AST: A library for modelling and manipulating coordinate systems☆

David S. Berry[a,b,*], Rodney F. Warren-Smith[c], Tim Jenness[a,d]

[a] *Joint Astronomy Centre, 660 N. A'ohōkū Place, Hilo, HI 96720, USA*
[b] *East Asian Observatory, 660 N. A'ohōkū Place, Hilo, HI 96720, USA*
[c] *RAL Space, STFC Rutherford Appleton Laboratory, Harwell Oxford, Didcot, Oxfordshire OX11 0QX, UK*
[d] *LSST Project Office, 933 N. Cherry Ave, Tucson, AZ 85721, USA*

Abstract

This paper describes the Starlink AST library.

Keywords:
WCS, data models, Starlink

1. Introduction

The Starlink AST library (Warren-Smith and Berry, 2013, ascl:1404.016) provides a generalised scheme for modelling, manipulating and storing inter-related coordinate systems. Whilst written in C, it has bindings for several other languages including Python, Java, Perl and Fortran. It has specialised support for many of the coordinate systems and projections commonly used to describe astronomical World Coordinate Systems (WCS), including all those described by the FITS-WCS standard, plus various popular distortion schemes currently in use. However, it is not limited to WCS, and may be used in any situation requiring transformation between different coordinate systems.

Unlike FITS-WCS, which supports only a relatively small set of prescribed transformation recipes reflecting the coordinate transformations within an optical telescope, AST allows arbitrarily complex transformations to be constructed by combining simple atomic transformations in series or in parallel. This allows a much wider range of transformations to be described than is possible using FITS-WCS, and so can accommodate a wider range of data storage forms without the need to re-grid the data.

AST was released internally in 1997 (Warren-Smith and Berry, 1998, included in "Twenty Years of ADASS"). Since then it has been in continuous use within the Starlink Software Collection (Currie et al., 2014, ascl:1110.012) and is also used by various other major astronomical software tools such as DS9 (Joye and Mandel, 2003, ascl:0003.002). Interest in flexible schemes for representing inter-related coordinate systems has increased recently[1], and so it seems an appropriate time to review the lessons learned from AST.

This paper first presents an account of the historical issues that drove the initial development of AST, together with the reasoning behind some of the design decisions, and then presents an over-view of the more important aspects of the data model used by AST.

2. Historical Perspective

2.1. Initial Problems

The first public release of the AST library was in 1998 (Draper, 1998; Lawden, 1998) but some of the underlying concepts date from the late 1980s, when the Starlink Project was designing its NDF data format for gridded astronomical data (Jenness et al., 2015). There was clearly a need to relate positions within gridded data, using coordinates based on pixel indices, to real-world positions on the sky, wavelengths in a spectrum and so on. These non-pixel coordinate systems are now generally known as world coordinate systems (WCS).

Calibration of spectra, for example, was commonly performed by fitting a polynomial to express wavelength as a function of pixel position and then either storing the polynomial coefficients, or tabulating the polynomial value at each pixel centre. While not completely general, the latter option was an acceptable solution and was adopted as part of the Starlink data format. An array giving the central wavelength at each pixel was stored as the `AXIS` component in the NDF data structure and did good service in spectroscopic applications. It was also possible, in a simple minded way, to attach an `AXIS` array to each dimension of an image or any gridded data set of higher dimensionality. This allowed each of its axes to be calibrated in terms of world coordinates.

This approach was adequate if the axes represented independent quantities (like wavelength and position for a long-slit spectrum), but did not suffice if the axes were inter-independent. Unfortunately, in the common case of celestial coordinates (such as Right Ascension and Declination), the axes are almost always inter-dependent. This is because the sky is

☆This code is registered at the ASCL with the code entry ascl: 1404.016

*Corresponding author
Email address: `d.berry@eaobservatory.org` (David S. Berry)

[1] For instance, in the discussions about possible successors to the FITS format (Mink et al., 2015), and within the Astropy project (Astropy Collaboration, 2013)

essentially spherical and its coordinates are therefore naturally curvilinear when projected into two dimensions. This interdependence is a common feature of world coordinate systems in practice, so a solution was clearly needed that addressed it properly.

The Flexible Image Transport System (FITS; Greisen and Calabretta, 1995; Wells et al., 1981), at that time, addressed the issue in a better but still rudimentary way. In essence, it stored a physical pixel size (e.g., in seconds of arc), allowed for a linear scaling of an image (typically to allow for the position angle rotation of the telescope) and then projected it on to the celestial sphere using one of a defined set of map projections. This representation was clearly based on a model of a physical telescope and how it imaged an observed region of the sky in its focal plane.

While successfully accommodating the curvilinear nature of sky coordinates, this FITS approach was still limited in many ways. In essence, it defined a small set of functional forms (based on map projections) through which pixel coordinates could be mapped on to celestial coordinates and back again. However, if the actual relationship between pixel coordinates and world coordinates didn't correspond to one of these functional forms, then it wasn't possible to use FITS to store the coordinate information[2].

For instance, if astronomical instrumentation were to use a novel map projection, if arbitrary instrumental distortions were present or if the data were re-gridded into a non-physical space, then the FITS approach would fail. It also had limited support for high-accuracy astrometry, where the departure of the sky from a perfect sphere, for a variety of reasons, has to be taken into account. In addition, there are many other non-celestial world coordinate systems that one might use (involving energy, velocity, time, frequency, *etc*.) that no contemporary system could represent adequately.

Unfortunately, this list of limitations only scratches the surface of the problem as it was perceived at the time. Other considerations, such as the time-dependent relationship between non-inertial celestial coordinate systems, the dependence of apparent positions on the position and velocity of the observer (and also on the wavelength of observation and atmospheric conditions) and periodic revisions to the fundamental definitions of celestial coordinate and time systems would all have to be accommodated, as would numerous other issues specific to particular domains (celestial coordinates, time systems, radial velocities, wavelength/energy, *etc*.). This was several years before the FITS community commenced work on what was eventually to become the current FITS-WCS standard.

2.2. *TRANSFORM*

In the late 1980s, no immediate and general solution to these problems could be seen. Recognising the limitations in the FITS approach, however, the Starlink Project decided to take a hard line and to omit completely any component dedicated to world coordinate systems from its new NDF data format. Instead, this *astrometry extension* (from which the name AST is derived) was to be added at a later date when a suitable solution had been formulated.

This decision was undoubtedly strongly influenced by Patrick Wallace's presence in the Project and the major work he had done on the SLALIB library (Wallace, 1994, ascl:1403.025) to encapsulate best-practice in astrometric calculations (and also in other domains such as time systems). Discussion within the Project rapidly convinced us that if we adopted the FITS approach as it existed at the time, we would cut ourselves off from the proper rigorous treatment of astrometric data that is needed for the highest accuracy.

Consequently, a pilot project was conducted to explore alternative approaches. The most important limitation of the FITS approach was felt to be the use of a fixed set of functional forms (map projections) each of which was associated with a small fixed set of parameters. This simplified storing the information in 80-character FITS *header cards*, but clearly the set of functional forms that might ultimately be needed was much larger than had been recognised. Adding new ones might become a never-ending project and that, in turn, raised the prospect of continually upgrading all software that had to read and process FITS headers and handle coordinate systems.

The alternative approach that we explored was to write an expression parser that would accept sets of arithmetic expressions similar to those used in Fortran and C, along with the usual set of mathematical functions. Together with a method of passing named parameter values into these expressions, this greatly increased the set of functional forms that could be represented. The expressions themselves (encoded as character strings) and the associated parameter values could easily be stored in astronomical data sets. Typically, one set of expressions would relate pixel coordinates to world coordinates (e.g., sky coordinates) and a second, optional, set would define the inverse transformation. The expression syntax was powerful enough to represent a wide range of map projections plus many other transformations into alternative world coordinate systems.

A processing engine was also provided that could use the stored expression data to transform actual coordinate values.

A library implementing this, called TRANSFORM, was released in 1989 (Lawden, 1989; Warren-Smith, 1989). It stored its data (the expressions and parameters) in Starlink's Hierarchical Data Format (HDS; Jenness, 2015; Lupton, 1989; Warren-Smith et al., 2008) and was thus able to integrate with the Starlink NDF data format to attach arbitrary world coordinates to gridded astronomical data sets.

2.3. *TRANSFORM Lessons*

Ultimately, TRANSFORM turned out not to be a full solution to the WCS problem and did not become part of the NDF data format[3]. It was, however, used for two initially unforeseen purposes which turned out to be very significant:

[2] Unless the data was first re-gridded into a form supported by FITS.

[3] Although it it was the precursor of the `MathMap` class in AST.

1. Associating coordinate systems with plotting surfaces in a "graphics database" (see e.g., Eaton and McIlwrath, 2013). This allowed plotting applications to store a coordinate system for (say) a graph plotted in logarithmic coordinates so that those coordinates could later be recovered from the position of a cursor. This demonstrated that plotting was a major application area for this type of technology, especially when using curvilinear coordinates such as Right Ascension and Declination which are notoriously difficult to handle properly with standard plotting software.
2. Transformation and combination of bulk image data using general arithmetic expressions (as an alternative to combining images using a manual sequence of add/subtract/multiply/divide and similar applications). This showed that (a) the approach could easily be efficient enough to handle large data sets and (b) the data values in an image were just another coordinate that could be transformed into different representations (logarithmic, different units, *etc*.) in much the same way as its axes.

With these insights, it was clear that the ideas behind TRANSFORM had potential, but some serious deficiencies had also emerged:

- Arithmetic expressions, while fairly general, could not easily cope with coordinate transformations that required iterative solution, nor with discontinuous transformations, nor with look-up tables or a variety of other computational techniques. While arithmetic expressions provided a valuable increase in the flexibility of coordinate transformations, clearly other classes were still needed.

- It was a major problem for the average writer of astronomical software to formulate the required coordinate expressions correctly even when dealing with quite simple sky coordinate systems. The core of this issue is that celestial coordinate systems are rather complex and a good deal of specialist knowledge is needed to formulate even simple cases correctly. Clearly a better solution would be to encapsulate this knowledge in the WCS software and provide a simpler API that dealt only in high-level concepts.

- For high accuracy work, further complex calculations arise. These are related, for example, to atmospheric refraction and special & general relativistic effects (like the observer's motion and the sun's gravity). These require the use of a dedicated library of astrometric functions and cannot in practice be handled by simple expressions. They also require additional data about the observing context (time, position, velocity, wavelength, *etc*.) and any practical solution must define how these are stored and processed.

- TRANSFORM had no ability to store additional information about data axes, such as labels and units.

- It became clear that coordinate transformations frequently needed to be combined, for example by applying one transformation after another, and that this process was often inefficient. The key to better efficiency lay in knowing more about each transformation, like whether it was linear or had a variety of other properties. With this information it was possible to merge (or cancel out) consecutive transformations for better efficiency. TRANSFORM had a rudimentary system for encoding this information, but it was not really up to the task.

- Tying WCS software to a particular (Starlink) data system was a mistake and limited the uses to which it could be put. It would clearly be better if the data could be encoded (serialised) in alternative ways to make it data-system agnostic. The same agnosticism should also apply to other likely dependencies, like graphics systems and error reporting. The ability to implement these services in alternative ways would be especially important when designing graphical user interfaces that processed WCS information.

2.4. *Developments in FITS WCS*

At about the same time, the wider FITS community also came to recognise some of the limitations of WCS handling within FITS, and in 1992 work commenced on a new standard for storing WCS information within FITS files. However, in view of the "once FITS, always FITS" principle, that work consisted mainly in formalising and extending existing practices. So for instance, new keywords were defined to store the extra meta-data needed for a complete description of a celestial coordinates system, and new projection types were added, but the basic model remained unchanged. The new standard still required that the transformation be split into three components applied in series; an affine transformation that converts pixel coordinates into *intermediate world coordinates*, a spherical projection that converts these into *native spherical coordinates*, and a spherical rotation that converts these into the final world coordinate system.

In view of the decision to stay with this rigid and restrictive model, and in view of expected length of time needed to agree a new standard[4], the Starlink project decided in early 1996 to develope its own WCS system, informed by the earlier experiments with TRANSFORM, rather than adopt the new FITS standard.

2.5. *AST Principles*

One of the first decisions was to separate the representation of a coordinate system (that we called a Frame) from the computational recipe that transforms between coordinate systems (a Mapping). From the TRANSFORM experience we knew we would need multiple classes of both these data types, all of which would need to support the same basic operations, but each having its own specialisation. The correspondence with subclassing in object-oriented (OO) programming was irresistible and the decision to use an OO design immediately followed.

[4] An expectation that was justified when the standard was finally published in 2002.

This raised the issue of an implementation language. We planned to use the SLALIB library for astrometric calculations[5]. This had been developed with extreme portability in mind and had recently been re-written in ANSI C. We didn't want to compromise this portability, so decided also to work in strict ANSI C and to minimise software dependencies as much as possible. This meant providing portable interfaces to facilities that were intrinsically less portable, such as data file access, plotting, error reporting, *etc*. and providing simple implementations that users could re-write if necessary.

Deciding to write an OO system in a non-OO language took considerable thought. We needed to provide a Fortran-callable interface but, at the time, the portability of C++ code was quite limited if one needed to call it from Fortran, so that route was unattractive. Eventually, we were guided by the approach described by Holub (1992) for handling objects in C and were able to hide the detail from users using pre-processor macros.

One consequence of this is that users cannot easily create new sub-classes from AST objects without learning the internal conventions that it uses. At the time, this was seen as something of an advantage. The library is intended for data interchange and creating new sub-classes would inevitably allow persistent objects to be created that other users could not access. However, with hind-sight a more open architecture may have encouraged involvement from a wider user-base. [6].

As noted previously, we wanted the AST API to deal in high-level concepts and to hide as much specialist detail as possible from the user. This principle arose from considering the complex calculations involved in handling celestial coordinates and time systems. However, we soon realised that two other areas were similarly complex and could benefit from the same approach.

The first area was graphics. Plotting in curvilinear coordinates is a complicated business if one wants to handle all the corner cases correctly. Plotting and labelling celestial coordinate axes, for example, presents many problems; especially near the poles of an all-sky projection. It is made even harder if the projection contains discontinuities. But the high-level concepts involved in such plotting (coordinate systems and the mappings between them) are such a natural fit with other AST concepts that it seemed obvious to implement a class of coordinate system that is specialised for graphics. The high-level operations it supports would then hide the details of the complex and generalised plotting algorithms involved.

The second area is an aspect of data storage – namely the handling of FITS header cards. While the AST library could provide ways to serialise its own data transparently, possibly in multiple ways, it also needed to inter-operate with FITS. WCS data in FITS data files is stored in a series of 80-character header cards and, over the years, the number of different ways the information can be stored in these headers has grown. The complexity involved is now considerable (see e.g., Thomas et al., 2015). Again, detailed specialist knowledge is needed to extract this information reliably and to write it back (possibly modified) in a form that gives other FITS-handling software a chance of using it while not conflicting with the many other FITS headers typically present.

For a user of the AST library, we wanted this process of accessing FITS headers to appear as much like a simple read/write operation as possible, with all the implementation details hidden. This requirement arose from more than simply ease-of-use. FITS header conventions (many of them informal) are in constant flux and if these details are embedded in applications programs, those programs must constantly receive attention if they are to remain up to date. Embedding all these details in the AST library allows the problem to be addressed in one place and by someone with the necessary expertise.

FITS header handling has proved one of the most complex areas to tame in AST. But introducing the concept of a *destructive read* (which reads WCS data from FITS headers and simultaneously deletes the relevant headers) has made it possible to write applications with very little code that have completely general handling of FITS WCS headers.

3. The AST Data Model

The most basic principle behind the AST data model is a clear distinction between a *transformation* and a *coordinate system*.

A *transformation* is a mathematical recipe for converting a numerical input vector into a corresponding numerical output vector. The transformation itself has no knowledge of what the values within these vectors represent, other than that each one constitutes a position within some unspecified N-dimensional space[7].

A *coordinate system* is a collection of meta-data describing a set of one or more axes. This will include:

- the number of axes (i.e., dimensionality of the space)
- the physical quantity described by each axis
- the units used by each axis
- the geometry of the space that they describe (flat, spherical, *etc*)
- the nature of the coordinate system (Cartesian, polar, *etc*)
- other meta-data that may be needed to specify the coordinate system fully.

These two concepts are encapsulated within two separate classes in the AST data model: the *Mapping* class and the *Frame* class. This separation underlines the fundamental difference between the two main requirements of any coordinate handling system:

[5] A later version of AST eventually replaced SLALIB with SOFA and PAL.

[6] For Mappings, where this issue is most relevant, the problem has been mitigated in a controlled way by the IntraMap class that allows separately-compiled code to be imported into the library.

[7] The dimensionality of the output space need not be the same as that of the input space.

1. knowing *how* to convert numerical positions from one coordinate system to another.
2. knowing *what* those coordinate systems represent.

As an example of how this distinction is applied in practice, the *pixel size* of a typical 2-dimensional image of the sky is *not* considered to be a property of the (RA,Dec) Frame, since it is determined by the nature of the transformation from pixel coordinates to (RA,Dec) coordinates, rather than being an intrinsic property of the (RA,Dec) coordinate system itself.

The two base classes, *Mapping* and *Frame*, are extended to create a wide variety of sub-classes, each of which describes a specific form of transformation or coordinate system. New sub-classes can be added as required and slot naturally into the existing infra-structure provided by the rest of AST.

In addition, this separation into two "orthogonal" classes makes it easy to create complex compound objects from simple component objects. For instance, multiple Mappings can be combined into a new object, and the resulting object will itself be a Mapping. Likewise, multiple Frames can be combined into a new object, and the resulting object will itself be a Frame[8].

However, at some point these two classes need to be brought together to provide a complete description of a set of related coordinate systems. The *FrameSet* class is used for this purpose. A *FrameSet* encapsulates a collection of two or more Frames, with the Mappings that describe the transformation between the corresponding coordinate systems. The simplest FrameSet contains two Frames, together with a single Mapping that describes how to convert positions between these two Frames (see Fig. 1).

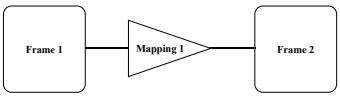

Figure 1: A FrameSet that describes two coordinate systems and the transformations between them. The Mapping's *forward* transformation transforms positions in *Frame 1* to the corresponding position in *Frame 2*. The Mapping's *inverse* transformation transforms positions in *Frame 2* to the corresponding position in *Frame 1*.

More complex FrameSets can be created that describe the relationships between multiple Frames in the form of a tree structure (see Fig. 2).

A selection of the more significant methods and properties of these three classes are described in appendix Appendix A.

3.1. Transformations and Mappings

Within AST, most *Mappings* encapsulates two transformations - one is designated as the *forward* transformation and the other as the *inverse* transformation. When a Mapping is used to transform a set of positions, the caller must indicate if the forward or inverse transformation is to be used. The *forward transformation* converts positions within the input space of the Mapping into corresponding positions within the output space, and the *inverse transformation* converts positions within the output space of the Mapping into corresponding positions within the input space. A Mapping can be *inverted*, which results in the two transformations being swapped.

[8]For instance, an 2-dimensional (RA,Dec) Frame can be combined with a 1-dimensional wavelength Frame to create a 3-dimensional (RA,Dec,Wavelength) Frame.

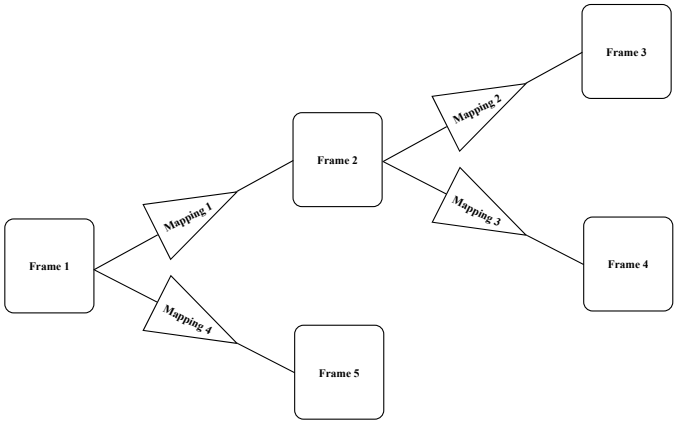

Figure 2: A FrameSet that describes five inter-related coordinate systems and the transformations between them.

For most classes of Mapping, the inverse transformation is a genuine mathematical inverse of the forward transformation. However, this is not an absolute requirement, and there are a few classes of Mapping where this is not the case (for instance the *PermMap* class, in which the axes of the output space are a permuted subset of the axes of the input space). In addition, the Mapping class does not require that *both* transformations are defined. For instance, the *MatrixMap* class, which multiplies each input vector by a specified matrix to create the output vector, will only have an inverse transformation if the matrix is square and invertable.

3.1.1. Classes of Mapping Provided by AST

AST provides many classes of Mapping that implement a wide range of different transformations. Most of these are *atomic* Mappings that implement a specific numerical transformation and, if possible, its inverse. But some are *compound* Mappings that combine together other Mappings (atomic or compound) in various ways to create a more complex Mapping. A compound Mappings does not define its own transformations, but instead inherits the transformations of the individual component Mappings which it encapsulates.

The most significant atomic Mapping classes are:

UnitMap: Copy positions from input to output without any change.

WinMap: Transform positions by scaling and shifting each axis.

ZoomMap: Transform positions by zoom all axes about the origin.

ShiftMap: Translate positions by adding an offset to each axis.

MatrixMap: Transform positions using a matrix.

PolyMap: A general N-dimensional polynomial transformation.

PermMap: Transform positions by permutating and selecting axes.

LutMap: Transform 1-dimensional coordinates using a look-up table.

MathMap: Transform coordinates using general algebraic mathematical expressions.

WcsMap: Implements a wide range of spherical projections.

SlaMap: Transforms positions between various celestial coordinate systems.

SpecMap: Transforms positions between various spectral coordinate systems.

SphMap: Map 3-d Cartesian to 2-d spherical coordinates.

TimeMap: Transforms positions between various time coordinate systems.

PcdMap: Apply 2-dimensional pincushion/barrel distortion.

DssMap: Map points using Digitised Sky Survey plate solution.

GrismMap: Models the spectral dispersion produced by a grism.

IntraMap: Map points using an externally supplied transformation function.

SelectorMap: Locates positions within a set of Regions (see section 3.1.2).

The most significant compound Mapping classes are (see section 3.1.2 for a more detailed explanation of some of these classes):

CmpMap: Combines two encapsulated Mappings in series or parallel.

SwitchMap: Selects from a set of alternate Mappings depending on the location of each input position.

TranMap: Use the forward transformations from one Mapping and the inverse transformation from another.

RateMap: A Mapping that represents an element of the Jacobian matrix of an encapsulated Mapping.

All classes of Mapping are immutable. That is, a Mapping cannot be changed once it has been created. This is unlike Frame and FrameSet objects, which *can* be changed.

3.1.2. Compound Mappings

The CmpMap class is a subclass of Mapping that encapsulates two other Mappings either in series or in parallel. Either or both of the two encapsulated Mappings can itself be a CmpMap, allowing arbitrarily complex Mappings to be created.

In a *series* CmpMap, each input position is transformed by the first component Mapping, and the output from that Mapping is then transformed by the second component Mapping. Consequently, the output dimensionality of the first Mapping must be the same as the input dimensionality of the second Mapping.

In a *parallel* CmpMap, the input space is split into two subspaces. The first component Mapping is used to transform the axis values corresponding to the first subspace, and the second component Mapping is used to transform the axis values corresponding to the second subspace. Thus the input dimensionality of the CmpMap is equal to the sum of the input dimensionalities of the two component Mappings, and the output dimensionality of the CmpMap is equal to the sum of the output dimensionalities of the two component Mappings.

The SwitchMap class is a subclass of Mapping that allows a different transformation to be used for different regions within the input space.

Each SwitchMap encapsulates any number of other Mappings, known as "route" Mappings, and one "selector" Mapping. All of these Mappings must have the same input dimensionality, and all the route Mappings must have the same output dimensionality. The selector Mapping must have a one-dimensional output space.

Each input position supplied to the SwitchMap is first transformed by the selector Mapping. The scalar output from the selector Mapping is used to index into the list of route mappings. The selected route mapping is then used to transform the input position to generate the output position returned by the SwitchMap.

There is a specialised subclass of Mapping, the SelectorMap class, that is designed specifically to fulfil the role of the selector Mapping within a SwitchMap, but in principle *any* suitable form of Mapping may be used. The SelectorMap class encapsulates several Regions (see section 3.4) and returns an output value that indicates which of the Regions (if any) contained the input value. Thus, the SwitchMap would typically contain one route Mapping for each of the Regions contained within the SelectorMap.

3.1.3. Simplification

There are a wide range of possible transformations that could potentially be applied to a data set during analysis. These including simple things such as rotation, scaling, shear, *etc*., but could in principle include more complex transformations such as re-projection, dis-continuous "patchwork" transformations, or even transformation using a general algebraic expression. A coordinate handling system should make it possible for a user to apply an arbitrary set of such transformations in series to a data set, without losing track of the coordinates of each data point. With a prescriptive scheme such as FITS-WCS this would require each transformation to locate the appropriate component

of the FITS-WCS pixel to world coordinate mapping, and modify the corresponding headers in a suitable way. This is often a difficult, if not impossible, task. Within AST, the chaining of transformations is accomplished simply by creating a Mapping that describes each new transformation and concatenating it with the existing pixel to world coordinate mapping by creating a new CmpMap (see section 3.1.2).

However, by itself this can lead to Mappings that becoming increasingly complex as transformations are stacked on top of each other. This is a problem because it leads to:

1. slower evaluation of the total transformation,
2. less accurate evaluation of the total transformation, and
3. more room being needed to store the total transformation.

To avoid this, the Mapping class provides a "astSimplify" method that takes a potentially complex Mapping and simplifies it as far as possible. Doing such simplification in a general and effective manner is one of the most difficult challenges faced by the AST model, but experience has shown that the current scheme implemented in AST handles most cases sufficiently well. The steps involved in simplification depend on the nature of the component Mappings in the total CmpMap. Each class of Mapping provides its own rules that indicate when and how it can be simplified, or combined with an adjacent Mapping in the chain. To illustrate the principle, some of the simplest examples include,

1. any Mapping can be combined with its own inverse to create a UnitMap,
2. UnitMaps can be removed entirely,
3. adjacent MatrixMaps in series can be combined using matrix multiplication to create a single MatrixMap,
4. adjacent MatrixMaps in parallel can be combined to create a single MatrixMap of higher dimensionality (filling the off-diagonal quadrants with zeros).
5. adjacent ShiftMaps can be combined to form a single ShiftMap (either in series or in parallel).

The whole simplification process is managed by the implementation of the astSimplify method provided by the CmpMap class. It expands the compound Mapping into a list of atomic Mappings to be applied in series or parallel, and then for each Mapping in the list, invokes that Mappings protected "astMapMerge" method. This method is supplied with the entire list of atomic Mappings, and determines if the nominated Mapping can be merged with any of its neighbours. If so, a new list of Mappings is returned containing the merged Mapping in place of the original mappings. Once all atomic Mappings in the CmpMap have been checked in this way, the same process is repeated again from the beginning in case any of the changes that have been made to the list allow further simplifications to be performed. This process is repeated until no further simplifications occur.

There are in general multiple ways in which a list of Mappings (either series or parallel) can be simplified. Since each class of Mapping has its own priorities about how to merge itself with its neighbours, it is possible for the simplification process to enter an infinite loop in which neighbouring Mappings disagree about the best way to simplify the Mapping list. When a particular Mapping is asked to merge with its neighbours, it may make changes to the Mapping list that are then undone when a neighbouring Mapping is asked to merge with its neighbours. The astSimplify method takes care to spot such loops and to assign priority to one or the other of the conflicting Mappings.

3.1.4. Missing or Bad Axis Values

AST flags unknown or missing axis values using a special numerical value, known as the "bad" value. If an input position supplied to a Mapping contains one or more bad axis values, then in general all output axis values will be bad[9].

Mappings may also generate bad output values if the input position corresponds to a singularity in the transformation, or is outside the region in which the transformation is defined.

3.2. Frames and Domains

As described in section 3, each instance of the Frame class contains all the meta-data necessary to give a complete description of a particular coordinate system. Each Frame is associated with a specific *domain* as explained in the next section.

Unlike Mappings, the properties of a Frame can be changed at any time.

3.2.1. What is a Domain?

AST uses the word *domain* to refer to a physical (or abstract) space such as "time", "the sky", "the electro-magnetic spectrum", "the focal plane", "a pixel array". Points within such a space can in general be described using any one of several coordinate systems. For instance, any position on the sky can be described using ICRS coordinates, Galactic coordinates, *etc*. Similarly, positions in the eletro-magnetic spectrum can be described using frequency, wavelength, velocity, *etc*). Each subclass of Frame represents a specific domain, and in general will encapsulate all the metadata needed to create a Mapping between any pair of supported coordinate systems within its domain[10]. For instance, the *SkyFrame* class describes positions on the sky using a range of popular celestial coordinate systems. It extends the Frame class by adding various items of meta-data necessary to determine the conversion between any pair of supported celestial coordinate systems, the main items being the epoch of observation, and the reference equinox[11]. Thus given two SkyFrames describing, say, FK5 (RA,Dec) and Galactic (l,b), it is possible to create a Mapping between them based on the mata-data stored within the two SkyFrames. This Mapping can then be used to convert (RA,Dec) positions into equivalent (l,b) positions, or vice-versa. The process of creating this Mapping is implemented within the *astConvert* method of the base Frame class. For instance, in Python:

[9] One exception is that a parallel CmpMap may be able to generate non-bad values for some of its output axes.

[10] In retrospect, the Frame class should probably have been named *Domain* rather than *Frame*.

[11] The base Frame class includes the observer's geodetic position.

```
my_frameset = frame1.convert( frame2 )
```

will, if possible, generate a Mapping from `frame1` to `frame2`. The returned Mapping is encapsulated within a FrameSet that also includes copies of `frame1` and `frame2`.

Likewise, the SpecFrame class encapsulates all the meta-data needed to create Mappings between any pair of supported spectral coordinate system, including rest frequency, standard of rest, celestial reference position, etc.

The principle that each class of Frame contains all the meta-data and intelligence required to create a Mapping between any two coordinate systems within the Frame's domain, extends to compound Frames as well as atomic Frames, as described in the next section. The domain associated with a specific Frame is stored as a simple string such as "SKY", "SPECTRUM", "TIME", etc.

The base Frame class itself is slightly unusual in that it can be used to describe any generic domain, and is restricted to a single Cartesian coordinate system within that domain. The domain associated with a basic Frame is specified by the caller and can be any arbitrary string. Clearly, this restricts the usefulness of a basic Frame (compared to other more specialised classes of Frames) in that it is not possible to include any knowledge about multiple coordinate systems given the arbitrary nature of the domain. The only exception to this is that the basic Frame class knows how to convert between different dimensionally equivalent units. Thus the implementation of the astConvert method provided by the basic Frame class can generate a Mapping between two basic Frames if they have the same domain name, the same number of axes, and the axes have dimensionally equivalent units.

3.2.2. Classes of Frame Provided by AST

AST provides several classes of Frame that describe different specialised domains. As with Mappings, these can be divided into *atomic* Frames that describe a single specific domain, and *compound* Frames that combine together other Frames (atomic or compound) to create a Frame describing a domain of higher dimensionality. The atomic Frame classes are:

Frame: An arbitrary N-dimensional domain with a single Cartesian coordinate system.

FluxFrame: A 1-dimensional domain describing several forms of flux measurement systems (all measured at a single spectral position). In this case, the "axis value" represents a flux value.

SkyFrame: A 2-dimensional domain describing several celestial coordinate systems.

SpecFrame: A 1-dimensional domain describing several spectral coordinate systems.

DSBSpecFrame: Extends the *SpecFrame* class to describe dual sideband spectral coordinate systems.

TimeFrame: A 1-dimensional domain describing several time coordinate systems.

The following compound Frame classes are provided:

CmpFrame: Combines the axes from any two other Frames.

SpecFluxFrame : Combines a FluxFrame with a SpecFrame.

3.2.3. Compound Frames

The *CmpFrame* (compound Frame) class describes a coordinate system that combines the axes from two other Frames, in any order. The name of the domain associated with a CmpFrame is constructed automatically from the domain names of the two component Frames. For instance if a CmpFrame contains a SkyFrame and a SpecFrame, then its domain name will be "SKY-SPECTRUM".

As described above, each class of Frame provides a *astConvert* method that determines if a Mapping can be created from an instance of the specific Frame class to any other Frame. In general, this will only be possible between instances of the same Frame class (i.e., Frames that represent alternative coordinates systems within the same physical domain). However, the implementation of the astConvert method supplied by the CmpFrame class allows a Mapping to be created between a CmpFrame and either of the two component Frames. So for instance a CmpFrame describing the "SKY-SPECTRUM" domain could be matched with any of the following:

- Another CmpFrame describing the SKY-SPECTRUM domain.
- Another CmpFrame describing the SPECTRUM-SKY domain.
- A SkyFrame (i.e., a Frame describing the SKY domain).
- A SpecFrame (i.e., a Frame describing the SPECTRUM domain).

If the destination Frame has fewer axes than the source Frame, then the Mapping will contain a *permMap* - an atomic Mapping that permutes and selects a subset of its input axes[12].

So for instance, if a general purpose program reads the WCS from a data file of arbitrary dimensionality and wants to ask the question "can I determine the ICRS (RA,Dec) of each pixel position?" it can create a SkyFrame describing ICRS (RA,dec) and then use the astConvert method to see if a Mapping can be created from the WCS Frame read from the data file to the *template* SkyFrame describing the required coordinate system. This will allow ICRS (RA,Dec) to be determined for any 2-dimensional data file calibrated in any of the support celestial coordinates systems, and also any multi-dimensional data file that contains a pair of celestial coordinate axes.

[12] The inverse Mapping will either supply a specified fixed value, or a "bad" (i.e., missing) value for the "missing" axes.

3.3. FrameSets

The *FrameSet* class represents a network of inter-related coordinate systems in the form of a tree-structure in which each node is a Frame, with the nodes being connected together by Mappings (see Fig. 2). It is usual (although not required) for each Frame in a FrameSet to describe a different domain. A single FrameSet can be used, for instance, to provide a complete description of the WCS associated with a data array. In this case, one of the Frames represents pixel coordinates, and the other Frames represent a collection of alternative world coordinate systems. For instance, a FrameSet describing an image taken by a telescope may have three Frames describing pixel coordinates, focal plane coordinates and sky coordinates.

The FrameSet class provides a method, *astGetMapping*, to return the Mapping between any two nominated Frames. This may involve concatenating several Mappings if the two Frames are not directly connected to each other. For instance, if asked to returned the Mapping between Frame 1 and Frame 4 in Fig. 2, the *astGetMapping* method will retrieve Mappings 1 and 3 from the FrameSet, and combine them in series into a single *CmpMap* (compound Mapping).

The FrameSet class also provides a method, *astGetFrame*, to return any nominated Frame from a FrameSet.

New Frames can be added to an existing FrameSet at any point in the tree structure. To so so, the caller must provide a Mapping that maps positions from an existing Frame in the FrameSet to the new Frame.

Frames can also be removed from a FrameSet. If the Frame is a leaf node in the tree structure, then it is simply removed, together with the Mapping that connects it to its parent Frame. If the Frame is *not* a leaf node, the Frame is removed but the Mapping that connects it to its parent Frame is retained so that its child Frames can still be reached.

3.3.1. Base and Current Frames

Two of the most common operations provided by systems such as AST are 1) converting positions from one Frame to another, and 2) enquiring or using the properties of a Frame. Performing these two operations repeatedly would become tedious if each such operation involved separate calls to *GetMapping* or *GetFrame* to extract the required Mapping or Frame from the FrameSet. To avoid this, AST is implemented in such a way that the FrameSet class effectively inherits from both the Mapping class and the Frame class. This means that any Mapping method, or any Frame method, can also be used on a FrameSet.

The FrameSet class allows two nominated Frames to be flagged within each FrameSet. One is referred to as the "current Frame", and the other as the "base Frame"[13]. When used as a Frame, a FrameSet is equivalent of its current Frame. When used as a Mapping, a FrameSet is equivalent to the Mapping from its base Frame to its current Frame.

In practice, the coordinate system in which axis values are initially generated or obtained is usually used as the base Frame. For instance a cursor application generates positions initially in graphics screen coordinates, and a centroiding application generates them in pixel coordinates. The coordinate system in which positions are required by subsequent user (code or human) is usually used as the current Frame. This will often be some form of WCS, such as (RA,Dec), but could potentially be any of the Frames available in the FrameSet.

When AST creates a FrameSet from the WCS information in a FITS file, the base Frame represents pixel coordinates, and the the current Frame represents the primary WCS. The FrameSet may also contain other Frames representing any alternate axis descriptions stored in the FITS-WCS. This means that the FrameSet can be used as a Mapping from pixel coordinates to primary WCS, and can also be used as a Frame to determine the properties of the primary WCS.

The caller is free to select new base and/or current Frames at any time.

3.3.2. Integrity Restoration

Consider the simple case mentioned above where a FrameSet is used to describe the WCS in a 2-dimensional image. The FrameSet could for instance contain a pixel Frame as the base Frame and an (RA,Dec) Frame as the current Frame, connected together by a suitable Mapping. It is clearly possible to break the integrity of such a FrameSet, such that the Mapping no longer accurately describes the transformation from pixel coordinates to world coordinates. One obvious way in which this could be done is to change the current Frame so that it describes, say, galactic coordinates rather than (RA,Dec). The Mapping is left unchanged and so will still generate (RA,Dec) values, even though the FrameSet now claims that these are galactic coordinates[14].

In order to retain the integrity of the FrameSet, the Mapping must be replaced with one that generates the appropriate galactic coordinate values rather than (RA,Dec) values. One way in which this could be done is as follows, starting from the original unmodified FrameSet:

1. Create a copy of the current Frame (i.e., the (RA,Dec) Frame), and change the attributes of the copy so that it describes Galactic coordinates.
2. Use the *astConvert* method on the (RA,Dec) Frame to generate a Mapping from (RA,Dec) to galactic coordinates.
3. Add the galactic coordinates Frame into the FrameSet, using the above Mapping to connect it to the existing (RA,Dec) Frame.
4. Remove the original (RA,Dec) Frame.

The final FrameSet is unchanged in the sense that it still contains two Frames, but now the Mapping that connects them correctly generates the values described by the new current Frame (i.e., Galactic coordinates).

However, the above process is quite involved and prone to error, and so the FrameSet class itself provides generalised "integrity restoration" along the same lines, meaning that client

[13] The FrameSet class provides methods to set and retrieve the index of these two Frames.

[14] Note, Mappings are immutable and so the integrity of a FrameSet cannot be broken by making changes to a Mapping.

code is relieved of the responsibility. Whenever any change is made to the current Frame within a FrameSet, the FrameSet class first takes a copy of the original Frame and makes the requested change to the copy. It then uses the *astConvert* method to generate a Mapping from the old Frame to the modified copy. It then adds the modified copy into the FrameSet, using the Mapping returned by *Convert* to connect it the original unmodified Frame. It finally removes the original original Frame.

In summary, the integrity restoration system within the FrameSet class means that whenever the properties of the current Frame are changed within a FrameSet, the Mappings within the FrameSet are automatically modified accordingly.

However there are times when this may not be what is wanted. For instance, if a mistake is made when setting the properties of a Frame prior to adding it into a FrameSet, then it should be possible to correct those mistakes without causing changes to be made to the Mappings within the FrameSet. This is possible by first getting a reference to the Frame using the FrameSet's *astGetFrame* method, and then changing the Frame properties using that Frame reference, rather than the original FrameSet reference. Integrity restoration only occurs when the current Frame properties are changed via a FrameSet reference.

3.3.3. Searching a FrameSet for a Frame with Required Properties

As described earlier, the *astConvert* method defined by the Frame class attempts to find a Mapping between two arbitrary Frames[15]. Since the FrameSet class inherits the methods of the Frame class, the *astConvert* method can also be used on FrameSets. In this case the *astConvert* method will search through all the Frames in the FrameSet, starting with the current Frame, until a Frame is found for which a Mapping can be created. Since, in general, the Frames within a FrameSet all describe different domains, it is unlikely that more than one Frame will generate a Mapping, but the search order can be controlled by the caller in order to assign priority to specific Frames.

This allows code to search a FrameSet for a Frame that has specific properties. For instance, a Frame can be created with the required properties and then used as a "template" to search a FrameSet:

```
template = Ast.SkyFrame()
result = frameset.convert( template )
if result is None:
  print( "No celestial coord system found" )
```

In this example `frameset` is searched for a SkyFrame. If found, a FrameSet is returned in which the base Frame is the matching Frame from `frameset`, and the current Frame is a copy of `template`. In addition, the base Frame of `frameset` is set to indicate the matching Frame. If no matching Frame is found, a null reference is returned by *Convert*.

In the above example the specific celestial coordinate system represented by `template` was left unspecified when the SkyFrame was created. Consequently, the copy of `template` included in the `result` FrameSet returned by *Convert* inherits the celestial coordinate system of the matching Frame within `frameset`. If a specific system was specified when `template` was created, then that system would be given priority and be included in the returned `result` FrameSet.

3.4. Regions

Previous sections have described:

1. *Mappings*, which transform numerical axis values.
2. *Frames*, which describe what those numerical axis values represent.
3. *FrameSets*, which associate Mappings with Frames.

However, nothing has been said yet about how the actual axis values themselves are stored. This section address this issue. In most cases, the calling application code will organise and store the axis values itself in some manner, creating appropriate Mappings and Frames to process them. However AST also provides a collection of *Region* classes, each of which encapsulates the axis values at a set of positions, along with a Frame describing the coordinate system to which those axis values refer. As the name implies, each Region subclass describe a specific region within the encapsulated Frame.

The simplest subclass of Region is the *PointList* that represents a simple list of one or more unconnected points within the Frame. In practice, PointLists are rarely used. This is because a typical application is usually interested in positions that it has determined itself in some way, and so can process them directly using Mappings without the need to create an intermediate container object such as a PointList. Applications that for instance displays the world coordinates of the cursor, or the centroid position of an object, will usually obtain the axis values in some "base" system such as graphics screen coordinates or pixel coordinates, and will then transform them directly into the coordinate system of interest using a suitable Mapping that it has obtained previously. There is no need to first wrap the axis values in a PointList, transform the PointList to a new coordinate system, and then extract the coordinates from the transformed PointList. Doing so can potentially introduces large overheads that can increase the overall time taken to do the transformation.

Other subclasses of Region exist to describe circles, ellipses, boxes polygons, *etc*. For instance, a Circle encapsulates two positions within a Frame - one being the centre of the required circle and the other being a point on its perimeter. A Polygon is similar to a PointList in that it encapsulates an arbitrary list of points. But in a Polygon, the points represent the vertices of a polygon with non-zero area, rather than a list of unconnected points with zero area.

The *CmpRegion* class is a compound Region that represents the union or intersection of two other Regions. Another form of compound Region is the Prism class that allows Regions to be extruded into other dimensions[16]. For instance, a 4-dimensional

[15] This may or may not be possible depending on the nature of the two Frames.

[16] Some Region classes, such as Polygon and Ellipse, are intrinsically 2-dimensional, whereas other such as Circle and Box, can have more than 2 dimensions.

Prism can be formed by combining a 2-dimensional Circle and a 2-dimensional Box. A 4-dimensional position is considered to be inside the Prism if the values on axes 1 and 2 are inside the Circle, and the values on axes 3 and 4 are inside the Box.

Like the FrameSet class, the base Region class inherits from both the Frame class and the Mapping class. When used as a Frame, it behaves like the encapsulated Frame. When used as a Mapping, it behaves like a unit transformation for positions that are inside the region, but returns "bad" axis values for positions that are outside the region (see section 3.1.4).

If the properties of the encapsulated Frame are changed so that it describes a different coordinate system, the encapsulated axis values are automatically transformed into the new system by a process similar to that described for FrameSets in section 3.3.2.

Regions can encapsulate a FrameSet in place of a Frame. In this case, the shape and extent of the Region are defined within the base Frame of the FrameSet, and the Region notionally represents a "view" of this shape transformed into the current Frame of the FrameSet. For instance, a "Box" could be created to represent the rectangular extent of an image in pixel coordinates. Such a Box will contain a Frame describing pixel coordinates, the pixel coordinates at the centre of the image, and the extent of each dimension of the image (in pixels). The Frame in this Box could be replaced by a FrameSet in which the base Frame is again pixel coordinates, but the current Frame is (RA,Dec). Internally, the Box still represents a rectangular area in pixel coordinates, but is now viewed externally as the corresponding curvilinear area in the (RA,Dec) Frame. When the Box is used as a Mapping, it will first transform each position from (RA,Dec) into pixel coordinates, and then test these transformed positions to see if they are within the rectangular area of pixel coordinates described by the Box.

4. Serialisation, and FITS-WCS

AST includes a set of *channel* classes, which allow AST objects to be serialised in various ways. The astWrite method of the basic astChannel class converts a in-memory AST object into a set of text strings. By default these text strings are simply written to standard output, but an external "sink" function can be supplied to the astChannel constructor which — if supplied — will be called by the astChannel class to store each text string in some associated external store. The astRead method of the astChannel class does the inverse — it calls a supplied "source" function to read a set of text strings from an external store, and then creates an in-memory AST object from the text strings. A round-trip (astWrite followed by astRead) is lossless. The textual descriptions of AST objects produced by the astChannel class use a bespoke block structured format specific to AST.

Within the Python interface, the *print* function uses an astChannel to produce a listing of the given AST object. As an example, the following Python code:

```
map1 = starlink.Ast.PermMap( [3,1],
                             [2,-1,1], 12.2 )
map2 = starlink.Ast.ZoomMap( 3, 4.0 )
cmpmap = starlink.Ast.CmpMap( map1, map2 )
print( cmpmap )
```

produces the following output from astChannel:

```
Begin CmpMap   # Compound Mapping
   Nin = 2     # Number of input coordinates
   Nout = 3    # Number of output coordinates
IsA Mapping    # Mapping between coordinate systems
   MapA =      # First component Mapping
      Begin PermMap    # Coordinate permutation
         Nin = 2       # Number of input coordinates
         Nout = 3      # Number of output coordinates
      IsA Mapping      # Mapping between coordinate systems
         Out1 = 2      # Output coordinate 1 = input coordinate 2
         Out2 = -1     # Output coordinate 2 = constant no. 1
         Out3 = 1      # Output coordinate 3 = input coordinate 1
         In1 = 3       # Input coordinate 1 = output coordinate 3
         In2 = 1       # Input coordinate 2 = output coordinate 1
         Nconst = 1    # Number of constants
         Con1 = 12.2   # Constant number 1
      End PermMap
   MapB =      # Second component Mapping
      Begin ZoomMap    # Zoom about the origin
         Nin = 3       # Number of input coordinates
      IsA Mapping      # Mapping between coordinate systems
         Zoom = 4      # Zoom factor
      End ZoomMap
End CmpMap
```

Attributes of the astChannel class can be used to enable or disable inclusion of comments, indentation, defaulted values, *etc*.

5. Fields of Application

This section describes some of the ways in which AST can be used.

Whilst the AST library provides many facilities that are useful when describing and using the WCS information attached to a data array, it is not limited to that field. Its generalised design enables it to be used in any situation where relationships between several different coordinate systems need to be managed. Just to emphasise that point, WCS handling is included as the last item in this section.

5.1. Generalised Plotting

Most plotting operations involve the use of several different Cartesian coordinate systems. A typical graphics package may need to deal with:

- A system defined by the graphics hardware (for instance, pixel indices on a computer screen, *etc*).
- A system describing some normalised version of the above system (for instance, a system in which the shortest edge of the screen is normalized to unit length).
- A system describing a restricted window within the total plotting space (for instance, a system in which the shortest edge of the window is normalized to unit length).
- A system describing a restricted window within the total plotting space (for instance, a system in which the shortest edge of the window is normalized to unit length).

- A user-supplied system that results in the graphics window being mapped onto some specified viewport within Cartesian "user" coordinates.

Whilst in principle AST could be used internally within such a graphics package to handle these different coordinate systems, this has not (so far) been done. A more common usage of AST is to allow application software to define extra — potentially non-linear — coordinate systems, and to integrate them with the coordinate systems provided by an existing graphics package. For this purpose AST provides two classes — *Plot*, which provides two-dimensional plotting facilities, and *Plot3D*, which provides three-dimensional plotting facilities.

5.1.1. Use of External Plotting Packages

The plotting classes within AST do not themselves include any facilities for placing "ink onto paper" — an external graphic library must be supplied to draw graphical primitives such as straight lines, markers and character strings. The plotting classes within AST will then use the primitive facilities of this underlying graphics system to draw more complex entities such as annotated coordinate grids, *etc*.

This separation between coordinate handling and drawing enables the sophisticated plotting capabilities of AST to be used within many different systems and languages[17].

The underlying graphics system can be specified in two ways:

1. At build-time. In this method, a module must be supplied (callable from C) that provides implementations of a set of wrapper functions that AST uses to perform primitive drawing operations. Each such wrapper function makes appropriate calls to the underlying graphics system to perform its work. This module is linked into the executable at build-time, in place of the default module provided by AST.
2. At run-time. In this method, application code registers pointers to the graphics wrapper functions with AST, whilst the executable is running. The registered function pointers are used in preference to any functions specified at build-time.

The AST library includes modules that allow the PGPLOT graphics package (Pearson, 1991, ascl:1103.002) to be used for drawing by both the Plot class and the Plot3D class (see the following sections).

5.1.2. Two Dimensional Plotting

A *Plot* is a subclass of FrameSet and can therefore be considered to be a FrameSet 'with some extra facilities". A Plot does not represent the graphical content itself, but is a route through which plotting operations, such as drawing lines and curves, are conveyed on to a plotting surface to appear as visible graphics.

When considered as a FrameSet, the base Frame within a Plot corresponds to the Cartesian coordinate system used to specify positions to the underlying graphics system[18]. The bounds of the plotting area within this coordinate system must be specified when the Plot is created. A typical Plot may for instance have a base Frame that describes millimetres from the bottom left corner of the plotting area.

The current Frame within a Plot corresponds to "user" coordinates — *i.e.*, the coordinate system in which the application code wishes to specify positions. Since the Plot is a form of FrameSet, it will also include the Mappings needed to transform "user" coordinates into the corresponding graphics coordinates. A typical Plot may for instance have a current Frame that describes (RA,Dec) positions on the sky.

When a Plot is created, an existing FrameSet must be supplied that includes the required "user" coordinate system together with some intermediate coordinate system (typically image pixel coordinates). The bounds of a viewport within this intermediate system must also be supplied. The FrameSet that *is* the Plot is then created by adding an extra Frame representing graphics coordinates into the supplied FrameSet. This Frame becomes the base Frame in the Plot, and is connected to the intermediate coordinate Frame (*e.g.*, pixel coordinates) using a Mapping that maps the requested viewport within intermediate coordinates onto the plotting area — either linearly or logarithmically. In effect, the Plot becomes attached to the plotting surface in rather the same way that a basic FrameSet might be attached to (say) an image.

The Plot class has methods that can draw markers, geodesic curves, text strings, *etc*. The application code supplies the positions of these objects within the "user" coordinate system (*i.e.*, the current Frame of the Plot — *e.g.*, (RA,Dec)). The Plot class then uses the Mappings stored within the Plot to transform them into the graphics coordinate system (*i.e.*, the base Frame of the Plot), before invoking the appropriate wrapper functions to instruct the underlying graphics system to draw the required primitives. Whilst this is a fairly simple process for items such as graphical markers that only have one associated coordinate (*i.e.*, the centre of the marker), it is a more complex process for items such as geodesic curves that span a wide range of different coordinates. In this case, the curve is represented by a set of positions spread along the curve in user coordinates. All these positions are transformed into graphics coordinates before being plotted in the form of a "poly-line". The density of points varies along this line, and is chosen to ensure that any discontinuities or highly non-linear sections of the curve are drawn with reasonable accuracy.

The most comprehensive drawing method supplied by the Plot class produces a complete set of annotated axes describing the area of user coordinates visible within the plotting area. Some examples are shown in Fig. 3.

[17] It is known that AST graphics have been used with plotting packages written in Java, Perl, Tk-Tcl and Python, as well as C and Fortran.

[18] This coordinate system must have been defined previously using appropriate calls to the underlying graphics system.

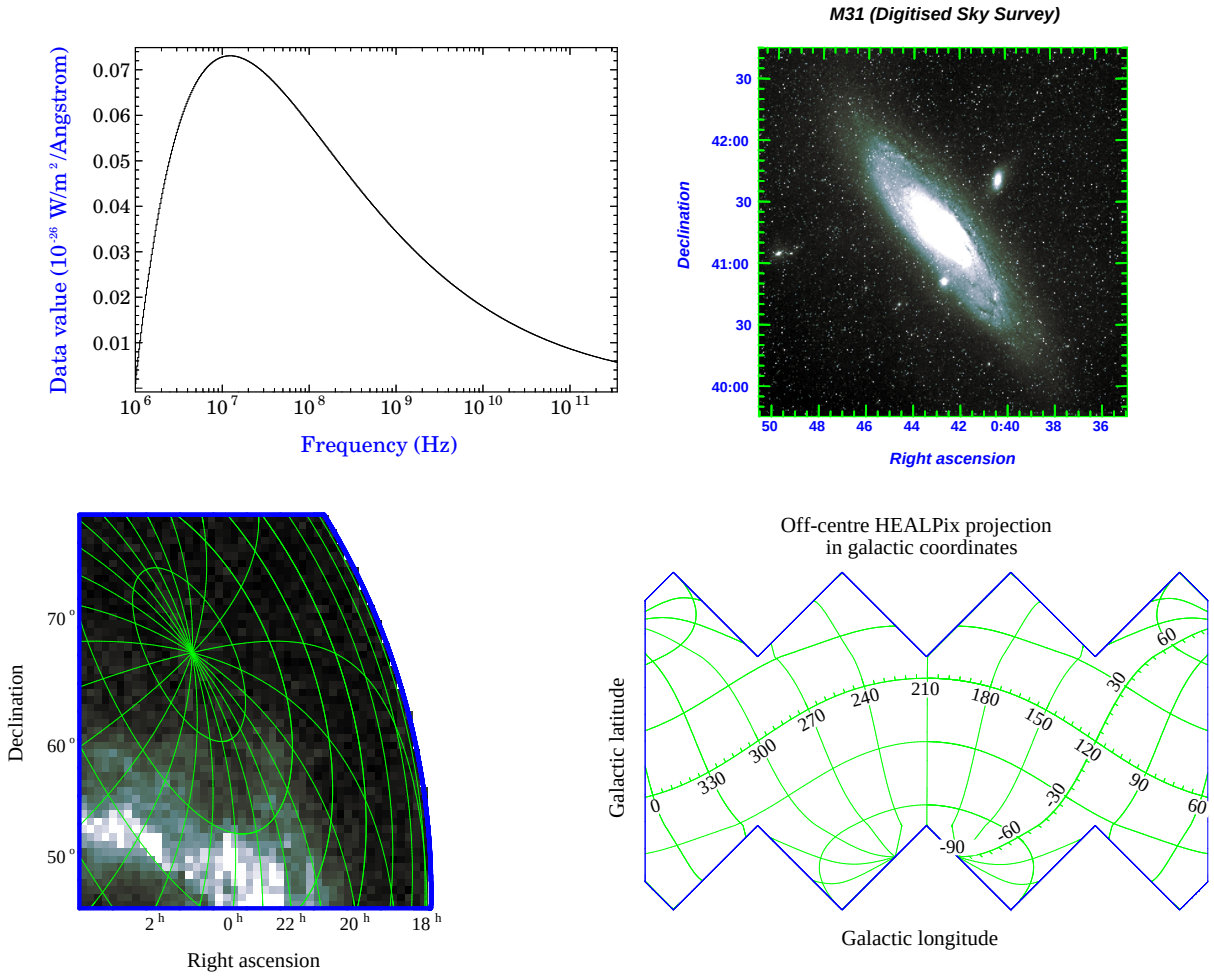

Figure 3: A selection of annotated axes created by the Plot class. Note, Plot only creates the annotated axes — the background images were produced using PGPLOT directly. The PGPLOT library was used as the underlying graphics package for these plots.

5.1.3. Three Dimensional Plotting

The *Plot3D* class extends the Plot class to provide plotting facilities in three dimensions. The basic model is the same as for the Plot class — a Plot3D object is a FrameSet in which the base Frame represents the 3-dimensional Cartesian coordinate system used by the underlying 3-dimensional plotting package, and the current Frame represents 3-dimensional "user" coordinates.

The projection from 3-dimensional graphics coordinates onto a 2-dimensional plotting surface is handled by the underlying graphics package. This may be surprising, given that AST could itself handle this sort of projection. But the decision was taken so that Plot3D graphics wrapper functions could use the full range of features offered by a dedicated 3-dimensional plotting packages, rather than restrict it to the more limited features offered by a typical 2-dimensional package.

Fig. 4 shows an example of the 3-dimensional annotated axes produced by Plot3D. The Plot3D class was originally developed to support 3-D coordinate grid plotting in the GAIA application using VTK (Draper et al., 2008).

5.2. Flux and Data Unit Transformation

In the majority of scientific data sets, each data point has a *position* and a *value*. Whilst the concepts described in this paper may seem more naturally associated with the positional information, they can also be applied to the value (or values) associated with each data point. A position within some N-dimensional space is specified by a set of N axis values. Likewise, a value can also be represented by a set of axis values. In the majority of common cases data values are 1-dimensional - for instance a temperature, or a sky brightness. But there are also some common multi-dimensional cases such as Stokes vector (I, Q, U, V).

The functional division into Mappings, Frames and FrameSets used throughout AST is equally applicable to the problem of describing data *values* as it is to describing data *positions*. To do so requires a set of specialised Frame classes to be created

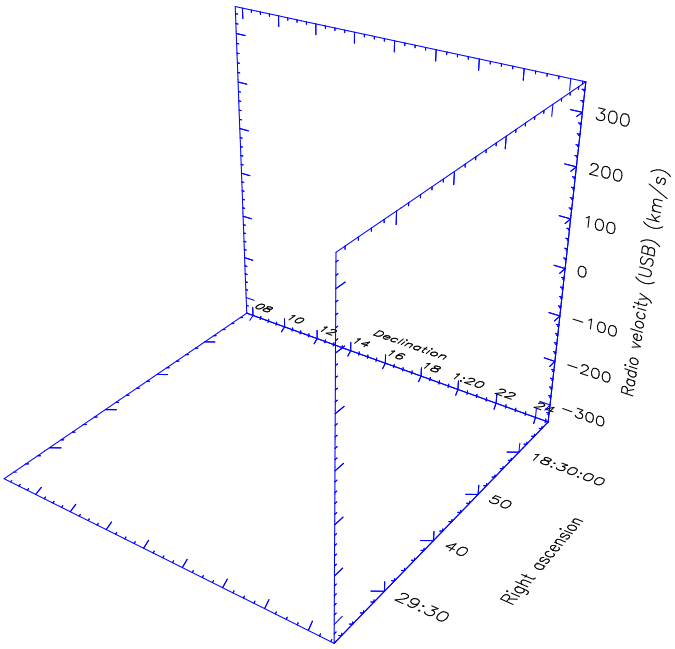

Figure 4: Annotated axes created by the Plot3D class. The 3D interface to the PGPLOT library included with AST was used as the underlying graphics package for these plots.

to describe each set of equivalent data value systems. For instance a hypothetical *TemperatureFrame* could be created that encapsulates the meta-data needed to transform between different temperature scales (Celsius, Fahrenheit, Kelvin, *etc*).

Currently, provision of such specialised Frames within AST is limited, but this may change in future:

FluxFrame : Supports 1-dimensional axes that represent the following astronomical flux systems:

- Flux per unit frequency ($W/m^2/Hz$)
- Flux per unit wavelength ($W/m^2/Angstrom$)
- Surface brightness in frequency units ($W/m^2/Hz/arcmin^2$)
- Surface brightness in wavelength units ($W/m^2/Angstrom/arcmin^2$)

Any dimensionally equivalent units can be used in place of the default units listed above. All flux values are assumed to be measured at the same frequency or wavelength (specified by an attribute of the FluxFrame). Thus this class is more appropriate for use with images than spectra.

Frame : The basic Frame class can be used to represent single-system data value axes (*i.e.* Frames that know only one system for specifying positions within its domain), using any suitable system of units. Such Frames can be used to convert axis values into any dimensionally equivalent set of units. For instance, a basic Frame used to represent stellar mass could be set to use Kg, Solar mass, *etc.* as its units, and will automatically modify the associated Mappings each time the units are changed (assuming the Frame is part of a FrameSet).

5.3. WCS Handling

The most common reason for using AST is to allow the position of one or more data values to be described in a range of possible coordinate systems. For this purpose, a serialised FrameSet is often stored with the data on the assumption that the "natural coordinate system" of the data structure corresponds to a known Frame (usually the base Frame) in the FrameSet. The most common data structure is an N-dimensional regular grid of values, in which case the base Frame in the FrameSet is assumed to describe "pixel coordinates" within the grid (generalising the term "pixel" to data of any dimensionality). Different systems exist for enumerating the pixels within an array - for instance, some start counting at zero and some at one. AST allows multiple pixel coordinate systems to be described and used within a single FrameSet, assuming that the Mappings between such Frames have been set up and incorporated into the FrameSet correctly.

The following sections describe just a few of the more common WCS operations that may be achieved using AST:

5.3.1. Validating WCS

The generalised description of WCS provided by AST makes it more possible to write applications in a domain-agnostic manner. For instance, an application that uses WCS to align two data sets need not know whether the data sets are two-dimensional images of the sky, one dimensional spectra, 3-dimensional time-space cubes, *etc*. However, this will not always be possible, and it will still often be the case that an applications needs to check that the WCS in the supplied data is consistent with the specific requirements of the application. The "astFindFrame" method can often be used for this purpose. The application creates a Frame that acts as a template for the coordinate systems required by the application, and passes this Frame, together with the WCS FrameSet read from the data, to the astFindFrame method, which then searches the FrameSet looking for a Frame that can be matched against the template. If such a Frame is found, information about the specific Frame found is returned, together with a Mapping that connects that Frame to one of the Frames in the WCS FrameSet.

As a trivial example, consider an application that requires 1-dimensional data but does not care what that one dimension represents, it creates a 1-dimensional basic Frame to act as a template, leaving all Frame attributes at their default values in the template (default values act as wild-cards during the searching process). The application then passes this template Frame, together with the WCS FrameSet read from the data file, to astFindFrame. Each Frame in the WCS FrameSet is then compared to the template to see if a match is possible. In this case, a basic Frame with no set attributes will match any 1-dimensional Frame of any form, but will not match Frames with more than one dimension.

A specialised application for processing spectral data cubes may use for its template a 3-dimensional CmpFrame (compound Frame) containing a 2-dimensional SkyFrame (celestial longitude/latitude axes) and 1-dimensional SpecFrame (a spectral axis). The astFindFrame will then only match

3-dimensional Frames with similar properties. Note, if the System attribute is set to specific values in the SkyFrame and/or SpecFrame contained within the template, then the Mapping returned by astFindFrame will include the transformations needed to convert from the system of the matching Frame to that of the template Frame. So for instance, if the template axes are $(RA, Dec, Wavelength)$ and the WCS FrameSet contains a CmpFrame with $(Frequency, GalacticLongitude, GalacticLatitude)$ axes, the Mapping returned by astFindFrame will include an axis permutation, and the transformations needed to convert from frequency to wavelength, and from Galactic coordinates to (RA, Dec).

In other words, astFindFrame provides arbitration between the WCS requirements of the application, and the WCS information that is available in the supplied data set, returning a null result if the requirements of the application are not met by the data set.

5.3.2. *Merging WCS information*

If a data processing operation involves combining two or more data sets in some way, it will usually be necessary to form a connection between the WCS in the two data sets, so that corresponding data values in each data set can be identified[19]. This is accomplished in a completely general way within AST using the astConvert method described in section 3.2.1, which searches two FrameSets looking for a pair of Frames that can be connected together using some known Mapping.

5.3.3. *Modifying WCS Information*

If an application creates an output data array by applying some geometrical transformation to an input data array, it should also store appropriately modified WCS in the output. The concatenation and simplification of Mappings described in section 3.1 makes this easy. The application should create a Mapping that describes the geometric transformation that has been applied to the pixel array, and then "re-map" the pixel Frame within the WCS FrameSet read from the input data set. The resulting modified FrameSet should be stored in the output data set. Re-mapping a Frame within a FrameSet means appending the supplied Mapping to the existing Mapping that connects the Frame with its parent in the FrameSet. The astRemapFrame method is provided to do this. It also simplifies the resulting compound Mapping if possible.

For instance, if an application rotates an input image to create an output image, the application should create a 2-dimensional MatrixMap to describe the rotation[20] and then invoke astRemapFrame, supplying the MatrixMap and the WCS FrameSet from the input image. The modified FrameSet would then be stored in the output image.

[19] Possibly the most common example is co-adding several images into a single mosaic.

[20] A pair of ShiftMaps will also be needed if the rotation is not around the pixel origin.

5.3.4. *Using WCS for Data Resampling and Regridding*

The previous section described how to modify the WCS to take account of a geometric transformation of a data set, assuming a Mapping describing the transformation is available. If such a Mapping is available, it can also be used to perform the actual resampling or regridding of the pixel values themselves. AST provides several methods that will perform such resampling or regridding, using any of a wide range of alternative sampling kernels. Alternatively, an externally defined sampling kernel can be used.

The benefit of using these methods within AST rather than simply transforming every pixel position within the application code, is that the AST methods attempt to speed up the operation by using linear approximations to the supplied transformation if possible. Data sets are constantly increasing in size, and transforming every pixel position within a large data set using a long and complicated transformation can be an expensive operation. The AST methods divide the input data set in half along each axis, and find a linear approximation to the Mapping over each resulting quadrant. If one of these approximations meets a user-specified accuracy, the approximation is used to transform all pixel positions within its quadrant. Any quadrants that do not meet the accuracy requirements are subdivided again and new approximations are found for each of the new sub-quadrants. This process repeats recursively until the sub-quadrants become so small that there is likely to be little time-saving in sub-dividing them any further. At this point the full transformation is used on all pixels within any such sub-quadrants.

6. Things we Would do Differently Now

1. Be clearer about the distinction between a coordinate system and a domain. Coordinate systems are mathematical abstractions of various types (Cartesian, polar, *etc*). You use a coordinate system to describe a position within a physical space (domain). In this sense a domain can encapsulate several alternative coordinate systems, any of which can be used to describe positions in the domain.
2. Have a specific sub-class of Frame to describe a pixel array (a PixelFrame), allowing System=GRID or System=PIXEL.
3. Use degrees instead of radians for sky axes.
4. The restriction that Mappings can only transform doubles may be a problem for time axes (but this is an issue with the implementation rather than the model).
5. Make it more modular. Some sort of facility for optional extensions or plug-ins, so that the thing does not become such a huge monolithic lump.

7. The AST library

The AST library has been developed over a number of years (Berry and Draper, 2010; Berry, 2001, 2004, 2008; Berry and Jenness, 2012; Warren-Smith and Berry, 1998, 2000) and is written in C with no dependencies. It includes code from

WCSLIB (Calabretta, 2006, ascl:1108.003), PAL (Jenness and Berry, 2013) and SOFA (Hohenkerk, 2011, ascl:1403.026) but does not depend on those libraries. There are language bindings for Fortran, Java, Perl and Python.

8. Acknowledgements

This work was funded by the Council for the Central Laboratory of the Research Councils and subsequently by the Science and Technology Facilities Council. The AST library and the related Starlink software are currently maintained by the East Asian Observatory, Hawaii. The source code for the AST library is open-source using the Lesser Gnu Public License and is available online[21].

Appendix A. Methods and Properties

This appendix includes brief descriptions of the more significant methods and properties of the three main classes used by AST - Mapping, Frame and FrameSet.

Appendix A.1. The Mapping Class

The most important public methods of the base Mapping class are:

astInvert: Invert a Mapping (i.e., swap the input and output spaces).

astLinearApprox: Create a new Mapping which is a linear approximation to the supplied Mapping.

astMapSplit: Create (if possible) a new Mapping that represents a subspace within the output space of the supplied Mapping. For instance, given a Mapping from 3-dimensional pixel coordinates to (RA,Dec,wavelength), create a Mapping between 2-dimensional pixel coordinates and (RA,Dec). This is only possible if the requested subspace is independent of the other output axes.

astRate: Calculate the rate of change of a specified axis within the output space, with respect to a specified axis within the input space, at a given position within the input space.

astRebin: : Re-bin the pixel values within an input data array to create an output data array, using the forward transformation of the supplied Mapping to define the position within the output array at which each input pixel value is placed.

astResample: : Re-sample the pixel values within an input data array to create an output data array, using the inverse transformation of the supplied Mapping to define the position within the input array from which each output pixel value is copied. A range of different interpolation schemes are provided.

astSimplify: Simplify a compound Mapping (see section 3.1.3).

astTran: Transform coordinates using the supplied Mapping.

The most important public properties of the base Mapping class are:

IsLinear: A boolean flag indicating if the Mapping is linear.

Nin: The number of axes within the input space of the Mapping.

Nout: The number of axes within the output space of the Mapping.

TranForward: A boolean flag indicating if the Mapping has a defined forward transformation.

TranInverse: A boolean flag indicating if the Mapping has a defined inverse transformation.

[21] https://github.com/Starlink/starlink

References

Astropy Collaboration, 2013. Astropy: A community Python package for astronomy. Astron Astrophys 558, A33. doi:10.1051/0004-6361/201322068, arXiv:1307.6212.

Berry, D., Draper, P., 2010. Using the AST Library to Create and Use STC-S Region Descriptions, in: Mizumoto, Y., Morita, K.I., Ohishi, M. (Eds.), Astronomical Data Analysis Software and Systems XIX, volume 434 of *ASP Conf. Ser.*. p. 213.

Berry, D.S., 2001. Providing Improved WCS Facilities Through the Starlink AST and NDF Libraries, in: Harnden, Jr., F.R., Primini, F.A., Payne, H.E. (Eds.), Astronomical Data Analysis Software and Systems X, volume 238 of *ASP Conf. Ser.*. p. 129.

Berry, D.S., 2004. Developments in the Starlink AST Library - an Intelligent WCS Management System, in: Ochsenbein, F., Allen, M.G., Egret, D. (Eds.), Astronomical Data Analysis Software and Systems (ADASS) XIII, volume 314 of *ASP Conf. Ser.*. p. 412.

Berry, D.S., 2008. Developments in the AST Library, in: Argyle, R.W., Bunclark, P.S., Lewis, J.R. (Eds.), Astronomical Data Analysis Software and Systems XVII, volume 394 of *ASP Conf. Ser.*. p. 635.

Berry, D.S., Jenness, T., 2012. New Features in AST: A WCS Management and Manipulation Library, in: Ballester, P., Egret, D., Lorente, N.P.F. (Eds.), Astronomical Data Analysis Software and Systems XXI, volume 461 of *ASP Conf. Ser.*. p. 825.

Calabretta, M.R., 2006. Implementation of -TAB coordinates in WCSLIB, in: Gabriel, C., Arviset, C., Ponz, D., Enrique, S. (Eds.), Astronomical Data Analysis Software and Systems XV, volume 351 of *ASP Conf. Ser.*. p. 591.

Currie, M.J., Berry, D.S., Jenness, T., Gibb, A.G., Bell, G.S., Draper, P.W., 2014. Starlink Software in 2013, in: Manset, N., Forshay, P. (Eds.), Astronomical Data Analysis Software and Systems XXIII, volume 485 of *ASP Conf. Ser.*. p. 391.

Draper, P., 1998. New Features in GAIA. Starlink Bulletin 20, 7.

Draper, P.W., Berry, D.S., Jenness, T., Economou, F., Currie, M.J., 2008. GAIA-3D: Volume Visualization of Data-Cubes, in: Argyle, R.W., Bunclark, P.S., Lewis, J.R. (Eds.), Astronomical Data Analysis Software and Systems XVII, volume 394 of *ASP Conf. Ser.*. p. 339.

Eaton, N., McIlwrath, B., 2013. AGI — Applications Graphics Interface Library. Starlink User Note 48, Starlink Project.

Evans, I. (Ed.), 2013. Twenty Years of ADASS. volume MP-006 of *ASP Monograph*. Astronomical Society of the Pacific. ISBN: 978-1-58381-822-0.

Greisen, E.W., Calabretta, M., 1995. Representations of Celestial Coordinates in FITS, in: Shaw, R.A., Payne, H.E., Hayes, J.J.E. (Eds.), Astronomical Data Analysis Software and Systems IV, volume 77 of *ASP Conf. Ser.*. p. 233.

Hohenkerk, C., 2011. Standards of Fundamental Astronomy. Scholarpedia 6, 11404. doi:10.4249/scholarpedia.11404.

Holub, A.I., 1992. Programming with Objects in C and C++. McGraw-Hill. ISBN 0-07-029662-6.

Jenness, T., 2015. Reimplementing the Hierarchical Data System using HDF5. Astron. Comp. in press. doi:10.1016/j.ascom.2015.02.003, arXiv:1502.04029.

Jenness, T., et al., 2015. Learning from 25 years of the extensible *N*-Dimensional Data Format. Astron. Comp. in press. doi:10.1016/j.ascom.2014.11.001, arXiv:1410.7513.

Jenness, T., Berry, D.S., 2013. PAL: A Positional Astronomy Library, in: Friedel, D.N. (Ed.), Astronomical Data Analysis Software and Systems XXII, volume 475 of *ASP Conf. Ser.*. p. 307.

Joye, W.A., Mandel, E., 2003. New Features of SAOImage DS9, in: Payne, H.E., Jedrzejewski, R.I., Hook, R.N. (Eds.), Astronomical Data Analysis Software and Systems XII, volume 295 of *ASP Conf. Ser.*. p. 489.

Lawden, M., 1989. Software and documentation news. Starlink Bulletin 4, 7.

Lawden, M., 1998. New Products. Starlink Bulletin 20, 6.

Lupton, W.F., 1989. HDS — Hierarchical Data System. Starlink System Note 27, Starlink Project.

Mink, J., Mann, R.G., Hanisch, R., Rots, A., Seaman, R., Jenness, T., Thomas, B., 2015. The Past, Present and Future of Astronomical Data Formats, in: Taylor, A.R., Stil, J.M. (Eds.), Astronomical Data Analysis Software and Systems XXIV, volume in press of *ASP Conf. Ser.*. arXiv:1411.0996.

Pearson, T.J., 1991. Caltech VLBI Analysis Programs, California Institute of Technology. Bulletin of the American Astronomical Society 23, 991–992.

Thomas, B., et al., 2015. Learning from FITS: Limitations in use in modern astronomical research. Astron. Comp. in press. doi:10.1016/j.ascom.2015.01.009, arXiv:1502.00996.

Wallace, P.T., 1994. The SLALIB Library, in: Crabtree, D.R., Hanisch, R.J., Barnes, J. (Eds.), Astronomical Data Analysis Software and Systems III, volume 61 of *ASP Conf. Ser.*. p. 481.

Warren-Smith, R.F., 1989. TRANSFORM —Coordinate Transformation Facility. Starlink User Note 61, Starlink Project.

Warren-Smith, R.F., Berry, D.S., 1998. World Coordinate Systems as Objects, in: Albrecht, R., Hook, R.N., Bushouse, H.A. (Eds.), Astronomical Data Analysis Software and Systems VII, volume 145 of *ASP Conf. Ser.*. p. 41.

Warren-Smith, R.F., Berry, D.S., 2000. Recent Developments to the AST Astrometry Library, in: Manset, N., Veillet, C., Crabtree, D. (Eds.), Astronomical Data Analysis Software and Systems IX, volume 216 of *ASP Conf. Ser.*. p. 506.

Warren-Smith, R.F., Berry, D.S., 2013. AST – A Library for Handling World Coordinate Systems in Astronomy. Starlink User Note 211, Starlink Project, STFC.

Warren-Smith, R.F., Lawden, M.D., McIlwrath, B.K., Jenness, T., Draper, P.W., 2008. HDS — Hierarchical Data System. Starlink User Note 92, Starlink Project.

Wells, D.C., Greisen, E.W., Harten, R.H., 1981. FITS: a flexible image transport system. Astron Astrophys Supp 44, 363–370.