ASDF Standard

Release 0.1.0dev

Michael Droettboom, Erik Bray, et al
Space Telescope Science Institute

April 06, 2015

This document describes the Advanced Scientific Data Format (ASDF), pronounced *AZ*-diff.

This document is a work in progress and does not represent a released version of the ASDF standard.

INTRODUCTION

The Flexible Image Transport System (FITS) has been the de facto standard for storing and exchanging astronomical data for decades, but it is beginning to show its age. Developed in the late 1970s, the FITS authors made a number of implementation choices that, while common at the time, are now seen to limit its utility for the needs of modern science. As astronomy moves into a more varied set of data product types (data models) with richer and more complex metadata, FITS is being pushed to its breaking point. The issues with FITS are outlined in great detail in *[Thomas2015]* (page 55).

Newer formats, such as VOTable (http://www.ivoa.net/documents/VOTable/) have partially addressed the problem of richer, more structured metadata, by using tree structures rather than flat key/value pairs. However, those text-based formats are unsuitable for storing large amounts of binary data. On the other end of the spectrum, formats such as HDF5 (http://www.hdfgroup.org/HDF5/) and BLZ (http://blaze.pydata.org/docs/) address problems with large data sets and distributed computing, but don't really address the metadata needs of an interchange format. ASDF aims to exist in the same middle ground that made FITS so successful, by being a hybrid text and binary format: containing human editable metadata for interchange, and raw binary data that is fast to load and use. Unlike FITS, the metadata is highly structured and is designed up-front for extensibility.

ASDF has the following explicit goals:

- It has a hierarchical metadata structure, made up of basic dynamic data types such as strings, numbers, lists and mappings.

- It has human-readable metadata that can be edited directly in place in the file.

- The structure of the data can be automatically validated using schema.

- It's designed for extensibility: new conventions may be used without breaking backward compatibility with tools that do not understand those conventions. Versioning systems are used to prevent conflicting with alternative conventions.

- The binary array data (when compression is not used) is a raw memory dump, and techniques such as memory mapping can be used to efficiently access it.

- It is possible to read and write the file in as a stream, without requiring random access.

- It's built on top of industry standards, such as YAML (http://www.yaml.org) and JSON Schema (http://www.json-schema.org) to take advantage of a larger community working on the core problems of data representation. This also makes it easier to support ASDF in new programming languages and environments by building on top of existing libraries.

- Since every ASDF file has the version of the specification to which it is written, it will be possible, through careful planning, to evolve the ASDF format over time, allowing for files that use new features while retaining backward compatibility with older tools.

ASDF is primarily intended as an interchange format for delivering products from instruments to scientists or between scientists. While it is reasonably efficient to work with and transfer, it may not be optimal for direct use on large data sets in distributed and high performance computing environments. That is explicitly not a goal of the ASDF standard, as those requirements can sometimes be at odds with the needs of an interchange format.

ASDF still has a place in those environments as a delivery mechanism, even if it ultimately is not the actual format on which the computing is performed.

1.1 Implementations

The ASDF standard is being developed concurrently with a reference implementation written in Python (http://github.com/spacetelescope/pyasdf).

1.2 Incorporated standards

The ASDF format is built on top of a number of existing standards:

- YAML 1.1 (http://yaml.org/spec/1.1/)
- JSON Schema Draft 4:
    - Core (http://tools.ietf.org/html/draft-zyp-json-schema-04)
    - Validation (http://tools.ietf.org/html/draft-fge-json-schema-validation-00)
    - Hyper-Schema (http://tools.ietf.org/html/draft-luff-json-hyper-schema-00)
- JSON Pointer (http://tools.ietf.org/html/rfc6901)
- VOUnits (Units in the VO) (http://www.ivoa.net/documents/VOUnits/index.html)
- Zlib Deflate compression (http://www.zlib.net/feldspar.html)

LOW-LEVEL FILE LAYOUT

The overall structure of a file is as follows (in order):

- *Header* (page 5)
- *Tree* (page 6), optional
- Zero or more *Blocks* (page 6)

ASDF is a hybrid text and binary format. The header and tree are text, (specifically, in UTF-8), while the blocks are raw binary.

The low-level file layout is designed in such a way that the tree section can be edited by hand, possibly changing its size, without requiring changes in other parts of the file. The same is not true for resizing a block, which has an explicit size stored in the file (except for, optionally, the last block).

Note also that, by design, an ASDF file containing no binary blocks is also a completely standard and valid YAML file.

Additionally, the spec allows for extra unallocated space after the tree and between blocks. This allows libraries to more easily update the files in place, since it allows expansion of certain areas without rewriting of the entire file.

2.1 Header

All ASDF files must start with a short one-line header. For example:

```
#ASDF 0.1.0
```

It is made up of the following parts (described in EBNF form):

```
asdf_token = "#ASDF"
major      = integer
minor      = integer
micro      = integer
header     = asdf_token " " major "." minor "." micro ["\r"] "\n"
```

- asdf_token: The constant string #ASDF. This can be used to quickly identify the file as an ASDF file by reading the first 5 bytes. It begins with a # so it will be treated as a YAML comment such that the *Header* (page 5) and the *Tree* (page 6) together form a valid YAML file.
- major: The major version.
- minor: The minor version.
- micro: The bugfix release version.

2.2 Tree

The tree stores structured information using YAML Ain't Markup Language (YAML™) 1.1 (http://yaml.org/spec/1.1/) syntax. While it is the main part of most ASDF files, it is entirely optional, and a ASDF file may skip it completely. This is useful for creating files in *Exploded form* (page 8). Interpreting the contents of this section is described in greater detail in *The tree in-depth* (page 9). This section only deals with the serialized representation of the tree, not its logical contents.

The tree is always encoded in UTF-8, without an explicit byteorder marker (BOM). Newlines in the tree may be either DOS ("\r\n") or UNIX ("\n") format.

In ASDF 0.1.0dev, the tree must be encoded in YAML version 1.1 (http://yaml.org/spec/1.1/). At the time of this writing, the latest version of the YAML specification is 1.2, however most YAML parsers only support YAML 1.1, and the benefits of YAML 1.2 are minor. Therefore, for maximum portability, ASDF requires that the YAML is encoded in YAML 1.1. To declare that YAML 1.1 is being used, the tree must begin with the following line:

```
%YAML 1.1
```

The tree must contain exactly one YAML document, starting with `---` (YAML document start marker) and ending with `...` (YAML document end marker), each on their own line. Between these two markers is the YAML content. For example:

```
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/0.1.0/
--- !core/asdf
data: !core/ndarray
  source: 0
  datatype: float64
  shape: [1024, 1024]
...
```

The size of the tree is not explicitly specified in the file, so that it can easily be edited by hand. Therefore, ASDF parsers must search for the end of the tree by looking for the end-of-document marker (`...`) on its own line. For example, the following regular expression may be used to find the end of the tree:

```
\r?\n...\r?\n
```

Though not required, the tree should be followed by some unused space to allow for the tree to be updated and increased in size without performing an insertion operation in the file. It also may be desirable to align the start of the first block to a filesystem block boundary. This empty space may be filled with any content (as long as it doesn't contain the `block_magic_token` described in *Blocks* (page 6)). It is recommended that the content is made up of space characters (`0x20`) so it appears as empty space when viewing the file.

2.3 Blocks

Following the tree and some empty space, or immediately following the header, there are zero or more binary blocks.

Blocks represent a contiguous chunk of binary data and nothing more. Information about how to interpret the block, such as the data type or array shape, is stored entirely in `ndarray` structures in the tree, as described in *ndarray* (page 14). This allows for a very flexible type system on top of a very simple approach to memory management within the file. It also allows for new extensions to ASDF that might interpret the raw binary data in ways that are yet to be defined.

There may be an arbitrary amount of unused space between the end of the tree and the first block. To find the beginning of the first block, ASDF parsers should search from the end of the tree for the first occurrence of the

`block_magic_token`. If the file contains no tree, the first block must begin immediately after the header with no padding.

2.3.1 Block header

Each block begins with the following header:

- `block_magic_token` (4 bytes): Indicates the start of the block. This allows the file to contain some unused space in which to grow the tree, and to perform consistency checks when jumping from one block to the next. It is made up of the following 4 8-bit characters:

  - in hexadecimal: d3, 42, 4c, 4b

  - in ascii: "\323BLK"

- `header_size` (16-bit unsigned integer, big-endian): Indicates the size of the remainder of the header (not including the length of the `header_size` entry itself or the `block_magic_token`), in bytes. It is stored explicitly in the header itself so that the header may be enlarged in a future version of the ASDF standard while retaining backward compatibility. Importantly, ASDF parsers should not assume a fixed size of the header, but should obey the `header_size` defined in the file. In ASDF version 0.1, this should be at least 48, but may be larger, for example to align the beginning of the block content with a file system block boundary.

- `flags` (32-bit unsigned integer, big-endian): A bit field containing flags (described below).

- `compression` (4-byte byte string): The name of the compression algorithm, if any. Should be \0\0\0\0 to indicate no compression. See *Compression* (page 7) for valid values.

- `allocated_size` (64-bit unsigned integer, big-endian): The amount of space allocated for the block (not including the header), in bytes.

- `used_size` (64-bit unsigned integer, big-endian): The amount of used space for the block on disk (not including the header), in bytes.

- `data_size` (64-bit unsigned integer, big-endian): The size of the block when decoded, in bytes. If `compression` is all zeros (indicating no compression), it **must** be equal to `used_size`. If compression is being used, this is the size of the decoded block data.

- `checksum` (16-byte string): An optional MD5 checksum of the used data in the block. The special value of all zeros indicates that no checksum verification should be performed.

2.3.2 Flags

The following bit flags are understood in the `flags` field:

- `STREAMED` (0x1): When set, the block is in streaming mode, and it extends to the end of the file. When set, the `allocated_size`, `used_size` and `data_size` fields are ignored. By necessity, any block with the `STREAMED` bit set must be the last block in the file.

2.3.3 Compression

Currently, two block compression types are supported:

- `zlib`: The zlib lossless compression algorithm. It is widely used, patent-unencumbered, and has an implementation released under a permissive license in zlib (http://www.zlib.net/).

- `bzp2`: The bzip2 lossless compression algorithm. It is widely used, assumed to be patent-unencumbered, and has an implementation released under a permissive license in the bzip2 library (http://www.bzip.org/).

2.3.4 Block content

Immediately following the block header, there are exactly `used_space` bytes of meaningful data, followed by `allocated_space - used_space` bytes of unused data. The exact content of the unused data is not enforced. The ability to have gaps of unused space allows an ASDF writer to reduce the number of disk operations when updating the file.

2.4 Exploded form

Exploded form expands a self-contained ASDF file into multiple files:

- An ASDF file containing only the header and tree, which by design is also a valid YAML file.

- *n* ASDF files, each containing a single block.

Exploded form is useful in the following scenarios:

- Not all text editors may handle the hybrid text and binary nature of the ASDF file, and therefore either can't open an ASDF file or would break an ASDF file upon saving. In this scenario, a user may explode the ASDF file, edit the YAML portion as a pure YAML file, and implode the parts back together.

- Over a network protocol, such as HTTP, a client may only need to access some of the blocks. While reading a subset of the file can be done using HTTP Range headers, it still requires one (small) request per block to "jump" through the file to determine the start location of each block. This can become time-consuming over a high-latency network if there are many blocks. Exploded form allows each block to be requested directly by a specific URI.

- An ASDF writer may stream a table to disk, when the size of the table is not known at the outset. Using exploded form simplifies this, since a standalone file containing a single table can be iteratively appended to without worrying about any blocks that may follow it.

Exploded form describes a convention for storing ASDF file content in multiple files, but it does not require any additions to the file format itself. There is nothing indicating that an ASDF file is in exploded form, other than the fact that some or all of its blocks come from external files. The exact way in which a file is exploded is up to the library and tools implementing the standard. In the simplest scenario, to explode a file, each *ndarray source property* (page 17) in the tree is converted from a local block reference into a relative URI.

THE TREE IN-DEPTH

The ASDF tree, being encoded in YAML, is built out of the basic structures common to most dynamic languages: mappings (dictionaries), sequences (lists), and scalars (strings, integers, floating-point numbers, booleans, etc.). All of this comes "for free" by using YAML (http://yaml.org/spec/1.1/).

Since these core data structures on their own are so flexible, the ASDF standard includes a number of schema that define the structure of higher-level content. For instance, there is a schema that defines how *n-dimensional array data* (page 14) should be described. These schema are written in a language called *YAML Schema* (page 45) which is just a thin extension of JSON Schema, Draft 4 (http://json-schema.org/latest/json-schema-validation.html). (Such extensions are allowed and even encouraged by the JSON Schema standard, which defines the $schema attribute as a place to specify which extension is being used.) *ASDF schema definitions* (page 13), provides a reference to all of these schema in detail. *Extending ASDF* (page 45) describes how to use YAML schema to define new schema.

3.1 Tags

YAML includes the ability to assign *Tags* (page 9) (or types) to any object in the tree. This is an important feature that sets it apart from other data representation languages, such as JSON. ASDF defines a number of custom tags, each of which has a corresponding schema. For example the tag of the root element of the tree must always be `tag:stsci.edu:asdf/0.1.0/core/asdf`, which corresponds to the *asdf schema* (page 13) –in other words, the top level schema for ASDF trees. A validating ASDF reader would encounter the tag when reading in the file, load the corresponding schema, and validate the content against it. An ASDF library may also use this information to convert to a native data type that presents a more convenient interface to the user than the structure of basic types stored in the YAML content.

For example:

```
%YAML 1.1
--- !<tag:stsci.edu:asdf/0.1.0/core/asdf>
data: !<tag:stsci.edu:asdf/0.1.0/core/ndarray>
  source: 0
  datatype: float64
  shape: [1024, 1024]
  byteorder: little
...
```

All tags defined in the ASDF standard itself begin with the prefix `tag:stsci.edu:asdf/0.1.0/`. This can be broken down as:

- `tag:` The standard prefix used for all YAML tags.

- `stsci.edu` The owner of the tag.

- `asdf` The name of the standard.

- `0.1.0` The version of the standard.

Following that is the "module" containing the schema (see *ASDF schema definitions* (page 13) for a list of the available modules). Lastly is the tag name itself, for example, `asdf` or `ndarray`. Since it is cumbersome to type out these long prefixes for every tag, it is recommended that ASDF files declare a prefix at the top of the YAML file and use it throughout. (Most standard YAML writing libraries have facilities to do this automatically.) For example, the following example is equivalent to the above example, but is more user-friendly. The %TAG declaration declares that the exclamation point (!) will be replaced with the prefix `tag:stsci.edu:asdf/0.1.0/`:

```
%YAML 1.1
%TAG ! tag:stsci.edu:asdf/0.1.0/
--- !core/asdf
data: !core/ndarray
  source: 0
  datatype: float64
  shape: [1024, 1024]
  byteorder: little
```

An ASDF parser may use the tag to look up the corresponding schema in the ASDF standard and validate the element. The schema definitions ship as part of the ASDF standard.

An ASDF parser may also use the tag information to convert the element to a native data type. For example, in Python, an ASDF parser may convert a *ndarray* (page 14) tag to a Numpy (http://www.numpy.org) array instance, providing a convenient and familiar interface to the user to access *n*-dimensional data.

The ASDF standard does not require parser implementations to validate or perform native type conversion, however. A parser may simply leave the tree represented in the low-level basic data structures. When writing an ASDF file, however, the elements in the tree must be appropriately tagged for other tools to make use of them.

ASDF parsers must not fail when encountering an unknown tag, but must simply retain the low-level data structure and the presence of the tag. This is important, as end users will likely want to store their own custom tags in ASDF files alongside the tags defined in the ASDF standard itself, and the file must still be readable by ASDF parsers that do not understand those tags.

3.2 References

It is possible to directly reference other items within the same tree or within the tree of another ASDF file. This functionality is based on two IETF standards: JSON Pointer (IETF RFC 6901) (http://tools.ietf.org/html/rfc6901) and JSON Reference (Draft 3) (http://tools.ietf.org/html/draft-pbryan-zyp-json-ref-03).

A reference is represented as a mapping (dictionary) with a single key/value pair. The key is always the special keyword $ref and the value is a URI. The URI may contain a fragment (the part following the # character) in JSON Pointer syntax that references a specific element within the external file. This is a /-delimited path where each element is a mapping key or an array index. If no fragment is present, the reference refers to the top of the tree.

Note: JSON Pointer is a very simple convention. The only wrinkle is that because the characters '~' (0x7E) and '/' (0x2F) have special meanings, '~' needs to be encoded as '~0' and '/' needs to be encoded as '~1' when these characters appear in a reference token.

When these references are resolved, this mapping should be treated as having the same logical content as the target of the URI, though the exact details of how this is performed is dependent on the implementation, i.e., a library may copy the target data into the source tree, or it may insert a proxy object that is lazily loaded at a later time.

For example, suppose we had a given ASDF file containing some shared reference data, available on a public webserver at the URI `http://www.nowhere.com/reference.asdf`:

```
wavelengths:
  - !core/ndarray
    source: 0
    shape: [256, 256]
    datatype: float
    byteorder: little
```

Another file may reference this data directly:

```
reference_data:
  $ref: "http://www.nowhere.com/reference.asdf#wavelengths/0"
```

It is also possible to use references within the same file:

```
data: !core/ndarray
  source: 0
  shape: [256, 256]
  datatype: float
  byteorder: little
  mask:
    $ref: "#/my_mask"

my_mask: !core/ndarray
  source: 0
  shape: [256, 256]
  datatype: uint8
  byteorder: little
```

Reference resolution should be performed *after* the entire tree is read, therefore forward references within the same file are explicitly allowed.

Note: The YAML 1.1 standard itself also provides a method for internal references called "anchors" and "aliases". It does not, however, support external references. While ASDF does not explicitly disallow YAML anchors and aliases, since it explicitly supports all of YAML 1.1, their use is discouraged in favor of the more flexible JSON Pointer/JSON Reference standard described above.

3.3 Comments

It is quite common in FITS files to see comments that describe the purpose of the key/value pair. For example:

```
DATE    = '2015-02-12T23:08:51.191614' / Date this file was created (UTC)
TACID   = 'NOAO    '                   / Time granting institution
```

Bringing this convention over to ASDF, one could imagine:

```
# Date this file was created (UTC)
creation_date: !time/utc
  2015-02-12T23:08:51.191614
# Time granting institution
time_granting_institution: NOAO
```

It should be obvious from the examples that these kinds of comments, describing the global meaning of a key, are much less necessary in ASDF. Since ASDF is not limited to 8-character keywords, the keywords themselves can be much more descriptive. But more importantly, the schema for a given key/value pair describes its purpose in

---

detail. (It would be quite straightforward to build a tool that, given an entry in a YAML tree, looks up the schema's description associated with that entry.) Therefore, the use of comments to describe the global meaning of a value are strongly discouraged.

However, there still may be cases where a comment may be desired in ASDF, such as when a particular value is unusual or unexpected. The YAML standard includes a convention for comments, providing a handy way to include annotations in the ASDF file:

```
# We set this to filter B here, even though C is the more obvious
# choice, because B is handled with more accuracy by our software.
filter:
  type: B
```

Unfortunately, most YAML parsers will simply throw these comments out and do not provide any mechanism to retain them, so reading in an ASDF file, making some changes, and writing it out will remove all comments. Even if the YAML parser could be improved or extended to retain comments, the YAML standard does not define which values the comments are associated with. In the above example, it is only by standard reading conventions that we assume the comment is associated with the content following it. If we were to move the content, where should the comment go?

To provide a mechanism to add user comments without swimming upstream against the YAML standard, we recommend a convention for associating comments with objects (mappings) by using the reserved key name //. In this case, the above example would be rewritten as:

```
filter:
  //: |
    We set this to filter B here, even though C was used, because B
    is handled with more accuracy by our software.
  type: B
```

ASDF parsers must not interpret or react programmatically to these comment values: they are for human reference only. No schema may use // as a meaningful key.

ASDF SCHEMA DEFINITIONS

This reference section describes the schema files for the built-in tags in ASDF.

ASDF schemas are arranged into "modules". All ASDF implementations must support the "core" module, but the other modules are optional.

4.1 Core

The core module contains schema that must be implemented by every asdf library.

4.1.1 asdf: Top-level schema for every ASDF file.

Type: object.

Top-level schema for every ASDF file.

This schema contains the top-level attributes for every ASDF file.

Properties:

data

Type: ndarray (page 14).

The data array corresponds to the main science data array in the file. Oftentimes, the data model will be much more complex than a single array, but this array will be used by applications that just want to convert to a display an image or preview of the file. It is recommended, but not required, that it is a 2-dimensional image array.

fits

Type: fits (page 21).

A way to specify exactly how this ASDF file should be converted to FITS.

wcs

Type: wcs (page 42).

The location of the main WCS for the main data.

4.1.2 complex: Complex number value.

Type: string (regex ([-+]?[0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?)?([-+][0-9]*\.?[0-9]+([eE][-+]?[0-9]+)?[JjIi])?).

Complex number value.

Represents a complex number matching the following EBNF grammar

```
plus-or-minus = "+" | "-"
suffix        = "J" | "j" | "I" | "i"
complex       = [ieee754] [plus-or-minus ieee754 suffix]
```

Where `ieee754` is a floating point number in IEEE 754 decimal format.

Examples:

1 real, -1 imaginary:

```
!core/complex 1-1j
```

0 real, 1 imaginary:

```
!core/complex 1J
```

-1 real, 0 imaginary:

```
!core/complex -1
```

4.1.3 ndarray: An *n*-dimensional array or table.

Type: *definitions/inline-data* (page 16) or object.

An *n*-dimensional array or table.

There are two ways to store the data in an ndarray.

- Inline in the tree: This is recommended only for small arrays. In this case, the entire `ndarray` tag may be a nested list, in which case the type of the array is inferred from the content. (See the rules for type inference in the `inline-data` definition below.) The inline data may also be given in the `data` property, in which case it is possible to explicitly specify the `datatype` and other properties.

- External to the tree: The data comes from a *block* (page 6) within the same ASDF file or an external ASDF file referenced by a URI.

Definitions:

scalar-datatype

Type: string from ["int8", "uint8", "int16", "uint16", "int32", "uint32", "int64", "uint64", "float32", "float64", "complex64", "complex128", "bool8"] or array.

Describes the type of a single element.

There is a set of numeric types, each with a single identifier:

- `int8, int16, int32, int64`: Signed integer types, with the given bit size.

- `uint8, uint16, uint32, uint64`: Unsigned integer types, with the given bit size.

- `float32`: Single-precision floating-point type or "binary32", as defined in IEEE 754.

- `float64`: Double-precision floating-point type or "binary64", as defined in IEEE 754.

- complex64: Complex number where the real and imaginary parts are each single-precision floating-point ("binary32") numbers, as defined in IEEE 754.

- complex128: Complex number where the real and imaginary parts are each double-precision floating-point ("binary64") numbers, as defined in IEEE 754.

There are two distinct fixed-length string types, which must be indicated with a 2-element array where the first element is an identifier for the string type, and the second is a length:

- ascii: A string containing ASCII text (all codepoints < 128), where each character is 1 byte.

- ucs4: A string containing unicode text in the UCS-4 encoding, where each character is always 4 bytes long. Here the number of bytes used is 4 times the given length.

Any of:

    Type: string from ["int8", "uint8", "int16", "uint16", "int32", "uint32", "int64", "uint64", "float32", "float64", "complex64", "complex128", "bool8"].

    Type: array.

    Items:

        index[0]

        Type: string from ["ascii", "ucs4"].

        index[1]

        Type: integer $\geq$ 0.

datatype

Type: *definitions/scalar-datatype* (page 14) or array of (*definitions/scalar-datatype* (page 14) or object).

The data format of the array elements. May be a single scalar datatype, or may be a nested list of datatypes. When a list, each field may have a name.

Any of:

    Type: *definitions/scalar-datatype* (page 14).

    Type: array of (*definitions/scalar-datatype* (page 14) or object).

    Items:

        Type: *definitions/scalar-datatype* (page 14) or object.

        Any of:

            Type: *definitions/scalar-datatype* (page 14).

            Type: object.

            Properties:

name

Type: string (regex [A-Za-z_][A-Za-z0-9_]*).

The name of the field

datatype

Type: *definitions/datatype* (page 15). Required.

byteorder

Type: string from ["big", "little"].

The byteorder for the field. If not provided, the byteorder of the datatype as a whole will be used.

shape

Type: array of (integer ≥ 0).

Items:

Type: integer ≥ 0.

inline-data

Type: array of (number or string or null or complex (page 14) or *definitions/inline-data* (page 16) or boolean).

Inline data is stored in YAML format directly in the tree, rather than referencing a binary block. It is made out of nested lists.

If the datatype of the array is not specified, it is inferred from the array contents. Type inference is supported only for homogeneous arrays, not tables.

- If any of the elements in the array are YAML strings, the datatype of the entire array is ucs4, with the width of the largest string in the column, otherwise...

- If any of the elements in the array are complex numbers, the datatype of the entire column is complex128, otherwise...

- If any of the types in the column are numbers with a decimal point, the datatype of the entire column is float64, otherwise..

- If any of the types in the column are integers, the datatype of the entire column is int64, otherwise...

- The datatype of the entire column is bool8.

Items:

Type: number or string or null or complex (page 14) or *definitions/inline-data* (page 16) or boolean.

Any of:

Type: number.

Type: string.

Type: null.

Type: complex (page 14).

Type: *definitions/inline-data* (page 16).

Type: boolean.

Any of:

Type: *definitions/inline-data* (page 16).

Type: object.

Properties:

`source`

Type: integer or string (format uri).

The source of the data.

- If an integer: If positive, the zero-based index of the block within the same file. If negative, the index from the last block within the same file. For example, a source of `-1` corresponds to the last block in the same file.

- If a string, a URI to an external ASDF file containing the block data. Relative URIs and `file:` and `http:` protocols must be supported. Other protocols may be supported by specific library implementations.

The ability to reference block data in an external ASDF file is intentionally limited to the first block in the external ASDF file, and is intended only to support the needs of *exploded* (page 8). For the more general case of referencing data in an external ASDF file, use tree *references* (page 10).

Any of:

Type: integer.

Type: string (format uri).

`data`

Type: *definitions/inline-data* (page 16).

The data for the array inline.

If `datatype` and/or `shape` are also provided, they must match the data here and can be used as a consistency check. `strides`, `offset` and `byteorder` are meaningless when `data` is provided.

`shape`

Type: array of (integer ≥ 0 or any from ["*"]).

The shape of the array.

The first entry may be the string `*`, indicating that the length of the first index of the array will be automatically determined from the size of the block. This is used for streaming support.

Items:

Type: integer ≥ 0 or any from ["*"].

Any of:

Type: integer ≥ 0.

Type: any from ["*"].

datatype

Type: *definitions/datatype* (page 15).

The data format of the array elements.

byteorder

Type: string from ["big", "little"].

The byte order (big- or little-endian) of the array data.

offset

Type: integer ≥ 0.

The offset, in bytes, within the data for this start of this view.

Default: 0

strides

Type: array of (integer ≥ 1 or integer ≤ -1).

The number of bytes to skip in each dimension. If not provided, the array is assumed by be contiguous and in C order. If provided, must be the same length as the shape property.

Items:

Type: integer ≥ 1 or integer ≤ -1.

Any of:

Type: integer ≥ 1.

Type: integer ≤ -1.

mask

Type: number or complex (page 14) or ndarray (page 14).

Describes how missing values in the array are stored. If a scalar number, that number is used to represent missing values. If an ndarray, the given array provides a mask, where non-zero values represent missing values in this array. The mask array must be broadcastable to the dimensions of this array.

Any of:

Type: number.

Type: complex (page 14).

```
---
```

Type: ndarray (page 14).

Examples:

An inline array, with implicit data type:

```
!core/ndarray
  [[1, 0, 0],
   [0, 1, 0],
   [0, 0, 1]]
```

An inline array, with an explicit data type:

```
!core/ndarray
  datatype: float64
  data:
    [[1, 0, 0],
     [0, 1, 0],
     [0, 0, 1]]
```

An inline table, where the types of each column are automatically detected:

```
!core/ndarray
  [[M110, 110, 205, And],
   [ M31,  31, 224, And],
   [ M32,  32, 221, And],
   [M103, 103, 581, Cas]]
```

An inline table, where the types of each column are explicitly specified:

```
!core/ndarray
  datatype: [['ascii', 4], uint16, uint16, ['ascii', 4]]
  data:
    [[M110, 110, 205, And],
     [ M31,  31, 224, And],
     [ M32,  32, 221, And],
     [M103, 103, 581, Cas]]
```

A double-precision array, in contiguous memory in a block within the same file:

```
!core/ndarray
  source: 0
  shape: [1024, 1024]
  datatype: float64
  byteorder: little
```

A view of a tile in that image:

```
!core/ndarray
  source: 0
  shape: [256, 256]
  datatype: float64
  byteorder: little
  strides: [8192, 8]
  offset: 2099200
```

A table datatype, with nested columns for a coordinate in (*ra*, *dec*), and a 3x3 convolution kernel:

```
!core/ndarray
  source: 0
  shape: [64]
  datatype:
    - name: coordinate
      datatype:
        - name: ra
          datatype: float64
        - name: dec
          datatype: float64
    - name: kernel
      datatype: float32
      shape: [3, 3]
  byteorder: little
```

An array in Fortran order:

```
!core/ndarray
  source: 0
  shape: [1024, 1024]
  datatype: float64
  byteorder: little
  strides: [8192, 8]
```

An array where values of -999 are treated as missing:

```
!core/ndarray
  source: 0
  shape: [256, 256]
  datatype: float64
  byteorder: little
  mask: -999
```

An array where another array is used as a mask:

```
!core/ndarray
  source: 0
  shape: [256, 256]
  datatype: float64
  byteorder: little
  mask: !core/ndarray
    source: 1
    shape: [256, 256]
    datatype: bool8
    byteorder: little
```

An array where the data is stored in the first block in another ASDF file.:

```
!core/ndarray
  source: external.asdf
  shape: [256, 256]
  datatype: float64
  byteorder: little
```

4.1.4 constant: Specify that a value is a constant.

Type: any.

Specify that a value is a constant.

Used as a utility to indicate that value is a literal constant.

4.2 FITS

The `fits` module contains schema that support backward compatibility with FITS.

Requires:

Core (page 13)

4.2.1 fits: A FITS file inside of an ASDF file.

Type: array of (object).

A FITS file inside of an ASDF file.

This schema is useful for distributing ASDF files that can automatically be converted to FITS files by specifying the exact content of the resulting FITS file.

Not all kinds of data in FITS are directly representable in ASDF. For example, applying an offset and scale to the data using the BZERO and BSCALE keywords. In these cases, it will not be possible to store the data in the native format from FITS and also be accessible in its proper form in the ASDF file.

Items:

    Type: object.

    Each item represents a single header/data unit (HDU).

    Properties:

        header

        Type: array of (array $0 \leq len \leq 3$). Required.

        A list of the keyword/value/comment triples from the header, in the order they appear in the FITS file.

        Items:

            Type: array $0 \leq len \leq 3$.

            Items:

                index[0]

                Type: string ($len \leq 8$ regex [A-Z0-9]*).

                The keyword.

                index[1]

                Type: string ($len \leq 60$) or number or boolean.

                The value.

                Any of:

                    Type: string ($len \leq 60$).

Type: number.

Type: boolean.

`index[2]`

Type: string ($len \leq 60$).

The comment.

`data`

Type: ndarray (page 14) or null.

The data part of the HDU.

Default: null

Any of:

Type: ndarray (page 14).

Type: null.

Examples:

A simple FITS file with a primary header and two extensions:

```
!fits/fits
  - header:
    - [SIMPLE, true, conforms to FITS standard]
    - [BITPIX, 8, array data type]
    - [NAXIS, 0, number of array dimensions]
    - [EXTEND, true]
    - []
    - ['', Top Level MIRI Metadata]
    - []
    - [DATE, '2013-08-30T10:49:55.070373', The date this file was created (UTC)]
    - [FILENAME, MiriDarkReferenceModel_test.fits, The name of the file]
    - [TELESCOP, JWST, The telescope used to acquire the data]
    - []
    - ['', Information about the observation]
    - []
    - [DATE-OBS, '2013-08-30T10:49:55.000000', The date the observation was made (UTC)]
  - data: !core/ndarray
      datatype: float32
      shape: [2, 3, 3, 4]
      source: 0
      byteorder: big
    header:
    - [XTENSION, IMAGE, Image extension]
    - [BITPIX, -32, array data type]
    - [NAXIS, 4, number of array dimensions]
    - [NAXIS1, 4]
    - [NAXIS2, 3]
    - [NAXIS3, 3]
    - [NAXIS4, 2]
    - [PCOUNT, 0, number of parameters]
```

```
      - [GCOUNT, 1, number of groups]
      - [EXTNAME, SCI, extension name]
      - [BUNIT, DN, Units of the data array]
   - data: !core/ndarray
      datatype: float32
      shape: [2, 3, 3, 4]
      source: 1
      byteorder: big
    header:
    - [XTENSION, IMAGE, Image extension]
    - [BITPIX, -32, array data type]
    - [NAXIS, 4, number of array dimensions]
    - [NAXIS1, 4]
    - [NAXIS2, 3]
    - [NAXIS3, 3]
    - [NAXIS4, 2]
    - [PCOUNT, 0, number of parameters]
    - [GCOUNT, 1, number of groups]
    - [EXTNAME, ERR, extension name]
    - [BUNIT, DN, Units of the error array]
```

4.3 Unit

The `unit` module contains schema to support the units of physical quantities.

Requires:

Core (page 13)

4.3.1 unit: Physical unit.

Type: any or any.

Physical unit.

This represents a physical unit, in VOUnit syntax, Version 1.0 (http://www.ivoa.net/documents/VOUnits/index.html).
Where units are not explicitly tagged, they are assumed to be in VOUnit syntax.

Any of:

> ---
>
> Type: any.
>
> ---
>
> Type: any.

Examples:

Example unit:

```
!unit/unit "2.1798721 10-18kg m2 s-2"
```

4.3.2 defunit: Define a new physical unit.

Type: object.

Define a new physical unit.

Defines a new unit. It can be used to either:

- Define a new base unit.

- Create a new unit name that is a equivalent to a given unit.

The new unit must be defined before any unit tags that use it.

Properties:

name

Type: string (regex [A-Za-z_][A-Za-z0-9_]+). Required.

The name of the new unit.

unit

Type: unit (page 23) or null.

The unit that the new name is equivalent to. It is optional, and if not provided, or null, this defunit defines a new base unit.

Any of:

Type: unit (page 23).

Type: null.

4.4 Transform

The transform module contains schema used to describe transformations.

Requires:

Core (page 13)

4.4.1 Basics

transform: A generic type used to mark where other transforms are accepted.

Type: object or any.

A generic type used to mark where other transforms are accepted.

These objects are designed to be nested in arbitrary ways to build up transformation pipelines out of a number of low-level pieces.

Any of:

Type: object.

Properties:

name

Type: string.

A user-friendly name for the transform, to give it extra meaning.

domain

Type: array of (domain (page 26)).

The domain (range of valid inputs) to the transform. Each entry in the list corresponds to an input dimension.

Items:

Type: domain (page 26).

inverse

Type: transform (page 24).

Explicitly sets the inverse transform of this transform.

If the transform has a direct analytic inverse, this property is usually not necessary, as the ASDF-reading tool can provide it automatically.

Type: any.

generic: A generic transform.

Type: transform (page 24) and object.

A generic transform.

This is used **entirely** for bootstrapping purposes so one can create composite models including transforms that haven't yet been written. **IT WILL NOT BE IN THE FINAL VERSION OF THE SPEC.**

All of:

0

Type: transform (page 24).

1

Type: object.

Properties:

n_inputs

Type: integer. Required.

n_outputs

Type: integer. Required.

identity: The identity transform.

Type: transform (page 24) and object.

The identity transform.

Invertibility: The inverse of this transform is also the identity transform.

All of:

> 0
>
> Type: transform (page 24).
>
> 1
>
> Type: object.
>
> **Properties:**
>
>> n_dims
>>
>> Type: integer.
>>
>> The number of dimensions.
>>
>> Default: 1

constant: A transform that takes no inputs and always outputs a constant value.

Type: transform (page 24) and object.

A transform that takes no inputs and always outputs a constant value.

Invertibility: All ASDF tools are required to be able to compute the analytic inverse of this transform, which always outputs zero values.

All of:

> 0
>
> Type: transform (page 24).
>
> 1
>
> Type: object.
>
> **Properties:**
>
>> value
>>
>> Type: number. Required.

domain: Defines the domain of an input axis.

Type: any.

Defines the domain of an input axis.

Describes the range of acceptable input values to a particular axis of a transform.

Examples:

The domain [0, 1).:

```
!transform/domain
  lower: 0
  upper: 1
  includes_lower: true
```

4.4.2 Compound transformations

compose: Perform a list of subtransforms in series.

Type: transform (page 24) and any.

Perform a list of subtransforms in series.

The output of each subtransform is fed into the input of the next subtransform.

The number of output dimensions of each subtransform must be equal to the number of input dimensions of the next subtransform in list. To reorder or add/drop axes, insert remap_axes transforms in the subtransform list.

Invertibility: All ASDF tools are required to be able to compute the analytic inverse of this transform, by reversing the list of transforms and applying the inverse of each.

All of:

0

Type: transform (page 24).

1

Type: any.

Examples:

A series of transforms:

```
!transform/compose
  forward:
    - !transform/generic
      n_inputs: 1
      n_outputs: 2
    - !transform/generic
      n_inputs: 2
      n_outputs: 1
```

concatenate: Send axes to different subtransforms.

Type: transform (page 24) and any.

Send axes to different subtransforms.

Transforms a set of separable inputs by splitting the axes apart, sending them through the given subtransforms in parallel, and finally concatenating the subtransform output axes back together.

The input axes are assigned to each subtransform in order. If the number of input axes is unequal to the sum of the number of input axes of all of the subtransforms, that is considered an error case.

The output axes from each subtransform are appended together to make up the resulting output axes.

For example, given 5 input axes, and 3 subtransforms with the following orders:

1. transform A: 2 in -> 2 out

2. transform B: 1 in -> 2 out

3. transform C: 2 in -> 1 out

The transform is performed as follows:

```
:    i0    i1         i2        i3    i4
:    |     |          |         |     |
:  +---------+ +---------+ +----------+
:  |    A    | |    B    | |    C     |
:  +---------+ +---------+ +----------+
:    |     |     |     |         |
:    o0    o1    o2    o3        o4
```

If reordering of the input or output axes is required, use in series with the `remap_axes` transform.

Invertibility: All ASDF tools are required to be able to compute the analytic inverse of this transform.

All of:

0

Type: transform (page 24).

1

Type: any.

Examples:

The example in the description:

```
!transform/concatenate
  forward:
    - !transform/generic
      n_inputs: 2
      n_outputs: 2
    - !transform/generic
      n_inputs: 1
      n_outputs: 2
    - !transform/generic
      n_inputs: 2
      n_outputs: 1
```

remap_axes: Reorder, add and drop axes.

Type: transform (page 24) and any.

Reorder, add and drop axes.

This transform allows the order of the input axes to be shuffled and returned as the output axes.

It is a list made up of integers or "constant markers". Each item in the list corresponds to an output axis. For each item:

- If an integer, it is the index of the input axis to send to the output axis.

- If a constant, it must be a single item which is a constant value to send to the output axis.

If only a list is provided, the number of input axes is automatically determined from the maximum index in the list. If an object with `mapping` and `n_inputs` properties is provided, the number of input axes is explicitly set by the `n_inputs` value.

Invertibility: TBD

Definitions:

> `mapping`
>
> Type: array of (integer or constant (page 20)).
>
> **Items**:
>
>> Type: integer or constant (page 20).
>>
>> **Any of**:
>>
>>> ---
>>>
>>> Type: integer.
>>>
>>> ---
>>>
>>> Type: constant (page 20).

All of:

> 0
>
> Type: transform (page 24).
>
> 1
>
> Type: any.

Examples:

For 2 input axes, swap the axes:

```
!transform/remap_axes
  mapping: [1, 0]
```

For 2 input axes, return the second axis and drop the first:

```
!transform/remap_axes
  mapping: [1]
```

For 2 input axes, return the first axis twice, followed by the second:

```
!transform/remap_axes
  mapping: [0, 0, 1]
```

For 2 input axes, add a third axis which is a constant:

```
!transform/remap_axes
  mapping: [0, 1, !core/constant 42]
```

The above example is equivalent to the following, and ASDF implementations are free to normalize it thusly:

```
!transform/concatenate
  forward:
    - !transform/remap_axes
      mapping: [0]
    - !transform/remap_axes
```

```
    mapping: [1]
  - !transform/constant
    value: 42
```

Here we have 3 input axes, but we are explicitly dropping the last one:

```
!transform/remap_axes
  mapping: [0, 1]
  n_inputs: 3
```

4.4.3 Arithmetic operations

add: Perform a list of subtransforms in parallel and then add their results together.

Type: transform (page 24) and any.

Perform a list of subtransforms in parallel and then add their results together.

Each of the subtransforms must have the same number of inputs and outputs.

All of:

0

Type: transform (page 24).

1

Type: any.

Examples:

A list of transforms, performed in parallel and added together:

```
!transform/add
  forward:
    - !transform/generic
      n_inputs: 1
      n_outputs: 2
    - !transform/generic
      n_inputs: 1
      n_outputs: 2
```

subtract: Perform a list of subtransforms in parallel and then subtract their results.

Type: transform (page 24) and any.

Perform a list of subtransforms in parallel and then subtract their results.

Each of the subtransforms must have the same number of inputs and outputs.

Invertibility: This transform is not automatically invertible.

All of:

0

Type: transform (page 24).

1

Type: any.

Examples:

A list of transforms, performed in parallel, and then combined through subtraction.:

```
!transform/subtract
  forward:
    - !transform/generic
      n_inputs: 1
      n_outputs: 2
    - !transform/generic
      n_inputs: 1
      n_outputs: 2
```

multiply: Perform a list of subtransforms in parallel and then multiply their results.

Type: transform (page 24) and any.

Perform a list of subtransforms in parallel and then multiply their results.

Each of the subtransforms must have the same number of inputs and outputs.

Invertibility: This transform is not automatically invertible.

All of:

0

Type: transform (page 24).

1

Type: any.

Examples:

A list of transforms, performed in parallel, and then combined through multiplication.:

```
!transform/multiply
  forward:
    - !transform/generic
      n_inputs: 1
      n_outputs: 2
    - !transform/generic
      n_inputs: 1
      n_outputs: 2
```

divide: Perform a list of subtransforms in parallel and then divide their results.

Type: transform (page 24) and any.

Perform a list of subtransforms in parallel and then divide their results.

Each of the subtransforms must have the same number of inputs and outputs.

Invertibility: This transform is not automatically invertible.

All of:

0

Type: transform (page 24).

1

Type: any.

Examples:

A list of transforms, performed in parallel, and then combined through division.:

```
!transform/divide
  forward:
    - !transform/generic
      n_inputs: 1
      n_outputs: 2
    - !transform/generic
      n_inputs: 1
      n_outputs: 2
```

power: Perform a list of subtransforms in parallel and then raise each result to the power of the next.

Type: transform (page 24) and any.

Perform a list of subtransforms in parallel and then raise each result to the power of the next.

Each of the subtransforms must have the same number of inputs and outputs.

Invertibility: This transform is not automatically invertible.

All of:

0

Type: transform (page 24).

1

Type: any.

4.4.4 Simple Transforms

shift: A Shift opeartion.

Type: object.

A Shift opeartion.

Apply an offset in one direction.

Properties:

offset

Type: number.

Offset in one direction.

scale: A Scale model.

Type: object.

A Scale model.

Multiply the input by a factor.

Properties:

factor

Type: number.

Multiplication factor.

4.4.5 Projections

Affine

affine: An affine transform.

Type: transform (page 24) and object.

An affine transform.

Invertibility: All ASDF tools are required to be able to compute the analytic inverse of this transform.

All of:

0

Type: transform (page 24).

1

Type: object.

Properties:

matrix

Type: ndarray (page 14) or array of (array of (number) $len = 2$) $len = 2$. Required.

An array of size (n x n), where n is the number of axes, representing the linear transformation in an affine transform.

Any of:

Type: ndarray (page 14).

Type: array of (array of (number) $len = 2$) $len = 2$.

Items:

Type: array of (number) $len = 2$.

Items:

Type: number.

> translation

Type: ndarray (page 14) or array of (number) *len* = 2.

An array of size (*n*,), where *n* is the number of axes, representing the translation in an affine transform.

Any of:

> ---

> Type: ndarray (page 14).

> ---

> Type: array of (number) *len* = 2.

> **Items**:

> > Type: number.

rotate2d: A 2D rotation.

Type: transform (page 24) and object.

A 2D rotation.

A 2D rotation around the origin, in degrees. Invertibility: All ASDF tools are required to be able to compute the analytic inverse of this transform.

All of:

> 0

> Type: transform (page 24).

> 1

> Type: object.

> **Properties**:

> > angle

> > Type: number. Required.

> > Angle, in degrees.

rotate3d: Rotation in 3D space.

Type: transform (page 24) and object.

Rotation in 3D space.

Euler angle rotation around 3 axes.

Invertibility: All ASDF tools are required to be able to compute the analytic inverse of this transform.

All of:

> 0

> Type: transform (page 24).

> 1

> Type: object.

Properties:

phi

Type: number. Required.

Angle, in degrees.

theta

Type: number. Required.

Angle, in degrees.

psi

Type: number. Required.

Angle, in degrees.

direction

Type: any from [”zxz”, “zyz”, “yzy”, “yxy”, “xyx”, “xzx”, “native2celestial”, “celestial2native”]. Required.

Sequence of rotation axes: one of zxz, zyz, yzy, yxy, xyx, xzx or native2celestial, celestial2native.

If direction is native2celestial or celestial2native, phi, theta are the longitude and latitude of the native pole in the celestial system and psi is the longitude of the celestial pole in the native system.

Default: “native2celestial”

Examples:

The three Euler angles are 12.3, 34 and -1.2 in degrees.:

```
!transform/rotate3d
  phi: 12.3
  theta: 34
  psi: -1.2
  direction: zxz
```

Zenithal (azimuthal)

zenithal: Base class of all zenithal (or azimuthal) projections.

Type: transform (page 24) and object.

Base class of all zenithal (or azimuthal) projections.

Zenithal projections are completely specified by defining the radius as a function of native latitude, R_θ.

The pixel-to-sky transformation is defined as:

$$\phi = \arg(-y, x)$$
$$R_\theta = \sqrt{x^2 + y^2}$$

and the inverse (sky-to-pixel) is defined as:

$$x = R_\theta \sin \phi$$
$$y = R_\theta \cos \phi$$

All of:

```
0
```

Type: transform (page 24).

```
1
```

Type: object.

Properties:

```
direction
```

Type: any from ["pix2sky", "sky2pix"].

Default: "pix2sky"

gnomonic: The gnomonic projection.

Type: zenithal (page 35).

The gnomonic projection.

Corresponds to the TAN projection in the FITS WCS standard.

See *zenithal* (page 35) for the definition of the full transformation.

The pixel-to-sky transformation is defined as:

$$\theta = \tan^{-1}\left(\frac{180°}{\pi R_\theta}\right)$$

And the sky-to-pixel transformation is defined as:

$$R_\theta = \frac{180°}{\pi}\cot\theta$$

Invertibility: All ASDF tools are required to provide the inverse of this transform.

zenithal_perspective: The zenithal perspective projection.

Type: zenithal (page 35) and object.

The zenithal perspective projection.

Corresponds to the AZP projection in the FITS WCS standard.

The pixel-to-sky transformation is defined as:

$$\phi = \arg(-y\cos\gamma, x)$$
$$\theta = \begin{cases} \psi - \omega \\ \psi + \omega + 180° \end{cases}$$

where:

$$\psi = \arg(\rho, 1)$$
$$\omega = \sin^{-1}\left(\frac{\rho\mu}{\sqrt{\rho^2 + 1}}\right)$$
$$\rho = \frac{R}{\frac{180°}{\pi}(\mu + 1) + y\sin\gamma}$$
$$R = \sqrt{x^2 + y^2\cos^2\gamma}$$

And the sky-to-pixel transformation is defined as:

$$x = R \sin \phi$$
$$y = -R \sec \gamma \cos \theta$$

where:

$$R = \frac{180°}{\pi} \frac{(\mu + 1) \cos \theta}{(\mu + \sin \theta) + \cos \theta \cos \phi \tan \gamma}$$

Invertibility: All ASDF tools are required to provide the inverse of this transform.

All of:

  0

  Type: zenithal (page 35).

  1

  Type: object.

  Properties:

    mu

    Type: number.

    Distance from point of projection to center of sphere in spherical radii.

    Default: 0

    gamma

    Type: number.

    Look angle, in degrees.

    Default: 0

stereographic: The stereographic projection.

Type: zenithal (page 35).

The stereographic projection.

Corresponds to the STG projection in the FITS WCS standard.

See *zenithal* (page 35) for the definition of the full transformation.

The pixel-to-sky transformation is defined as:

$$\theta = 90° - 2 \tan^{-1} \left(\frac{\pi R_\theta}{360°} \right)$$

And the sky-to-pixel transformation is defined as:

$$R_\theta = \frac{180°}{\pi} \frac{2 \cos \theta}{1 + \sin \theta}$$

Invertibility: All ASDF tools are required to provide the inverse of this transform.

slant_orthographic: The slant orthographic projection.

Type: zenithal (page 35).

The slant orthographic projection.

Corresponds to the SIN projection in the FITS WCS standard.

See *zenithal* (page 35) for the definition of the full transformation.

The pixel-to-sky transformation is defined as:

$$\theta = \cos^{-1}\left(\frac{\pi}{180°}R_\theta\right)$$

And the sky-to-pixel transformation is defined as:

$$R_\theta = \frac{180°}{\pi}\cos\theta$$

Invertibility: All ASDF tools are required to provide the inverse of this transform.

Cylindrical

cylindrical: Base class of all cylindrical projections.

Type: transform (page 24) and object.

Base class of all cylindrical projections.

The surface of cylindrical projections is a cylinder.

All of:

    0

    Type: transform (page 24).

    1

    Type: object.

    Properties:

        direction

        Type: any from ["pix2sky", "sky2pix"].

        Default: "pix2sky"

cylindrical_perspective: The cylindrical perspective projection.

Type: cylindrical (page 38) and object.

The cylindrical perspective projection.

Corresponds to the CYP projection in the FITS WCS standard.

The pixel-to-sky transformation is defined as:

$$\phi = \frac{x}{\lambda}$$

$$\theta = \arg(1, \eta) + \sin^{-1}\left(\frac{\eta\mu}{\sqrt{\eta^2 + 1}}\right)$$

And the sky-to-pixel transformation is defined as:

$$x = \lambda \phi$$
$$y = \frac{180°}{\pi} \left(\frac{\mu + \lambda}{\mu + \cos \theta} \right) \sin \theta$$

Invertibility: All ASDF tools are required to provide the inverse of this transform.

All of:

> 0
>
> Type: cylindrical (page 38).
>
> 1
>
> Type: object.
>
> **Properties:**
>
> > mu
> >
> > Type: number.
> >
> > Distance from center of sphere in the direction opposite the projected surface, in spherical radii.
> >
> > Default: 0
> >
> > lambda
> >
> > Type: number.
> >
> > Radius of the cylinder in spherical radii, default is 0.
> >
> > Default: 0

cylindrical_equal_area: The cylindrical equal area projection.

Type: cylindrical (page 38) and object.

The cylindrical equal area projection.

Corresponds to the CEA projection in the FITS WCS standard.

The pixel-to-sky transformation is defined as:

$$\phi = x$$
$$\theta = \sin^{-1} \left(\frac{\pi}{180°} \lambda y \right)$$

And the sky-to-pixel transformation is defined as:

$$x = \phi$$
$$y = \frac{180°}{\pi} \frac{\sin \theta}{\lambda}$$

Invertibility: All ASDF tools are required to provide the inverse of this transform.

All of:

> 0
>
> Type: cylindrical (page 38).
>
> 1

Type: object.

Properties:

lambda

Type: number.

Radius of the cylinder in spherical radii, default is 0.

Default: 0

plate_carree: The plate carrée projection.

Type: cylindrical (page 38).

The plate carrée projection.

Corresponds to the CAR projection in the FITS WCS standard.

The main virtue of this transformation is its simplicity.

The pixel-to-sky transformation is defined as:

$$\phi = x$$
$$\theta = y$$

And the sky-to-pixel transformation is defined as:

$$x = \phi$$
$$y = \theta$$

Invertibility: All ASDF tools are required to provide the inverse of this transform.

mercator: The Mercator projection.

Type: cylindrical (page 38).

The Mercator projection.

Corresponds to the MER projection in the FITS WCS standard.

The pixel-to-sky transformation is defined as:

$$\phi = x$$
$$\theta = 2 \tan^{-1} \left(e^{y \pi / 180°} \right) - 90°$$

And the sky-to-pixel transformation is defined as:

$$x = \phi$$
$$y = \frac{180°}{\pi} \ln \tan \left(\frac{90° + \theta}{2} \right)$$

Invertibility: All ASDF tools are required to provide the inverse of this transform.

4.4.6 Polynomials

polynomial: A Polynomial model.

Type: object.

A Polynomial model.

A polynomial model represented by its coefficients stored in an ndarray of shape $(n+1)$ for univariate polynomials or $(n+1, n+1)$ for polynomials with 2 variables, where n is the highest total degree of the polynomial.

$$P = \sum_{i,j=0}^{i+j=n} c_{ij} * x^i * y^j$$

Invertibility: This transform is not automatically invertible.

Properties:

coefficients

Type: ndarray (page 14) or array. Required.

An array with coefficients.

Any of:

Type: ndarray (page 14).

Type: array.

Items:

Examples:

$P = 1.2 + 0.3 * x + 56.1 * x^2$:

```
!transform/polynomial
  coefficients: !core/ndarray
                [1.2, 0.3, 56.1]
```

$P = 1.2 + 0.3 * x + 3 * x * y + 2.1 * y^2$:

```
!transform/polynomial
  coefficients: !core/ndarray
                [[1.2, 0.0, 2.1],
                 [0.3, 3.0, 0.0],
                 [0.0, 0.0, 0.0]]
```

4.5 WCS

The WCS module contains schema used to describe generalized world coordinate system transformations.

Requires:

Core (page 13), Unit (page 23), Transform (page 24)

4.5.1 wcs: A system for describing generalized world coordinate transformations.

Type: object.

A system for describing generalized world coordinate transformations.

ASDF WCS is a way of specifying transformations (usually from detector space to world coordinate space and back) by using the transformations in the `transform-schema` module.

Properties:

> steps
>
> Type: steps (page 42). Required.
>
> A list of steps in the forward transformation from detector to world coordinates.
>
> The inverse transformation is determined automatically by reversing this list, and inverting each of the individual transforms according to the rules described in *inverse* (page 25).

4.5.2 steps: A list of WCS steps performed in series that make up a WCS pipeline.

Type: array of (step (page 42)) *len* $\geq$ 2.

A list of WCS steps performed in series that make up a WCS pipeline.

Items:

> Type: step (page 42).

4.5.3 step: Describes a single step of a WCS transform pipeline.

Type: object.

Describes a single step of a WCS transform pipeline.

Properties:

> name
>
> Type: string.
>
> The name of this step on the WCS transform pipeline. This name may be used by a library to allow the user to say "transform from pixel space to the focal plane" without performing the entire WCS pipeline.
>
> axes
>
> Type: array of (axis (page 43)).
>
> A description of each of the input axes for this step.
>
> Items:
>
> > Type: axis (page 43).
>
> transform
>
> Type: transform (page 24).
>
> The transform from this step to the next one. The last step in a WCS should not have a transform, but exists only to describe the frames and units of the final output axes.

4.5.4 axis: Specifies details about an axis in a WCS step.

Type: object.

Specifies details about an axis in a WCS step.

A standard frame name should be used when appropriate, though custom frame names may be used when necessary.

Properties:

type

Type: any from [”celestial”, “time”, “spectral”, “cartesian”, “detector”, “focal_plane”, “custom”].

The type of frame for this axis. It must have one of the following values:

- `celestial`: Celestial coordinates. A `celestial_type` property must also be given.
- `time`: Time coordinate. A `time_scale` property must also be given.
- `spectral`: Spectral coordinate.
- `cartesian`: Cartesian coordinate.
- `detector`: Native detector coordinate.
- `focal_plane`: A coordinate in the focal plane. Naturally, these coordinates will be instrument specific.
- `custom`: A custom axis frame that is not one of the above. Additional information about the frame may be stored in additional properties as a non-standard extension.

celestial_type

Type: any from [”FK4”, “FK5”, “Galactic”, “ICRS”].

If `type` is “celestial”, this property must be given.

- FK4: Celestial coordinates in the FK4 system. `equinox` and `observation_time` may also be given. `name` should be `ra` or `dec`.
- FK5: Celestial coordinates in the FK5 system. `equinox` may also be given. `name` should be `ra` or `dec`.
- ICRS: Celestial coordinates in the ICRS system. `name` should be `ra` or `dec`.
- galactic: Galactic coordinates. `name` should be `lon` or `lat`.

Default: “ICRS”

equinox

Type: string.

The equinox of the celestial frame.

Default: “J2000”

observation_time

Type: string.

The observation time of the celestial frame, if different from `equinox`.

time_scale

Type: any from [”TAI”, “TCB”, “TCG”, “TDB”, “TT”, “UT1”, “UTC”].

The time scale of the time axis. Must be one of:

- TAI: International Atomic Time

- TCB: Barycentric Coordinate Time

- TCG: Geocentric Coordinate Time

- TDB: Barycentric Dynamical Time

- TT: Terrestrial Time

- UT1: Universal Time

- UTC: Coordinated Universal Time

`spectral_type`

Type: any.

TODO

`name`

Type: string or any from ["ra", "dec", "lon", "lat", "x", "y", "z"].

The name of the axis. If the `type` is multidimensional, such as "celestial", this name has meaning, and must correspond to it. For example, for a celestial ICRS frame, it must be either "ra" or "dec". Otherwise, it may be an arbitrary name for the axis.

Any of:

    Type: string.

    Type: any from ["ra", "dec", "lon", "lat", "x", "y", "z"].

`unit`

Type: unit (page 23).

The unit for the axis.

The following graph shows the dependencies between modules:

EXTENDING ASDF

ASDF is designed to be extensible so outside teams can add their own types and structures while retaining compatibility with tools that don't understand those conventions.

5.1 YAML Schema

5.1.1 draft-01: YAML Schema

Type: schema and object.

YAML Schema

A metaschema extending JSON Schema's metaschema to add support for some YAML-specific constructions.

All of:

0

Type: schema.

1

Type: object.

Properties:

tag

Type: string ($len \geq 6$).

A fully-qualified YAML tag name that should be associated with the object type returned by the YAML parser; for example, the object must be an instance of the class registered with the parser to create instances of objects with this tag. Implementation of this validator is optional and depends on details of the YAML parser.

propertyOrder

Type: array of (string).

Specifies the default order of the properties when writing out. Any keys not listed in propertyOrder will be in arbitrary order at the end.

Items:

Type: string.

flowStyle

Type: string from ["block", "flow"].

Specifies the default serialization style to use for an array or object. YAML supports multiple styles for arrays/sequences and objects/maps, called "block style" and "flow style". For example:

```
Block style: !!map
  Clark : Evans
  Ingy  : döt Net
  Oren  : Ben-Kiki


Flow style: !!map { Clark: Evans, Ingy: döt Net, Oren: Ben-Kiki }
```

This property gives a hint to the tool outputting the YAML which style to use. If not provided, the library is free to use whatever heuristics it wishes to determine the output style. This property does not enforce any particular style on YAML being parsed.

style

Type: string from ["inline", "literal", "folded"].

Specifies the default serialization style to use for a string. YAML supports multiple styles for strings:

```
Inline style: "First line\nSecond line"

Literal style: |
  First line
  Second line

Folded style: >
  First
  line
```

```
Second
line
```

This property gives a hint to the tool outputting the YAML which style to use. If not provided, the library is free to use whatever heuristics it wishes to determine the output style. This property does not enforce any particular style on YAML being parsed.

examples

Type: array of (array).

A list of examples to help document the schema. Each pair is a prose description followed by a string containing YAML content.

Items:

> Type: array.

> Items:

>> index[0]

>> Type: string.

>> index[1]

>> Type: string.

YAML Schema (page 45) is a small extension to JSON Schema Draft 4 (http://json-schema.org/latest/json-schema-validation.html) created specifically for ASDF. Understanding JSON Schema (http://spacetelescope.github.io/understanding-json-schema/) provides a good resource for understanding how to use JSON Schema, and further resources are available at json-schema.org (http://json-schema.org). A working

understanding of JSON Schema is assumed for this section, which only describes what makes YAML Schema different from JSON Schema.

Writing a new schema is described in *Designing a new tag and schema* (page 47).

YAML Schema adds three new keywords to JSON Schema.

5.1.2 `tag` keyword

`tag`, which may be attached to any data type, declares that the element must have the given YAML tag.

For example, the root *asdf* (page 13) schema declares that the `data` property must be an *ndarray* (page 14). It does this not by using the `tag` keyword directly, but by referencing the `ndarray` schema, which in turn has the `tag` keyword. The ASDF schema includes:

```
properties:
  data:
    $ref: "ndarray"
```

And the *ndarray* (page 14) schema includes:

```
tag: "tag:stsci.edu:asdf/0.1.0/core/ndarray"
```

This has the net effect of requiring that the `data` property at the top-level of all ASDF files is tagged as `tag:stsci.edu:asdf/0.1.0/core/ndarray`.

5.1.3 `propertyOrder` keyword

`propertyOrder`, which applies only to objects, declares that the object must have its properties presented in the given order.

TBD: It is not yet clear whether this keyword is necessary or desirable.

5.1.4 `examples` keyword

The schema may contain a list of examples demonstrating how to use the schema. It is a list where each item is a pair. The first item in the pair is a prose description of the example, and the second item is YAML content (as a string) containing the example.

For example:

```
examples:
  -
    - Complex number: 1 real, -1 imaginary
    - "!complex 1-1j"
```

5.2 Designing a new tag and schema

The schema included in the ASDF standard will not be adequate for all needs, but it is possible to mix them with custom schema designed for a specific purpose. It is also possible to extend and specialize an existing schema (described in *Extending an existing schema* (page 50)).

This section will walk through the development of a new tag and schema. In the example, suppose we work at the Space Telescope Science Institute, which can be found on the world wide web at `stsci.edu`. We're developing a

new instrument, F00, and we need a way to define the specialized metadata to describe the exposures that it will be generating.

5.2.1 Header

Every ASDF schema should begin with the following header:

```
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
```

This declares that the file is YAML 1.1 format, and that the structure of the content conforms to YAML Schema defined above.

5.2.2 Tags and IDs

All of the tags defined by the ASDF standard itself have the following prefix:

```
tag:stsci.edu:asdf/0.1.0
```

This prefix is reserved for tags and schemas defined within the ASDF standard itself. ASDF can, of course, include any tags, as long as the tag names are globally unique. So, for our example instrument, we'll declare the tag to be:

```
tag:stsci.edu:F00/0.1.0/metadata
```

Each tag should be associated with a schema in order to validate it. Each schema must also have a universally unique id, which is in the form of unique URI. For the ASDF built-in tags, the mapping from tag name to schema URI is quite simple:

```
tag:stsci.edu:XXX
```

maps to:

```
http://stsci.edu/schemas/XXX
```

Note that this URI doesn't actually have to resolve to anything. In fact, visiting that URL in your web browser is likely to bring up a 404 error. All that's necessary is that it is universally unique and that the tool reading the ASDF file is able to map from a tag name to a schema URI, and then load the associated schema.

Again following with our example, we will assign the following URI to refer to our schema:

```
http://stsci.edu/schemas/F00/0.1.0/metadata
```

Therefore, in our schema file, we have the following keys, one declaring the name of the YAML tag, and one defining the id of the schema:

```
tag: "tag:stsci.edu:F00/0.1.0/metadata"
id: "http://stsci.edu/schemas/F00/0.1.0/metadata"
```

5.2.3 Descriptive information

Each schema has some descriptive fields: title, description and examples. These fields may contain core markdown syntax.

- `title`: A one-line summary of what the schema is for.

- `description`: A lengthier prose description of the schema

- `examples`: A list of example content that conforms to the schema, illustrating how to use it.

Continuing our example:

```
title: |
  Metadata for the FOO instrument.
description: |
  This stores some information about an exposure from the FOO instrument.
examples:
  -
    - A minimal description of an exposure.
    - |
        !FOO/0.1.0/metadata
          exposure_time: 0.001
```

5.2.4 The schema proper

The rest of the schema describes the acceptable data types and their structure. The format used for this description comes straight out of JSON Schema, and rather than documenting all of the things it can do here, please refer to Understanding JSON Schema (http://spacetelescope.github.io/understanding-json-schema/), and the further resources available at json-schema.org (http://json-schema.org).

In our example, we'll define two metadata elements: the name of the investigator, and the exposure time, each of which also have a description:

```
type: object
properties:
  investigator:
    type: string
    description: |
      The name of the principal investigator who requested the
      exposure.

  exposure_time:
    type: number
    description: |
      The time of the exposure, in nanoseconds.
```

We'll also define an optional element for the exposure time unit. This is a somewhat contrived example to demonstrate how to include elements in your schema that are based on the custom types defined in the ASDF standard:

```
exposure_time_units:
  $ref: "http://stsci.edu/schemas/asdf/0.1.0/unit/unit"
  description: |
    The unit of the exposure time.
  default:
    s
```

Lastly, we'll declare `exposure_time` as being required, and allow extra elements to be added:

```
requiredProperties: [exposure_time]
additionalProperties: true
```

5.2.5 The complete example

Here is our complete schema example:

```yaml
%YAML 1.1
---
$schema: "http://stsci.edu/schemas/yaml-schema/draft-01"
tag: "tag:stsci.edu:FOO/0.1.0/metadata"
id: "http://stsci.edu/schemas/FOO/0.1.0/metadata"

title: |
  Metadata for the FOO instrument.
description: |
  This stores some information about an exposure from the FOO instrument.
examples:
  -
    - A minimal description of an exposure.
    - |
        !FOO/0.1.0/metadata
          exposure_time: 0.001

type: object
properties:
  investigator:
    type: string
    description: |
      The name of the principal investigator who requested the
      exposure.

  exposure_time:
    type: number
    description: |
      The time of the exposure, in nanoseconds.

  exposure_time_units:
    $ref: "http://stsci.edu/schemas/asdf/0.1.0/unit/unit"
    description: |
      The unit of the exposure time.
    default:
      s

requiredProperties: [exposure_time]
additionalProperties: true
```

5.3 Extending an existing schema

TODO

KNOWN LIMITS

The following is a catalogue of known limits in ASDF 0.1.0dev.

6.1 Tree

While there is no hard limit on the size of the tree, in most practical implementations it will need to be read entirely into main memory in order to interpret it, particularly to support forward references. This imposes a practical limit on its size relative to the system memory on the machine. It is not recommended to store large data sets in the tree directly, instead it should reference blocks.

6.2 Blocks

The maximum size of a block header is 65536 bytes.

Since the size of the block is stored in a 64-bit unsigned integer, the largest possible block size is around 18 exabytes. It is likely that other limitations on file size, such as an operating system's filesystem limitations, will be met long before that.

CHANGES

7.1 Version 0.1.0

First pre-release.

[Thomas2015] Thomas, B., Jenness. T. et al. 2015, "The Future of Astronomical Data Formats I. Learning from FITS". Astronomy & Computing, in press, arXiv e-print: 1502.00996. `https://github.com/timj/aandc-fits`.