:

- `operator=` (to assign when swapping elements)
- `operator<`, `operator>` and `operator==` (to compare elements)
- the default constructor (to allocate a temporary)

If it is impossible or too expensive to define these functions, the `Sort` class can be used instead. This sorts indirectly using an index array. Instead of the functions mentioned above it requires a comparison routine.

The following can be sorted:

- A "C-array" of values
- An `Array` of values. The array can have any shape and the increment can be $> 1$.
- A `Block` of values. There is a special functions to sort less elements than the size of the `Block`.

The options field can be used to choose a sort algorithm.

- `Sort::QuickSort` is the fastest. It is about 4–6 times faster than the `qsort` function on the SUN. No worst case has been found, even not for cases where `qsort` is terrible slow.
- `Sort::HeapSort` is about twice as slow as `quicksort`. It has the advantage that the worst case is always $o(n*log(n))$, while `quicksort` can have hypothetical inputs with $o(n*n)$.
- `Sort::InsSort` is $o(n*n)$ for random inputs. It is, however, the only stable sort (i.e. equal keys remain in the same order).

Furthermore `Sort::NoDuplicates` can be given in the options field, to indicate that duplicate keys should be removed. The return value is the nr of unique keys. Default is `Sort::QuickSort`.

The order field can be `Sort::Ascending` or `Sort::Descending`. Default is `Sort::Ascending`.

### 2.4.3 Fallible

The `Fallible` class is used to mark a value, in particular a return value, as valid or invalid.

### 2.4.4 PtrHolder

The `PtrHolder` class holds pointers to be deleted when exceptions are thrown.

### 2.4.5 Assert

There is a set of classes for making assertions about the results of program execution. This assertion mechanism can be turned on or off at compile time with the AIPS_DEBUG preprocessor macro.

The exception handling mechanism is used to throw the errors when an assertion fails. An arbitrary exception can be thrown when an assertion fails with `lAssert`, but `Assert` always throws an `AbortError`.

The two ways of using the assertions mechanism are:

- `Assert(expr)`
- `lAssert(expr, exception)`

The `Assert(expr)` form is intended to be used by "end users" because it causes the program to abort if `expr` evaluates to a null value. This is for the "end users" because presumably at their level there is no way to recover from errors.

The `lAssert(expr,exception)` form throws the specified exception if the `expr` is null. This exception can be caught in the regular way.

### 2.4.6 Counted Pointer

The `Counted Pointer` classes implement a simple reference counting mechanism. They allow `Counted Pointer`s to be passed around freely, incrementing or decrementing the reference count as needed when one `Counted Pointer` is assigned to another. When the reference count reaches zero the internal storage is deleted by default, but this behavior can be overridden.

There are several different types of `Counted Pointer`s:

- SimpleCountedPtr
- SimpleCountedConstPtr

- CountedPtr
- CountedConstPtr

The *Simple* classes do not have the `operator->()` operator. This means that it puts less demands on the underlying type.

The *Const* classes do not allow the underlying value to be modified.

### 2.4.7 Dynamic Buffer.

`DynBuffer` allows one to store data in dynamically allocated buffers. If a buffer is full, another one is allocated. This means the data is not consecutive in memory. After having stored the data, one can get the buffer addresses and sizes to write the data for instance.

This class is developed for `AipsIO` as an intermediate buffer, but it may serve other purposes as well.

### 2.4.8 Sequence

The `Sequence` class is the virtual base class for sequences in the library. It is templated to allow users to base derived sequences on any type, e.g. `libg++`'s `Integers`.

### 2.4.9 Register (simple type identification)

The `Register` function provides a templated function to provide simple type identification. This function provides a "simple" type identification mechanism which will provide "typeid"s for classes which:

- Only involve single inheritance.
- Only need ids which are unique for a given execution.

### 2.4.10 KeyWords

`KeywordSet` is the abstract base class for keyword classes like `ScalarKeywordSet`. It defines the common keyword name pool to be used by the `TypedKeywords<T>` members of the `XKeywordSet` classes. All these members share the same name pool to prevent them from using the same names. Furthermore `KeywordSet` defines the basic data members and functions for the derived classes. Useful functionalities are:

- It is possible link to another keyword set to enforce that the keywords in the 2 sets have different names.
- It is possible to exclude keyword names and data types.
- A comment string can be defined for each keyword.
- A map iterator can be constructed to iterate through all the keywords in the set.
- Testing if one set is a subset or superset of another.

## 2.5 GNU Utilities

### 2.5.1 Regular Expressions

AIPS++ provides a regular expression class.

### 2.5.2 Strings

AIPS++ `String`s are the gnu string implementation, modified by AIPS++ to use AIPS++ style exceptions, move some things out of line, etc.

## 2.6 Math. Utilities

### 2.6.1 Constants

Constants class for mathematical and physical constants.

Implementation–defined limits usually defined in <`limits.h`>, <`float.h`>, and <`values.h`> as preprocessor defines. Inclusion of `aips/Constants.h` is sufficient to ensure that they are defined for any particular implementation, and the correct functioning of the `tConstants` test program guarantees this.

Refer to:

```
Section 3.2c, pp28-30 of
"The Annotated C++ Reference Manual",
Ellis, M.A., and Stroustrup, B.,
Addison-Wesley Publishing Company, 1990.
IBSN 0-201-51459-1.
```

and

```
Appendix B11, pp257-8 of
"C Programming Language", 2nd ed.,
Kernighan, B.W., and Ritchie, D.M.,
Prentice Hall Software Series, 1988.
IBSN 0-13-110362-8.
```

All constants and conversion factors are here defined as double precision values. Where single precision calculations are done in a situation where processing speed is of concern, for example within the inner loop of an expensive algorithm, a separate single precision variable should be defined for use within the loop.

### 2.6.2 Primes

The `Primes` class provides some prime number operations.

### 2.6.3 Hashing

`ExtendHash`<`T`> is a templated hash class. It uses extendible hashing for storing objects of type `T`.

`Bucket` is a fixed size array of `T`. The size of the bucket is user definable; default is 4.

`ExtendHash` contains a number of buckets which grows as objects are added. The user provides the hash function which is tuned for the particular application and `T`. The hash function should return uInt.

The user must provide the operators `==` and `!=` for the type `T`.

### 2.6.4 Interpolation

Set of templated letter/envelope classes for interpolation.

### 2.6.5 Math. Functions

AIPS++ provides a templated letter/envelope set of classes for packaging of specific single dependent variable functions.

The `MathFunc` class is the abstract base class for 1–dimensional math functions.

Actual math functions are then an inherited class from the base class. This approach allows one to define actual function values for each derived class. Then, one can pass a generic `MathFunc` pointer to other objects, but the other objects will still get function values from the actual inherited function.

By defining each math function as an object, we can place parameters which will not change from one call to the function value to another in the class definition and they only have to be initialized once.

## 2.7 GNU Math. Utilities

### 2.7.1 Random Numbers

AIPS++ provides a set of random number classes.

They are essentially just a concatenation of:

| | | | |
|---|---|---|---|
| `ACG.h` | `Geom.h` | `NegExp.h` | `Random.h` |
| `Binomial.h` | `HypGeom.h` | `Normal.h` | `RndInt.h` |
| `DiscUnif.h` | `LogNorm.h` | `Poisson.h` | `Uniform.h` |
| `Erlang.h` | `MLCG.h` | `RNG.h` | `Weibull.h` |

from `libg++--2.2`. Random number classes written by Dirk Grunwald (`grunwald@cs.uiuc.edu`) and modified for use in AIPS++ .

## 2.8 "Simple" Containers

### 2.8.1 Bit Vector

`BitVector` is a class of bit vectors with variable size.

A variable utilized as a discrete collection of bits is referred to as a bit vector. Bit Vectors are an efficent method of keeping True/False information on set of items or conditions.

### 2.8.2 Block

`Block<T>` is a simple templated 1–D array class. Indices are always 0–based. For efficiency reasons, no index checking is done unless the preprocessor symbol `AIPS_ARRAY_INDEX_CHECK` is defined. `Block<T>`'s may be assigned to and constructed from other `Block<T>`'s. As no reference counting is done this can be an expensive operation, however.

The net effect of this class is meant to be unsurprising to users who think of 1–D arrays as first class objects. The name "Block" is intended to convey the concept of a solid “chunk” of things without any intervening “fancy” memory management, etc. This class was written to be used in the implementations of more functional `Vector`, `Matrix`, etc. classes, although it is expected `Block<T>` will be useful on its own.

The `Block` class should be efficient. You should normally use `Block` in preference to C–style arrays, particularly those obtained from `new`. `Block<T>` is not derived from `Cleanup` so if an exception is thrown the storage won’t be reclaimed: `Block`’s should be used to build other classes which are themselves reclaimed.

### 2.8.3 General Iterator support

The `NoticeSource` class provides a mechanism for keeping all of the observers of an object up to date. Each time a change is made to the observed object, the observers are notified of the change. When the observers are updated, they can choose whether to update themselves or not.

Classes which have many other dependent objects which need to be updated should derive from this class, e.g. doubly linked lists which have dependent iterators.

### 2.8.4 Linked Lists

The `Link` class provides a minimal singly linked list implementation, while the `Dlink` class, derived from `Link`, provides a minimal doubly linked list implementation. In these classes, all of the work is performed by the constructor. The classes do not keep track of the head of the list; this is left to the user of the class. These classes can be thought of as the “nodes” of a linked list, but conceptually each of the nodes is a list itself.

Although they are functional linked list implementations, these classes will typically not be used by the average user. Instead, various kinds of linked list classes have been built using these classes.

In particular, the following user–level classes have been built:

- `List`: a singly linked list.
- `Dlist`: a doubly linked list.

One uses the associated iterator classes to traverse the lists:

- `ListIter`
- `ConstListIter`

- `DlistIter`
- `ConstDlistIter`

The *ListIter* classes traverse `Lists`, while the *DlistIter* classes traverse `Dlists`.

The *Const* classes do not allow the underlying value in the list to be modified.

### 2.8.5 Stacks

This class, `Stack<t>`, defines an implementation of a stack using the singly linked list primitive, `Link<t>`. It has the standard push and pop operations.

### 2.8.6 Maps

The `Map` class is the abstract base class for several "Map" classes which implement associative arrays. The classes derived from `Map` are:

- `ListMap`: a `Map` with list ordering / operations.
- `OrderedMap`: a `Map` with ordered keys.

There is also a `SimpleOrderedMap` class, which is not derived from `Map`.. `SimpleOrderedMap<key,value>` is a templated class. It is similar to `OrderedMap<key,value>`, but lacks its sophisticated iterator capability. Instead, iteration can be done using the `getKey` and `getVal` functions with a simple index.

## 2.9 High Level "Containers"

### 2.9.1 Arrays

`Array<T>` is a templated, N–dimensional, Array class. The origin is variable, but by default indices are zero–based. This Array class is the base class for specialized `Vector<T>`, `Matrix<T>`, and `Cube<T>` classes.

Indexing into the array, and positions in general, are given with `IPosition` (essentially a vector of integers) objects. That is, an N–dimensional array requires a length–N `IPosition` to define a position within the array. Unlike C, indexing is done with (), not []. Also, the storage order is the same as in FORTRAN, i.e. memory varies most rapidly with the first index.

Aside from the explicit `reference()` member function, a user will most commonly encounter an array which references another array when he takes an array slice (or section). A slice is a sub–region of an array (which might also have a stride: every nth row, every mth column, . . . ).

The `Array` classes are intended to operate on relatively large amounts of data. While they haven't been extensively tuned yet, they are relatively efficient in terms of speed. Presently they are not space efficient – the overhead is about 15 words. While this will be improved (probably to about 1/2 that), these array classes are not appropriate for very large numbers of very small arrays. The `Block<T>` class may be what you want in this circumstance.

Element by element mathematical and logical operations are available for arrays (defined in `aips/ArrayMath.h` and `aips/ArrayLogical.h`). Because arithmetic and logical functions are split out, it is possible to create an `Array<T>` (and hence `Vector<T>` etc) for any type `T` that has a default constructor, assignment operator, and copy constructor. In particular, `Array<String>` works.

A tutorial for the Array classes is available.

### 2.9.2 Tables

The AIPS++ table system is designed to be the main data system used in the AIPS++ software. It is based on the ideas of Allen Farris.

A table consists of a number of keywords and columns. A column can be a filled one (i.e. actually stored in a file) or it can be a virtual one (i.e. calculated on the fly). Filled columns are stored using a storage manager. A table can have more than one storage manager; in fact, each column can have its own storage manager. A table can therefore consist of multiple files. The table name is the name of the main file.

Virtual columns are handled by so–called virtual column engines. These engine classes have to be written by the application programmer, since each virtual column will be calculated in a dedicated way.

A column can contain scalar values of any data type, arrays of any shape and any data type, or tables. The latter allows for a hierarchy of tables.

Only standard data types can be used in filled columns. These are: `Bool`, `uChar`, `Int`, `uInt`, `float`, `double`, `Complex`, `DComplex`, and `String`. Arrays can be direct or indirect. Direct arrays have to have the same shape in all cells of a column and are stored in the main storage manager data file. Indirect arrays can have varying shapes in the column cells and are stored in a separate file. Direct columns are meant for small, fixed sized arrays.

The table keywords make use of the keyword module (i.e. class `KeywordSet` and related classes).

## 2.10 High Level Utilities

### 2.10.1 Logging

This module contains the AIPS++ logging/history mechanism classes.

The `LogMessage` class is the means to providing a logging system that will allow selective messages to be stored or possibly replayed as GLISH scripts.

The `LogMessage` class is the class author's interface into the Log system. By "throwing" `LogMessage`s at regular intervals or important times, the programmer assures no gaps will exist in the run–time history. The coder simply creates a robust `LogOrigin` (if wished — the default `LogMessage` constructor fills in most of the important stuff.) This `LogOrigin` instance is then passed as an argument to the `LogMessage` constructor along with the pertinent `String` message. The global `Log(LogMessage)` function will then pass the message through to the external log containers.

The `LogtoTable` class creates a table of logs to be used later. The method is to create the `LogFilters` you want and add them to the `LogtoTable` via the inherited functions of `LogSink`.

The `Tables` module allows robust sorting and selection. These techniques can be used to select/scan/re–run any logged information from the `LogtoTable`'s `Table`. Additionally, the `Table`'s persistance is assured at destructor time. No log will be lost.

### 2.10.2 Visualize

The `Visualize` class is a class to do line plotting, histogram and contouring using PGPLOT.

### 2.10.3 Gridding

The `GridTool` class contains methods for gridding, or interpolating, an n–dimensional data point onto an n–dimensional grid. Reverse methods for de–gridding data are also supplied.

For a complete description of the methods associated with this class see note 158 "The AIPS++ GridTool Class"

#### 2.10.4 Fourier Transforms

The `DFTServer` class contains methods for doing n–dimensional Slow Fourier Transforms. (In practice, the maximum dimension is 3)

The `FFTServer` class contains methods for doing n–dimensional Fast Fourier Transforms. (In practice, the maximum dimension is 3).

At present the actual FFTs are done by FORTRAN functions which call the `FFTpack` FFT functions available from `netlib`.

The `FourierTool` class provides pack and unpack operations that manipulate Nyquist data. These tools may handle either real or complex numbers, and pack/unpack operations are provided for each.

Note: This class is defined as

```
template<class T, class S> class FourierTool
```

but the only legal instantiations are:

```
FourierTool<float, Complex>
FourierTool<double, DComplex>
```

For a complete description of the methods associated with these classes see note 157 "The AIPS++ FFTServer Class"

#### 2.10.5 Deconvolution

This module contains classes for deconvolvers using some form of the CLEAN algorithm (Hogbom, Clark, Steer–Dewdney–Ito, . . . ).

#### 2.10.6 Dummy Image

The `DummyImage` class is a very simple image for experimentation. It is deprecated in favor of the soon–to–be–released `Image` class.

### 2.11 Orphaned Code to be Picked Up

The code in this section is currently orphaned, i.e. its author no longer works on AIPS++ . It will be revived in the near future. While it does work, it is somewhat out–of–date with respect to the current thinking / state–of–the–art of AIPS++ . It will all be reviewed, and will be redesigned and rewritten where necessary.

### 2.11.1 ObjectID

The `ObjectID` class provides a unique identifier for each high level object.

### 2.11.2 Catalogs

`CatalogHook` is the class which encapsulates the interaction of high level objects with the catalog system.

Catalog behavior:

There are two catalogs, the `Active` catalog and the `File` catalog. An item may be in one only or in both. `CatalogHook` provides two methods, `update` and `downdate` for updating these catalogs. `update` is called when the actual `Table` is initialized and adds the current `CatalogHook` to the `Active` list. `downdate` is called by the destructor, removes the `CatalogHook` from the `Active` catalog, and adds the `CatalogHook` to the `File` catalog if a filename is present in the `Table` header. (If the `Table` is constructed with a `FileName`, that `Filename` is also placed in the `Table`'s header; if the `Table` is constructed with a `Filename`, the `Table` destructor writes the `Table` to disk before destruction.)

A third catalog, the `Handle` catalog, maps a `Handle` (an object's name as known to the user) into an `ObjectID`.

### 2.11.3 Measurement Model

The `MeasurementModel` class is the telescope measurement equation virtual base class.

It is the base class for all *mapping* objects for AIPS++ . At the moment it is very light weight, and literally does nothing. As the system evolves, all functionality which will be common for the mapping sub–system will go in here. Currently one can think of Cataloging, association and contact with UI/GUI happening in this class.

Built on top of `MeasurementModel` is the `IntMeasurementModel` class, which is the telescope measurement equation for Synthesis telescopes.

The purpose of this class is to pass the Telescope data through a measurement equation appropriate for an Interferometer in the continum mode. It makes one image (or image + dirtybeam) from the data given to it. The dimensionality for which it will work is determined by the dimentions of the Image (works as a 3D invertor/predictor if the supplied image is a cube). This, of course depends on `FFTServer` and `GridTool` being able to handle the case of various dimentions ("existential" design?).

Any selection of Freq/Poln etc. must be done outside this. It is hoped that wherever it is used, the over all design would be existential and `IntImgModel:Invert(...)` and `IntImgModel:Pridect(...)` would get called polymorphically. All specializations of `MeasurementModel` would be inherited from `MeasurementModel` class, which is the base class for the mapping tree of AIPS++ .

All selections on the UVData given to this will be done by a wrapper around this level (i.e. at the UVMAP level ) — this level worries about the forward/revers transforms that represent the telescope — and worries about that alone!

#### 2.11.4 Telescope Model

The Telescope Model describes (or models) the state of the physical telescope at a given point during the data calibration process. It is responsible for determining calibration parameters, via the `solve` method, and applying corrections to raw data in the Measurement Set (MS), via the `apply` method.

The process of calibration involves successive iterations of `solve` and `apply`, until the Telescope Model is judged to accurately describe the state of the telescope during the observation.

The Telescope Model is composed of a number of Telescope Components (TCs) which model platform characteristics, environmental parameters, receptor characteristics, etc. A common representation of a Telescope Component is a calibration parameter solution table; but the representation is not limited to calibration parameter solutions.

Calibration involves the addition, deletion, and modification of the Telescope Components until a desired model is found. The role of the Telescope Model is to

- Represent a related set of Telescope Components (TCSet),
- Allow the specification of Components to "solve for" (via the `SolveSet`), and
- Allow the specification of order in which to "apply" the Telescope Components to the MS (via the `ApplySequence`).

A `solve` operation involves the invocation of the `solve` method for the chosen set of Telescope Components. Likewise, an `apply` involves the invocation of the `apply` method for the chosen set of Telescope Components in the given order.

A `solve` operation requires that a "skeleton" Telescope Component exist, be attached to the Telescope Model, and be selected in the SolveSet before `TM.solve` is invoked.

## 2.12 New Code Soon to be Released

There is quite alot of code which is currently under development and which will be released in the near future. The following sections provide brief descriptions.

### 2.12.1 Queue

A `Queue` class.

### 2.12.2 MaskedArray

`MaskedArray` is a class for masking elements of an `Array` while performing operations on that `Array`.

A `MaskedArray` is an association between an `Array` and a mask. The mask selects elements of the `Array`. Only elements of the `Array` where the corresponding element of the mask is True are defined. Thus, operations on a `MaskedArray` only operate on those elements of the `Array` where the corresponding element of the mask is True.

A `MaskedArray` should be thought of as a manipulator for an `Array`, analogous to an iterator. It allows one to perform whole `Array` operations on selected elements of the `Array`.

### 2.12.3 Lattice

A `Lattice` base class which encapsulates finite linear, rectangular, or hyper-rectangular data structures, plus associated `LatticeIterator` classes for iterating through a `Lattice` conveniently.

### 2.12.4 PagedArray

A `PagedArray` class, which is derived from `Lattice`, for holding arrays larger than memory.

### 2.12.5 Image

An `Image` which is a `PagedArray` with associated coordinates.

### 2.12.6 Unit

`Unit` classes for physical measurements.

### 2.12.7 Linear Least Squares Fitting

Classes for general linear least squares fitting.

### 2.12.8 Single Dish

Classes for position-switched calibration and on-the-fly mapping of single dish data.

### 2.12.9 Graphics

Graphics classes (based on Karma graphics) for the manipulation of images and other graphical items.