# Cornish, A Python Interface to the Starlink AST WCS Library

***Release 1.0a7***

**Demitri Muna**

**Aug 24, 2021**

# CONTENTS

Cornish is a Python interface over the Starlink AST astronomical software library, part of the Starlink Software Collection. Cornish is designed and written by Demitri Muna.

The Starlink AST library is a collection of tools for working with world coordinate systems and excels at coordinate system transformations, representing and working with regions on the celestial sphere (e.g. polygons, circles, boxes, etc.), plotting, and more.

A Python interface called starlink-pyast (`pip install starlink-pyast`) written by the library's authors, David Berry and Tim Jenness, is available, however it thinly exposes the C interface. A working knowledge of the C API is realistically a prerequisite to using `starlink-pyast`. Similarly, the primary documentation for the library is the that of the C version which does not refer to the Python interface.

The aim of Cornish is to be a fully Pythonic interface to the library. It doesn't replace the existing `starlink-pyast` Python interface; rather, it is a wrapper around it. The current development focus is defining and working with regions and coordinate systems on the sky. It accepts and returns Astropy objects where possible. While care is taken to mimic the original API where it makes sense, the library places greater emphasis on making the API as Pythonic as possible and will rename methods where it will make operations and concepts more clear to the Python user.

Cornish is currently under active development and all APIs are subject to change. It is not recommended to be used in a pipeline yet, but it is becoming increasingly mature and particularly useful for interactive use. The project is being released in this state as it is a dependency of the Trillian and SciDD projects, also written by Demitri Muna.

Many thanks to David Berry for the generous and extremely responsive help and advice in the development of the library.

CHAPTER

ONE

# EXAMPLES

## 1.1 Working With Regions

Starlink AST is primarily a library for working with world coordinate systems (WCS) for astronomical data. It has extensive functionality related to coordinate system translations, platting, and more. One of the primary tools of the library (and the initial focus of the Cornish Python interface) is the handling of regions. Regions can be Cartesian, e.g. a pixel grid on a CCD, or else in a celestial sphere frame, e.g. a region on an `ASTSkyFrame`. For the latter, all lines connecting vertices are great circles.

All region objects are all subclassed from *ASTRegion*. See the C library reference documentation for the full details of the objects. Region classes include:

- *ASTCircle* [C AST Reference]
- *ASTPolygon* [C AST Reference]
- *ASTBox* [C AST Reference]
- *ASTCompoundRegion* [C AST Reference]

When creating a region, note that the frame the region to be defined in must be specified. The class `ASTICRSFrame` is provided as a convenience, defined as sky frame in the ICRS system with epoch 2000.0 and equinox 2000.0. Note that regions default to the `ASTICRSFrame` if one is not provided.

Region objects are defined in the top level of the `cornish` namespace.

### 1.1.1 All Regions

The *ASTRegion* class is the superclass for all Cornish region objects. It contains a great deal of functionality. The API reference is a worth looking at to see what is available. A short listing:

- *points*: Return a list of points that describe the region. The shape is dependent on the type of region.
- *boundingCircle()*: Returns a new *ASTCircle* object that bounds the given region.
- *overlaps()*: Check whether two regions overlap.
- *isIdenticalTo()*: Check whether two regions are identical.
- *isFullyWithin()*: Check whether one region is fully within another.
- *fullyEncloses()*: Check whether one region is fully encloses another.
- *boundaryPointMesh()*: Returns an array of evenly distributed points that cover the boundary of the region.
- *interiorPointMesh()*: Returns an array of evenly distributed points that cover the surface of the region.
- *containsPoint()*: Determine whether a given point lies within a given region.

See the API reference for more methods and properties.

### 1.1.2 Circles

#### Creating Circles

Circles can be defined as either a center point and a radius or else a center point and another on the circumference. Coordinates can be specified an `astropy.coodinates.SkyCoord` object or pairs of values in degrees. The examples below demonstrate various ways to create circle regions.

```
from cornish import ASTCircle, ASTICRSFrame, ASTSkyFrame
from cornish.constants import SYSTEM_GALACTIC, EQUINOX_J2010
from astropy.coordinates import SkyCoord
import astropy.units as u

# note that the default frame is ICRS, epoch=2000.0, equinox=2000.0

# defined as center + radius
# --------------------------

# using Astropy objects
center = SkyCoord(ra="12d42m22s", dec="-32d18m58s")
circle = ASTCircle(center=center, radius=2.0*u.deg)

# using float values, defaults to degrees
circle = ASTCircle(center=[12.7061, -31.6839], radius=2.0) # assumes degrees
circle = ASTCircle(center=[12.7061*u.deg, -31.6839*u.deg], radius=2.0*u.deg) #␣
↪Quanitites also accepted

# defined as center + circumference point
# ---------------------------------------
circle = ASTCircle(center=center, edge_point=[12.7061, -32.6839]) # edge_point also␣
↪takes SkyCoord

# define the circle in another frame
# ----------------------------------
gal_frame = ASTSkyFrame(system=SYSTEM_GALACTIC)
gal_frame.equinox = EQUINOX_J2010
ASTCircle(frame=gal_frame, center=center, radius=2.0*u.deg)
```

#### Circle Properties

Circles have *radius* and *centre* properties as one might expect, and both can be directly modified:

```
circle.radius
>>> <Quantity 2. deg>

circle.centre # or "center" if you prefer...
>>> array([ 12.70611111, -32.31611111]) # output in degrees
```

New circles can be created by a scale factor or increased by addition from an existing circle.

```
scaled_circle = circle * 2.0
scaled_circle.radius
>>> <Quantity 4. deg>

larger_circle = circle + 6*u.deg
larger_circle.radius
>>> <Quantity 8. deg>
```

### Converting to Polygons

For code that requires a polygon region as an input the method `toPolygon()` will convert a circle to an `ASTPolygon`. The default is to sample 200 points for the polygon but this can be customized by using the *npoints* parameter (often even 20 are sufficient). Note that all of the polygon points fall on the circle's circumference, so the resulting region is fully inscribed by the original circle.

```
polygon = circle.toPolygon()
finer_polygon = circle(toPolygon(npoints=200))
```

All regions have a `boundingCircle()` property that returns an `ASTCircle` that bounds the region. In the case of `ASTCircle` objects, this method returns the original circle.

## 1.1.3 Polygons

A polygon is a collection of vertices that lie in a specific frame. The default frame `ASTICRSFrame` is used if none is specified.

```
from cornish import ASTPolygon, ASTICRSFrame
import numpy as np

points = np.array([[ 12.70611111, -30.31611111],
                   [ 13.42262189, -30.41196836],
                   [ 14.07300863, -30.69069244],
                   [ 14.59623325, -31.12642801],
                   [ 14.94134955, -31.67835614],
                   [ 15.07227821, -32.29403528],
                   [ 14.97204342, -32.91392471],
                   [ 14.6459242 , -33.47688136],
                   [ 14.12273328, -33.92626054],
                   [ 13.4533703 , -34.21603194],
                   [ 12.70611111, -34.31611111],
                   [ 11.95885193, -34.21603194],
                   [ 11.28948894, -33.92626054],
                   [ 10.76629802, -33.47688136],
                   [ 10.4401788 , -32.91392471],
                   [ 10.33994401, -32.29403528],
                   [ 10.47087267, -31.67835614],
                   [ 10.81598897, -31.12642801],
                   [ 11.3392136 , -30.69069244],
                   [ 11.98960033, -30.41196836]])
polygon = ASTPolygon(frame=ASTICRSFrame(), points=points)
```

Points can be specified as an array of coordinate points (as above) or as parallel arrays of each dimension (which is just `points.T` from above):

```
points = np.array([[ 12.70611111,  13.42262189,  14.07300863,  14.59623325, ...],
                   [-30.31611111, -30.41196836, -30.69069244, -31.12642801, ...]])
```

**Todo:** Provide example of how to convert a region from one frame to another.

### 1.1.4 Boxes

**Todo:** Box section coming soon! (But it's pretty straightforward from the *ASTBox* API.)

### 1.1.5 Compound Regions

**Todo:** Compound regions section coming soon! (But it's pretty straightforward from the *ASTBox* API.)

### 1.1.6 From FITS Files

Cornish is able to create regions based on image FITS headers alone. The example below shows how to create a region object based on the area covered by a FITS image from the header. The example file below can be downloaded here.

```
from cornish import ASTPolygon
from astropy.io import fits

filename = "frame-g-006174-2-0094.fits.bz2"
with fits.open(filename) as hdu_list:
    hdu1 = hdu_list[0]

polygon = ASTPolygon.fromFITSHeader(hdu1.header)
```

## 1.2 Plotting Examples

The underlying starlink-pyast library has a rich functionality for handling plotting on multiple world coordinate systems. The interface is extensible and provides hooks for custom plot interfaces.

Plotting support in Cornish is not complete, but more than sufficient for checks like verifying regions.

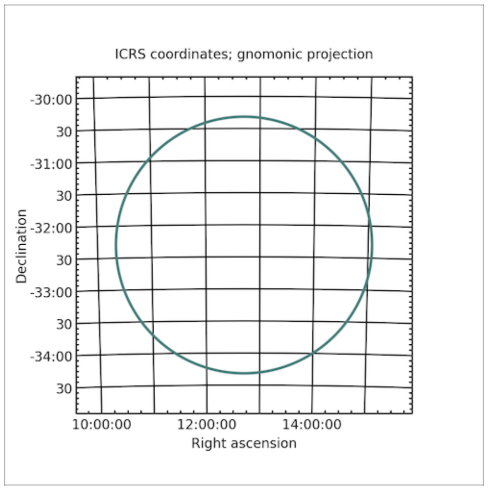
### 1.2.1 Matplotlib Interface

starlink-pyast provides a wrapper for plotting with Matplotlib. Cornish goes further by creating routines that overlay these to interact with the Cornish objects in as simple a manner as possible. The primary interface is the *SkyPlot* object.

The following is a very simple example of how to plot a circle in an ICRS frame on the sky:

```python
from cornish import ASTCircle
from cornish.plot.matplotlib import SkyPlot
import astropy.units as u
from astropy.coordinates import SkyCoord

# define a circle in the ICRS frame (used by default)
center = SkyCoord(ra="12d42m22s", dec="-32d18m58s")
circle = ASTCircle(center=center, radius=2.0*u.deg)

# define a new plot of 5x5 inches
# set the extent of the plot to the size of a circle
# with 1.25 x the circle radius to leave some room
skyplot = SkyPlot(extent=circle*1.25, figsize=(5,5))

# add region to plot
skyplot.addRegionOutline(circle)

# display
skyplot.show()
```

The result is:



Repeat the same with a polygon converted from the circle.

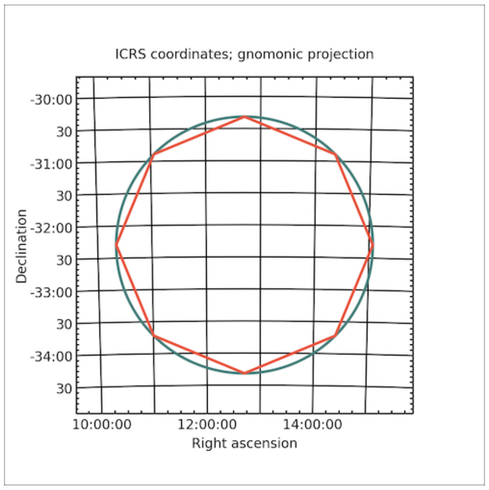
```python
from cornish import ASTCircle
from cornish.plot.matplotlib import SkyPlot
import astropy.units as u
from astropy.coordinates import SkyCoord

# define a circle in the ICRS frame (used by default)
center = SkyCoord(ra="12d42m22s", dec="-32d18m58s")
```

(continues on next page)

(continued from previous page)

```
circle = ASTCircle(center=center, radius=2.0*u.deg)

# define a new plot of 5x5 inches
# set the extent of the plot to the size of a circle
# with 1.25 x the circle radius to leave some room
skyplot = SkyPlot(extent=circle*1.25, figsize=(5,5))

# add region to plot
skyplot.addRegionOutline(circle)
skyplot.addRegionOutline(circle.toPolygon(npoints=8), color="#3b85f7")

# display
skyplot.show()
```

The result is:



**Todo:** Link to Trillian docs to demonstrate plotting via the Trillian API.

## 1.2.2 Other Interfaces

Currently only the Matplotlib interface is supported. More are planned.

CHAPTER

TWO

# CORNISH API

## 2.1 Region APIs

Classes that describe regions are documented here.

### 2.1.1 ASTRegion

**class** cornish.**ASTRegion**(*ast_object=None*, *uncertainty=None*)

Bases: *cornish.mapping.frame.frame.ASTFrame*

Represents a region within a coordinate system. This is an abstract superclass - there is no supported means to create an ASTRegion object directly (see *ASTBox*, *ASTPolygon*, etc.).

Accepted signatures for creating an ASTRegion:

```
r = ASTRegion(ast_object)
```

**Parameters**

- **ast_object** (Optional[Region]) –
- **uncertainty** –

**angle**(*vertex=None*, *points=None*)

Calculate the angle in this frame between two line segments connected by a point.

Let A = point1, C = point2, and B = the vertex point. This method calculates the angle between the line segments AB and CB.

If the frame is a sky frame, lines are drawn on great circles. Units are assumed to be degrees if not provided with units, e.g. as an astropy.coordinates.SkyCoord or astropy.units.Quantity values.

**Parameters**

- **vertex** (Optional[Iterable]) – a two element list/tuple/Numpy array or a SkyCoord of the vertex
- **points** (Optional[Container[Union[SkyCoord, Iterable]]]) – a two element list/tuple/etc. containing two points in this frame

**Return type** Quantity

**abstract property area: astropy.units.quantity.Quantity**

**Return type** Quantity

**property astString**
Return the AST serialization of this object.

**ast_description**()
A string description of this object, customized for each subclass of `ASTObject`.

**boundaryPointMesh**(*npoints=None*)
Returns an array of evenly distributed points that cover the boundary of the region. For example, if the region is a box, it will generate a list of points that trace the edges of the box.

The default value of 'npoints' is 200 for 2D regions and 2000 for three or more dimensions.

**Parameters** **npoints** (`Optional[int]`) – the approximate number of points to generate in the mesh

**Return type** `ndarray`

**Returns** list of points in degrees

**boundingBox**()
Returns an ASTBox region that bounds this region where the box sides align with RA, dec.

**Return type** *ASTBox*

**boundingCircle**()
Returns the smallest circle (*`ASTCircle`*) that bounds this region.

It is up to the caller to know that this is a 2D region (only minimal checks are made). :raises cornish.exc.NotA2DRegion: raised when attempting to get a bounding circle for a region that is not 2D

**Return type** *ASTCircle*

**property bounds: Tuple**
Returns lower and upper coordinate points that bound this region.

**Return type** `Tuple`

**containsPoint**(*point=None*)
Returns `True` if the provided point lies inside this region, `False` otherwise.

This method is a direct synonym for *`pointInRegion()`*. The name "containsPoint" is more appropriate from an object perspective, but the `pointInRegion` method is present for consistency with the AST library.

**Parameters** **point** (`Union[Iterable, SkyCoord, None]`) – a coordinate point in the same frame as this region

**Return type** `bool`

**distance**(*point1*, *point2*)
Distance between two points in this frame.

**Parameters**

- **point1** (`Union[Iterable, SkyCoord]`) – a two element list/tuple/Numpy array or a SkyCoord of the first point coordinates
- **point2** (`Union[Iterable, SkyCoord]`) – a two element list/tuple/Numpy array or a SkyCoord of the second point coordinates

**Return type** `Quantity`

**property domain: str**
The physical domain of the coordinate system (string value). The Domain attribute also controls how Frames align with each other. If the Domain value in a Frame is set, then only Frames with the same Domain value can be aligned with it.

Example values: `GRID`, `FRACTION`, `PIXEL`, `AXIS`, `SKY`, `SPECTRUM`, `CURPIC`, `NDC`, `BASEPIC`, `GRAPHICS`

Frames created by the user (for instance, using WCSADD) can have any Domain value, but the standard domain names should be avoided unless the standard meanings are appropriate for the Frame being created.

**Return type** `str`

**property fillFactor**
<Fraction of the Region which is of interest>

**frame()**
Returns a copy of the frame encapsulated by this region.

Note that the frame is not directly accessible; both this method and the underlying `starlink-pyast` function returns a copy.

**Return type** *ASTFrame*

**static frameFromAstObject**(*ast_object=None*)
Factory method that returns the appropriate Cornish frame object (e.g. `ASTSkyFrame`) for a given frame.

**Parameters** **ast_object** (`Optional[Frame]`) – an `Ast.Frame` object

**static frameFromFITSHeader**(*header*)
Factory method that returns a new ASTFrame from the provided FITS header.

**frameSet()**
Returns a copy of the frameset encapsulated by this region.

From AST docs:

```
The base Frame is the Frame in which the box was originally
defined, and the current Frame is the Frame into which the
Region is currently mapped (i.e. it will be the same as the
Frame returned by astGetRegionFrame).
```

**Return type** *ASTFrameSet*

**framesetWithMappingTo**(*template_frame=None*)
Search this frame (or set) to identify a frame that shares the same coordinate system as the provided template frame.

For example, this method can be used to see if this frame (or frame set) contains a sky frame.

Returns `None` if no mapping can be found.

**Parameters** **template_frame** (`Optional[`*ASTFrame*`]`) – an instance of the type of frame being searched for

**Return type** `Optional[`*ASTFrame*`]`

**Returns** a frame that matches the template

**classmethod fromFITSHeader**(*fits_header=None*, *uncertainty=4.848e-06*)
Factory method to create a region from the provided FITS header; the returned object will be as specific as possible (but probably an *ASTPolygon*).

The frame is determined from the FITS header.

**Parameters**

- **fits_header** – a FITS header (Astropy, fitsio, an array of cards)
- **uncertainty** (`float`) – defaults to 4.848e-6, an uncertainty of 1 arcsec

`fullyEncloses`(*region*)
: Returns 'True' if this region fully encloses the provided region.

  **Return type** `bool`

**`property id: str`**
: String which may be used to identify this object.

  NOTE: Unlike most other attributes, the value of the ID attribute is not transferred when an Object is copied. Instead, its value is undefined (and therefore defaults to an empty string) in any copy. However, it is retained in any external representation of an Object produced by the astWrite function.

  Not sure how to handle the above in this class.

  **Return type** `str`

  **Returns** string identifier that can be used to uniquely identify this object

`interiorPointMesh`(*npoints=None*)
: Returns an array of evenly distributed points that cover the surface of the region. For example, if the region is a box, it will generate a list of points that fill the interior area of the box.

  The default value of 'npoints' is 200 for 2D regions and 2000 for three or more dimensions.

  **Parameters** **`npoints`** (`Optional[int]`) – the approximate number of points to generate in the mesh

  **Returns** array of points in degrees

`inverseMapping`()
: Returns a new mapping object that is the inverse of this mapping.

  For example, if the forward transformation of this mapping is pixel to sky, then the forward transformation of the returned mapping will be sky to pixel.

**`property isAdaptive`**
: Boolean attribute that indicates whether the area adapt to changes in the coordinate system.

**`property isBounded: bool`**
: Boolean attribute that indicates whether the region is bounded.

  **Return type** `bool`

**`property isClosed: bool`**
: Boolean attribute that indicates whether the boundary be considered to be inside the region.

  **Return type** `bool`

`isFullyWithin`(*region*)
: Returns 'True' if this region is fully within the provided region.

  **Return type** `bool`

`isIdenticalTo`(*region*)
: Returns 'True' if this region is identical (to within their uncertainties) to the provided region, 'False' otherwise.

  **Return type** `bool`

**`property isLinear: bool`**
: Returns True if the mapping is linear

  **Return type** `bool`

**`property isNegated`**
: Boolean attribute that indicates whether the original region has been negated.

`isNegationOf`(*region*)
: Returns 'True' if this region is the exact negation of the provided region.

`property isSimple: bool`
: Returns True if the mapping has been simplified.

  **Return type** `bool`

`property isSkyFrame: bool`
: Returns `True` if this is a SkyFrame, `False` otherwise.

  **Return type** `bool`

`label`(*axis=None*)
: Return the label for the specified axis.

  **Return type** `str`

`mapRegionMesh`(*mapping=None*, *frame=None*)
: Returns a new ASTRegion that is the same as this one but with the specified coordinate system.

  **Parameters**

  - **mapping** (*~cornish.mapping.ASTMapping* class) – The mapping to transform positions from the current ASTRegion to those specified by the given frame.
  - **frame** (*~cornish.frame.ASTFrame* class) – Coordinate system frame to convert the current ASTRegion to.

  **Returns** **region** – A new region object covering the same area but in the frame specified in *frame*.

  **Return type** *ASTRegion*

  **Raises** `Exception` – An exception is raised for missing parameters.

`maskOnto`(*image=None*, *mapping=None*, *fits_coordinates=True*, *lower_bounds=None*, *mask_inside=True*, *mask_value=nan*)
: Apply this region as a mask on top of the provided image; note: the image values are overwritten!

  **Parameters**

  - **image** – numpy.ndarray of pixel values (or other array of values)
  - **mapping** – mapping from this region to the pixel coordinates of the provided image
  - **fits_coordinates** (`bool`) – use the pixel coordinates of a FITS file (i.e. origin = [0.5, 0.5] for 2D)
  - **lower_bounds** – lower bounds of provided image, only specify if not using FITS coordinates
  - **mask_inside** – True: mask the inside of this region; False: mask outside of this region
  - **mask_value** – the value to set the masked image pixels to

  **Returns** number of pixels in image masked

`property meshSize: int`
: Number of points used to create a mesh covering the region.

  **Return type** `int`

`property naxes: int`
: Returns the number of axes for the frame.

  **Return type** `int`

`negate()`
Negate the region, i.e. points inside the region will be outside, and vice versa.

`property numberOfInputCoordinates`
Number of dimensions of the space in which the Mapping's input points reside. This property gives the number of coordinate values required to specify an input point for a Mapping.

**Returns** number of input dimensions described by this mapper

`property numberOfOutputCoordinates`
Number of dimensions of the space in which the Mapping's output points reside. This property gives the number of coordinate values required to specify an output point for a Mapping.

**Returns** number of output dimensions described by this mapper

`offsetAlongGeodesicCurve`(*point1*, *point2*, *offset*)
Coordinates and offset value should be in the units of the frame, e.g. pixels, degrees.

In a sky frame, the line will be curved. In a basic frame, the line will be straight.

**Parameters**

- `point1` (`Iterable`) – a two element list/tuple/NumPy array of the first point coordinates
- `point2` (`Iterable`) – a two element list/tuple/NumPy array of the second point coordinates
- `offset` (`Quantity`) – a distance along the geodesic sphere connecting the two points

`overlaps`(*region*)
Return `True` if this region overlaps with the provided one.

**Return type** `bool`

`pointInRegion`(*point*)
Returns `True` if the provided point lies inside this region, `False` otherwise.

If no units are specified degrees are assumed.

**Parameters** `point` (`Union[Iterable, SkyCoord, ndarray]`) – a coordinate point in the same frame as this region

**Return type** `bool`

`property points: numpy.ndarray`
The array of points that define the region. The interpretation of the points is dependent on the type of shape in question.

Box: returns two points; the center and a box corner. Circle: returns two points; the center and a point on the circumference. CmpRegion: no points returned; to get points that define a compound region, call this method on each of the component regions via the method "decompose". Ellipse: three points: 1) center, 2) end of one axis, 3) end of the other axis Interval: two points: 1) lower bounds position, 2) upper bounds position (reversed when interval is an excluded interval) NullRegion: no points PointList: positions that the list was constructed with Polygon: vertex positions used to construct the polygon Prism: no points (see CmpRegion)

NOTE: points returned reflect the current coordinate system and may be different from the initial construction

**Return type** `ndarray`

**Returns** NumPy array of coordinate points in degrees, shape (ncoord,2), e.g. [[ra1,dec1], [ra2, dec2], . . . , [ra_n, dec_n]]

**regionWithMapping**(*map=None*, *frame=None*)

Returns a new ASTRegion with the coordinate system from the supplied frame.

Corresponds to the `astMapRegion` C function (`starlink.Ast.mapregion`).

**Parameters**

- **map** – A mapping that can convert coordinates from the system of the current region to that of the supplied frame.
- **frame** – A frame containing the coordinate system for the new region.

**Return type** *ASTRegion*

**Returns** new ASTRegion with a new coordinate system

**setLabelForAxis**(*label=None*)

Set the label for the specified axis.

**setUnitForAxis**(*axis=None*, *unit=None*)

Set the unit as a string value for the specified axis.

**property system**

String which identifies the coordinate system represented by the Frame. The system is `Cartesian` by default, but can have other values for subclasses of Frame, e.g. `FK4`, `Galactic`.

**property title: str**

Returns the frame title, a string describing the coordinate system which the frame represents.

**Return type** `str`

**abstract toPolygon**(*npoints=200*, *maxerr=<Quantity 1. arcsec>*)

Method that guarantees returning a polygon that describes or approximates this region.

This method provides a common interface to create polygons from different region types. Calling this on an ASTPolygon will return itself; calling it on an ASTCircle will return a polygon that approximates the circle. The parameters 'npoints' and 'maxerr' will be used only when appropriate.

**Parameters**

- **npoints** – number of points to sample from the region's boundary for the resulting polygon
- **maxerr** (`astropy.units.Quantity`) –

**Return type** *ASTPolygon*

**transform**(*points=None*)

Transform the coordinates of a set of points provided according the mapping defined by this object.

**Parameters**

- **in** – input list of coordinates as numpy.ndarray, any iterable list accepted 2-dimensional array with shape (nin,npoint)
- **out** – output coordinates

Format of points:

```
[ [ values on axis 1 ], [ values on axis 2 ], ... ]
```

e.g. sky to pixel:

```
[ [ra1, ra2, ...], [dec1, dec2, ...] ]
```

**Return type** `ndarray`

**property uncertainty**

Uncertainties associated with the boundary of the Box.

The uncertainty in any point on the boundary of the Box is found by shifting the supplied "uncertainty" Region so that it is centered at the boundary point being considered. The area covered by the shifted uncertainty Region then represents the uncertainty in the boundary position. The uncertainty is assumed to be the same for all points.

**unit**(*axis=None*)

Return the unit for the specified axis.

### 2.1.2 ASTBox

**class** cornish.**ASTBox**(*ast_object=None*)

Bases: `cornish.region.region.ASTRegion`

ASTBox is an ASTRegion that represents a box with sides parallel to the axes of an ASTFrame.

Accepted signatures for creating an ASTBox:

```
b = ASTBox(frame, cornerPoint, cornerPoint2)
b = ASTBox(frame, cornerPoint, centerPoint)
b = ASTBox(frame, dimensions)
b = ASTBox(ast_box) (where ast_box is an Ast.Box object)
```

Points and dimensions can be any two element container, e.g.

```
(1,2)
[1,2]
np.array([1,2])
```

If `dimensions` is specified, a box enclosing the entire area will be defined.

The 'frame' parameter may either be an ASTFrame object or a `starlink.Ast.frame` object.

A Box is similar to an Interval, the only real difference being that the Interval class allows some axis limits to be unspecified. Note, a Box will only look like a box if the Frame geometry is approximately flat. For instance, a Box centered close to a pole in a SkyFrame will look more like a fan than a box (the Polygon class can be used to create a box-like region close to a pole).

**Parameters**

- **ast_box** – an existing object of type `starlink.Ast.Box`
- **frame** – a frame the box is to be defined in, uses `ASTICRSFrame` if *None*
- **cornerPoint** –
- **cornerPoint2** –
- **centerPoint** –
- **dimensions** – dimensions of the box in pixels for use on a Cartesian frame (AST frame='Cartesian' and system='GRID')

**angle**(*vertex=None, points=None*)

Calculate the angle in this frame between two line segments connected by a point.

Let A = point1, C = point2, and B = the vertex point. This method calculates the angle between the line segments AB and CB.

If the frame is a sky frame, lines are drawn on great circles. Units are assumed to be degrees if not provided with units, e.g. as an astropy.coordinates.SkyCoord or astropy.units.Quantity values.

**Parameters**

- **vertex** (`Optional[Iterable]`) – a two element list/tuple/Numpy array or a SkyCoord of the vertex
- **points** (`Optional[Container[Union[SkyCoord, Iterable]]]`) – a two element list/tuple/etc. containing two points in this frame

**Return type** `Quantity`

**`property area: astropy.units.quantity.Quantity`**

The area of the box within its frame (e.g. on a Cartesian plane or sphere). [Not yet implemented.]

**Return type** `Quantity`

**`property astString`**

Return the AST serialization of this object.

**`ast_description()`**

A string description of this object, customized for each subclass of `ASTObject`.

**`boundaryPointMesh`**(*npoints=None*)

Returns an array of evenly distributed points that cover the boundary of the region. For example, if the region is a box, it will generate a list of points that trace the edges of the box.

The default value of 'npoints' is 200 for 2D regions and 2000 for three or more dimensions.

**Parameters** **npoints** (`Optional[int]`) – the approximate number of points to generate in the mesh

**Return type** `ndarray`

**Returns** list of points in degrees

**`boundingBox()`**

Returns an ASTBox region that bounds this region where the box sides align with RA, dec.

**Return type** *ASTBox*

**`boundingCircle()`**

Returns the smallest circle (*`ASTCircle`*) that bounds this region.

It is up to the caller to know that this is a 2D region (only minimal checks are made). :raises cornish.exc.NotA2DRegion: raised when attempting to get a bounding circle for a region that is not 2D

**Return type** *ASTCircle*

**`property bounds: Tuple`**

Returns lower and upper coordinate points that bound this region.

**Return type** `Tuple`

**`property center: numpy.ndarray`**

Returns "self.centre". This is a British library, after all.

**Return type** `ndarray`

**`property centre: numpy.ndarray`**

Returns the location of the Box's center as a coordinate pair, in degrees if a sky frame.

**Return type** `ndarray`

**Returns** a Numpy array of points (axis1, axis2) `containsPoint`(*point=None*)

Returns `True` if the provided point lies inside this region, `False` otherwise.

This method is a direct synonym for *`pointInRegion()`*. The name "containsPoint" is more appropriate from an object perspective, but the `pointInRegion` method is present for consistency with the AST library.

**Parameters** **`point`** (`Union`[`Iterable`, `SkyCoord`, `None`]) – a coordinate point in the same frame as this region

**Return type** `bool`

**`property corner: numpy.ndarray`**

Returns the location of one of the box's corners as a coordinate pair, in degrees if a sky frame.

**Return type** `ndarray`

**Returns** a Numpy array of points (axis1, axis2)

`corners`(*mapping=None*)

Returns a list of all four corners of box.

**Parameters** **`mapping`** (*ASTMapping*) – A mapping object.

**Returns** A list of points in degrees, e.g. `[(p1,p2), (p3, p4), (p5, p6), (p7, p8)]`

**Return type** list

`distance`(*point1*, *point2*)

Distance between two points in this frame.

**Parameters**

- **`point1`** (`Union`[`Iterable`, `SkyCoord`]) – a two element list/tuple/Numpy array or a SkyCoord of the first point coordinates
- **`point2`** (`Union`[`Iterable`, `SkyCoord`]) – a two element list/tuple/Numpy array or a SkyCoord of the second point coordinates

**Return type** `Quantity`

**`property domain: str`**

The physical domain of the coordinate system (string value). The Domain attribute also controls how Frames align with each other. If the Domain value in a Frame is set, then only Frames with the same Domain value can be aligned with it.

Example values: `GRID`, `FRACTION`, `PIXEL`, `AXIS`, `SKY`, `SPECTRUM`, `CURPIC`, `NDC`, `BASEPIC`, `GRAPHICS`

Frames created by the user (for instance, using WCSADD) can have any Domain value, but the standard domain names should be avoided unless the standard meanings are appropriate for the Frame being created.

**Return type** `str`

**`property fillFactor`**

<Fraction of the Region which is of interest>

`frame`()

Returns a copy of the frame encapsulated by this region.

Note that the frame is not directly accessible; both this method and the underlying `starlink-pyast` function returns a copy.

**Return type** *`ASTFrame`*

`static frameFromAstObject`(*ast_object=None*)

Factory method that returns the appropriate Cornish frame object (e.g. `ASTSkyFrame`) for a given frame.

**Parameters** **ast_object** (Optional[Frame]) – an Ast.Frame object

static **frameFromFITSHeader**(*header*)
Factory method that returns a new ASTFrame from the provided FITS header.

**frameSet**()
Returns a copy of the frameset encapsulated by this region.

From AST docs:

```
The base Frame is the Frame in which the box was originally
defined, and the current Frame is the Frame into which the
Region is currently mapped (i.e. it will be the same as the
Frame returned by astGetRegionFrame).
```

**Return type** *ASTFrameSet*

**framesetWithMappingTo**(*template_frame=None*)
Search this frame (or set) to identify a frame that shares the same coordinate system as the provided template frame.

For example, this method can be used to see if this frame (or frame set) contains a sky frame.

Returns None if no mapping can be found.

**Parameters** **template_frame** (Optional[*ASTFrame*]) – an instance of the type of frame being searched for

**Return type** Optional[*ASTFrame*]

**Returns** a frame that matches the template

classmethod **fromCentreAndCorner**(*frame*, *centre=None*, *corner=None*, *center=None*, *uncertainty=None*)
Create a new ASTBox object defined by the provided corner and centre points.

**Parameters**

- **frame** (Union[Frame, *ASTFrame*]) – the frame the provided points lie in, accepts either ASTFrame or starlink.Ast.Frame objects
- **centre** (Optional[Iterable]) – the coordinate of the point at the centre of the box in the frame provided
- **corner** (Optional[Iterable]) – the coordinate of the point at any corner of the box in the frame provided
- **center** (Optional[Iterable]) – synonym for ‘centre’, ignored if ‘centre’ is defined
- **uncertainty** (Union[*ASTRegion*, Region, None]) –

**Return type** *ASTBox*

classmethod **fromCorners**(*frame*, *corners=None*, *uncertainty=None*)
Create a new ASTBox object defined by two corner points.

**Parameters**

- **frame** (Union[Frame, *ASTFrame*]) – the frame the provided points lie in, accepts either ASTFrame or starlink.Ast.Frame objects
- **corners** (Optional[Iterable[Iterable]]) – a collection (list, tuple, array, etc.) of coordinates of two corners of the box in the frame provided

- **uncertainty** (`Union[ASTRegion, Region, None]`) –

**Return type** `ASTBox`

**classmethod fromFITSHeader**(*fits_header=None*, *uncertainty=4.848e-06*)

Factory method to create a region from the provided FITS header; the returned object will be as specific as possible (but probably an `ASTPolygon`).

The frame is determined from the FITS header.

**Parameters**

- **fits_header** – a FITS header (Astropy, fitsio, an array of cards)
- **uncertainty** (`float`) – defaults to 4.848e-6, an uncertainty of 1 arcsec

**fullyEncloses**(*region*)

Returns 'True' if this region fully encloses the provided region.

**Return type** `bool`

**property id: str**

String which may be used to identify this object.

NOTE: Unlike most other attributes, the value of the ID attribute is not transferred when an Object is copied. Instead, its value is undefined (and therefore defaults to an empty string) in any copy. However, it is retained in any external representation of an Object produced by the astWrite function.

Not sure how to handle the above in this class.

**Return type** `str`

**Returns** string identifier that can be used to uniquely identify this object

**interiorPointMesh**(*npoints=None*)

Returns an array of evenly distributed points that cover the surface of the region. For example, if the region is a box, it will generate a list of points that fill the interior area of the box.

The default value of 'npoints' is 200 for 2D regions and 2000 for three or more dimensions.

**Parameters** **npoints** (`Optional[int]`) – the approximate number of points to generate in the mesh

**Returns** array of points in degrees

**inverseMapping**()

Returns a new mapping object that is the inverse of this mapping.

For example, if the forward transformation of this mapping is pixel to sky, then the forward transformation of the returned mapping will be sky to pixel.

**property isAdaptive**

Boolean attribute that indicates whether the area adapt to changes in the coordinate system.

**property isBounded: bool**

Boolean attribute that indicates whether the region is bounded.

**Return type** `bool`

**property isClosed: bool**

Boolean attribute that indicates whether the boundary be considered to be inside the region.

**Return type** `bool`

**isFullyWithin**(*region*)

Returns 'True' if this region is fully within the provided region.

**Return type** `bool`

`isIdenticalTo`(*region*)
Returns 'True' if this region is identical (to within their uncertainties) to the provided region, 'False' otherwise.

**Return type** `bool`

`property isLinear: bool`
Returns True if the mapping is linear

**Return type** `bool`

`property isNegated`
Boolean attribute that indicates whether the original region has been negated.

`isNegationOf`(*region*)
Returns 'True' if this region is the exact negation of the provided region.

`property isSimple: bool`
Returns True if the mapping has been simplified.

**Return type** `bool`

`property isSkyFrame: bool`
Returns `True` if this is a SkyFrame, `False` otherwise.

**Return type** `bool`

`label`(*axis=None*)
Return the label for the specified axis.

**Return type** `str`

`mapRegionMesh`(*mapping=None*, *frame=None*)
Returns a new ASTRegion that is the same as this one but with the specified coordinate system.

**Parameters**

- **`mapping`** (*~cornish.mapping.ASTMapping* class) – The mapping to transform positions from the current ASTRegion to those specified by the given frame.
- **`frame`** (*~cornish.frame.ASTFrame* class) – Coordinate system frame to convert the current ASTRegion to.

**Returns** **region** – A new region object covering the same area but in the frame specified in *frame*.

**Return type** *ASTRegion*

**Raises** **`Exception`** – An exception is raised for missing parameters.

`maskOnto`(*image=None*, *mapping=None*, *fits_coordinates=True*, *lower_bounds=None*, *mask_inside=True*, *mask_value=nan*)
Apply this region as a mask on top of the provided image; note: the image values are overwritten!

**Parameters**

- **`image`** – numpy.ndarray of pixel values (or other array of values)
- **`mapping`** – mapping from this region to the pixel coordinates of the provided image
- **`fits_coordinates`** (`bool`) – use the pixel coordinates of a FITS file (i.e. origin = [0.5, 0.5] for 2D)
- **`lower_bounds`** – lower bounds of provided image, only specify if not using FITS coordinates

- **mask_inside** – True: mask the inside of this region; False: mask outside of this region
- **mask_value** – the value to set the masked image pixels to

**Returns** number of pixels in image masked

**property meshSize: int**

Number of points used to create a mesh covering the region.

**Return type** `int`

**property naxes: int**

Returns the number of axes for the frame.

**Return type** `int`

**negate()**

Negate the region, i.e. points inside the region will be outside, and vice versa.

**property numberOfInputCoordinates**

Number of dimensions of the space in which the Mapping's input points reside. This property gives the number of coordinate values required to specify an input point for a Mapping.

**Returns** number of input dimensions described by this mapper

**property numberOfOutputCoordinates**

Number of dimensions of the space in which the Mapping's output points reside. This property gives the number of coordinate values required to specify an output point for a Mapping.

**Returns** number of output dimensions described by this mapper

**offsetAlongGeodesicCurve**(*point1*, *point2*, *offset*)

Coordinates and offset value should be in the units of the frame, e.g. pixels, degrees.

In a sky frame, the line will be curved. In a basic frame, the line will be straight.

**Parameters**

- **point1** (`Iterable`) – a two element list/tuple/NumPy array of the first point coordinates
- **point2** (`Iterable`) – a two element list/tuple/NumPy array of the second point coordinates
- **offset** (`Quantity`) – a distance along the geodesic sphere connecting the two points

**overlaps**(*region*)

Return `True` if this region overlaps with the provided one.

**Return type** `bool`

**pointInRegion**(*point*)

Returns `True` if the provided point lies inside this region, `False` otherwise.

If no units are specified degrees are assumed.

**Parameters** **point** (`Union[Iterable, SkyCoord, ndarray]`) – a coordinate point in the same frame as this region

**Return type** `bool`

**property points: numpy.ndarray**

The array of points that define the region. The interpretation of the points is dependent on the type of shape in question.

Box: returns two points; the center and a box corner. Circle: returns two points; the center and a point on the circumference. CmpRegion: no points returned; to get points that define a compound region, call this method on each of the component regions via the method "decompose". Ellipse: three points: 1) center, 2) end of one axis, 3) end of the other axis Interval: two points: 1) lower bounds position, 2) upper bounds position (reversed when interval is an excluded interval) NullRegion: no points PointList: positions that the list was constructed with Polygon: vertex positions used to construct the polygon Prism: no points (see CmpRegion)

NOTE: points returned reflect the current coordinate system and may be different from the initial construction

**Return type** `ndarray`

**Returns** NumPy array of coordinate points in degrees, shape (ncoord,2), e.g. [[ra1,dec1], [ra2, dec2], . . . , [ra_n, dec_n]]

**`regionWithMapping`**(*map=None*, *frame=None*)

Returns a new ASTRegion with the coordinate system from the supplied frame.

Corresponds to the `astMapRegion` C function (`starlink.Ast.mapregion`).

**Parameters**

- **`map`** – A mapping that can convert coordinates from the system of the current region to that of the supplied frame.
- **`frame`** – A frame containing the coordinate system for the new region.

**Return type** *`ASTRegion`*

**Returns** new ASTRegion with a new coordinate system

**`setLabelForAxis`**(*label=None*)

Set the label for the specified axis.

**`setUnitForAxis`**(*axis=None*, *unit=None*)

Set the unit as a string value for the specified axis.

**`property system`**

String which identifies the coordinate system represented by the Frame. The system is `Cartesian` by default, but can have other values for subclasses of Frame, e.g. `FK4`, `Galactic`.

**`property title: str`**

Returns the frame title, a string describing the coordinate system which the frame represents.

**Return type** `str`

**`toPolygon`**(*npoints=200*, *maxerr=<Quantity 1. arcsec>*)

Returns a four-vertex ASTPolygon that describes this box in the same frame.

The parameters 'npoints' and 'maxerr' are ignored.

**Return type** *ASTPolygon*

**`transform`**(*points=None*)

Transform the coordinates of a set of points provided according the mapping defined by this object.

**Parameters**

- **`in`** – input list of coordinates as numpy.ndarray, any iterable list accepted 2-dimensional array with shape (nin,npoint)
- **`out`** – output coordinates

Format of points:

```
[ [ values on axis 1 ], [ values on axis 2 ], ... ]
```

e.g. sky to pixel:

```
[ [ra1, ra2, ...], [dec1, dec2, ...] ]
```

**Return type** `ndarray`

**property uncertainty**

Uncertainties associated with the boundary of the Box.

The uncertainty in any point on the boundary of the Box is found by shifting the supplied "uncertainty" Region so that it is centered at the boundary point being considered. The area covered by the shifted uncertainty Region then represents the uncertainty in the boundary position. The uncertainty is assumed to be the same for all points.

**unit**(*axis=None*)

Return the unit for the specified axis.

## 2.1.3 ASTCircle

**class** cornish.**ASTCircle**(*ast_object=None*, *frame=None*, *center=None*, *edge_point=None*, *radius=None*)

Bases: *cornish.region.region.ASTRegion*

ASTCircle is an *ASTRegion* that represents a circle.

Accepted signatures for creating an ASTCircle

**Parameters**

- **ast_object** – a circle object from the `starlink-pyast` module
- **frame** – a frame the circle is to be defined in, uses `ASTICRSFrame` if *None*
- **center** – two elements that describe the center point of the circle in the provided frame in degrees
- **edge_point** – two elements that describe a point on the circumference of the circle in the provided frame in degrees
- **radius** – radius of the circle in degrees

**angle**(*vertex=None*, *points=None*)

Calculate the angle in this frame between two line segments connected by a point.

Let A = point1, C = point2, and B = the vertex point. This method calculates the angle between the line segments AB and CB.

If the frame is a sky frame, lines are drawn on great circles. Units are assumed to be degrees if not provided with units, e.g. as an astropy.coordinates.SkyCoord or astropy.units.Quantity values.

**Parameters**

- **vertex** (`Optional[Iterable]`) – a two element list/tuple/Numpy array or a SkyCoord of the vertex
- **points** (`Optional[Container[Union[SkyCoord, Iterable]]]`) – a two element list/tuple/etc. containing two points in this frame

**Return type** `Quantity`

**property area**
The area of the circle within its frame (e.g. on a Cartesian plane or sphere). [Not yet implemented.]

**property astString**
Return the AST serialization of this object.

**ast_description()**
A string description of this object, customized for each subclass of `ASTObject`.

**boundaryPointMesh**(*npoints=None*)
Returns an array of evenly distributed points that cover the boundary of the region. For example, if the region is a box, it will generate a list of points that trace the edges of the box.

The default value of 'npoints' is 200 for 2D regions and 2000 for three or more dimensions.

**Parameters** **npoints** (`Optional[int]`) – the approximate number of points to generate in the mesh

**Return type** `ndarray`

**Returns** list of points in degrees

**boundingBox()**
Returns an ASTBox region that bounds this region where the box sides align with RA, dec.

**Return type** *ASTBox*

**boundingCircle()**
This method returns itself; a circle region is its own bounding circle.

**Return type** *ASTCircle*

**property bounds: Tuple**
Returns lower and upper coordinate points that bound this region.

**Return type** `Tuple`

**property center**
The center of this circle region in degrees (a synonym for `self.centre()`" for the Americans).

**Returns**

**Return type** returns: a list of points [x,y] that describe the centre of the circle in degrees

**property centre**
The centre of this circle region in degrees.

**Returns**

**Return type** returns: a list of points [x,y] that describe the centre of the circle in degrees

**containsPoint**(*point=None*)
Returns `True` if the provided point lies inside this region, `False` otherwise.

This method is a direct synonym for *pointInRegion()*. The name "containsPoint" is more appropriate from an object perspective, but the `pointInRegion` method is present for consistency with the AST library.

**Parameters** **point** (`Union[Iterable, SkyCoord, None]`) – a coordinate point in the same frame as this region

**Return type** `bool`

**distance**(*point1*, *point2*)
Distance between two points in this frame.

**Parameters**

- **point1** (Union[Iterable, SkyCoord]) – a two element list/tuple/Numpy array or a SkyCoord of the first point coordinates
- **point2** (Union[Iterable, SkyCoord]) – a two element list/tuple/Numpy array or a SkyCoord of the second point coordinates

**Return type** Quantity

**property domain: str**

The physical domain of the coordinate system (string value). The Domain attribute also controls how Frames align with each other. If the Domain value in a Frame is set, then only Frames with the same Domain value can be aligned with it.

Example values: GRID, FRACTION, PIXEL, AXIS, SKY, SPECTRUM, CURPIC, NDC, BASEPIC, GRAPHICS

Frames created by the user (for instance, using WCSADD) can have any Domain value, but the standard domain names should be avoided unless the standard meanings are appropriate for the Frame being created.

**Return type** str

**property fillFactor**

<Fraction of the Region which is of interest>

**frame()**

Returns a copy of the frame encapsulated by this region.

Note that the frame is not directly accessible; both this method and the underlying starlink-pyast function returns a copy.

**Return type** *ASTFrame*

**static frameFromAstObject**(*ast_object=None*)

Factory method that returns the appropriate Cornish frame object (e.g. ASTSkyFrame) for a given frame.

**Parameters ast_object** (Optional[Frame]) – an Ast.Frame object

**static frameFromFITSHeader**(*header*)

Factory method that returns a new ASTFrame from the provided FITS header.

**frameSet()**

Returns a copy of the frameset encapsulated by this region.

From AST docs:

```
The base Frame is the Frame in which the box was originally
defined, and the current Frame is the Frame into which the
Region is currently mapped (i.e. it will be the same as the
Frame returned by astGetRegionFrame).
```

**Return type** *ASTFrameSet*

**framesetWithMappingTo**(*template_frame=None*)

Search this frame (or set) to identify a frame that shares the same coordinate system as the provided template frame.

For example, this method can be used to see if this frame (or frame set) contains a sky frame.

Returns None if no mapping can be found.

**Parameters template_frame** (Optional[*ASTFrame*]) – an instance of the type of frame being searched for

**Return type** Optional[*ASTFrame*]

**Returns** a frame that matches the template

`classmethod fromFITSHeader`(*fits_header=None*, *uncertainty=4.848e-06*)

Factory method to create a region from the provided FITS header; the returned object will be as specific as possible (but probably an `ASTPolygon`).

The frame is determined from the FITS header.

**Parameters**

- **`fits_header`** – a FITS header (Astropy, fitsio, an array of cards)
- **`uncertainty`** (`float`) – defaults to 4.848e-6, an uncertainty of 1 arcsec

`fullyEncloses`(*region*)

Returns 'True' if this region fully encloses the provided region.

**Return type** `bool`

`property id: str`

String which may be used to identify this object.

NOTE: Unlike most other attributes, the value of the ID attribute is not transferred when an Object is copied. Instead, its value is undefined (and therefore defaults to an empty string) in any copy. However, it is retained in any external representation of an Object produced by the astWrite function.

Not sure how to handle the above in this class.

**Return type** `str`

**Returns** string identifier that can be used to uniquely identify this object

`interiorPointMesh`(*npoints=None*)

Returns an array of evenly distributed points that cover the surface of the region. For example, if the region is a box, it will generate a list of points that fill the interior area of the box.

The default value of 'npoints' is 200 for 2D regions and 2000 for three or more dimensions.

**Parameters** **`npoints`** (`Optional[int]`) – the approximate number of points to generate in the mesh

**Returns** array of points in degrees

`inverseMapping`()

Returns a new mapping object that is the inverse of this mapping.

For example, if the forward transformation of this mapping is pixel to sky, then the forward transformation of the returned mapping will be sky to pixel.

`property isAdaptive`

Boolean attribute that indicates whether the area adapt to changes in the coordinate system.

`property isBounded: bool`

Boolean attribute that indicates whether the region is bounded.

**Return type** `bool`

`property isClosed: bool`

Boolean attribute that indicates whether the boundary be considered to be inside the region.

**Return type** `bool`

`isFullyWithin`(*region*)

Returns 'True' if this region is fully within the provided region.

**Return type** `bool`

`isIdenticalTo`(*region*)
Returns 'True' if this region is identical (to within their uncertainties) to the provided region, 'False' otherwise.

**Return type** `bool`

**property isLinear: bool**
Returns True if the mapping is linear

**Return type** `bool`

**property isNegated**
Boolean attribute that indicates whether the original region has been negated.

`isNegationOf`(*region*)
Returns 'True' if this region is the exact negation of the provided region.

**property isSimple: bool**
Returns True if the mapping has been simplified.

**Return type** `bool`

**property isSkyFrame: bool**
Returns `True` if this is a SkyFrame, `False` otherwise.

**Return type** `bool`

`label`(*axis=None*)
Return the label for the specified axis.

**Return type** `str`

`mapRegionMesh`(*mapping=None*, *frame=None*)
Returns a new ASTRegion that is the same as this one but with the specified coordinate system.

**Parameters**

- **mapping** (*~cornish.mapping.ASTMapping* class) – The mapping to transform positions from the current ASTRegion to those specified by the given frame.
- **frame** (*~cornish.frame.ASTFrame* class) – Coordinate system frame to convert the current ASTRegion to.

**Returns** **region** – A new region object covering the same area but in the frame specified in *frame*.

**Return type** *ASTRegion*

**Raises** `Exception` – An exception is raised for missing parameters.

`maskOnto`(*image=None*, *mapping=None*, *fits_coordinates=True*, *lower_bounds=None*, *mask_inside=True*, *mask_value=nan*)
Apply this region as a mask on top of the provided image; note: the image values are overwritten!

**Parameters**

- **image** – numpy.ndarray of pixel values (or other array of values)
- **mapping** – mapping from this region to the pixel coordinates of the provided image
- **fits_coordinates** (`bool`) – use the pixel coordinates of a FITS file (i.e. origin = [0.5, 0.5] for 2D)
- **lower_bounds** – lower bounds of provided image, only specify if not using FITS coordinates
- **mask_inside** – True: mask the inside of this region; False: mask outside of this region

- `mask_value` – the value to set the masked image pixels to

**Returns** number of pixels in image masked

**property meshSize: int**
Number of points used to create a mesh covering the region.

**Return type** `int`

**property naxes: int**
Returns the number of axes for the frame.

**Return type** `int`

**negate()**
Negate the region, i.e. points inside the region will be outside, and vice versa.

**property numberOfInputCoordinates**
Number of dimensions of the space in which the Mapping's input points reside. This property gives the number of coordinate values required to specify an input point for a Mapping.

**Returns** number of input dimensions described by this mapper

**property numberOfOutputCoordinates**
Number of dimensions of the space in which the Mapping's output points reside. This property gives the number of coordinate values required to specify an output point for a Mapping.

**Returns** number of output dimensions described by this mapper

**offsetAlongGeodesicCurve**(*point1*, *point2*, *offset*)
Coordinates and offset value should be in the units of the frame, e.g. pixels, degrees.

In a sky frame, the line will be curved. In a basic frame, the line will be straight.

**Parameters**

- **point1** (`Iterable`) – a two element list/tuple/NumPy array of the first point coordinates
- **point2** (`Iterable`) – a two element list/tuple/NumPy array of the second point coordinates
- **offset** (`Quantity`) – a distance along the geodesic sphere connecting the two points

**overlaps**(*region*)
Return `True` if this region overlaps with the provided one.

**Return type** `bool`

**pointInRegion**(*point*)
Returns `True` if the provided point lies inside this region, `False` otherwise.

If no units are specified degrees are assumed.

**Parameters** **point** (`Union[Iterable, SkyCoord, ndarray]`) – a coordinate point in the same frame as this region

**Return type** `bool`

**property points: numpy.ndarray**
The array of points that define the region. The interpretation of the points is dependent on the type of shape in question.

Box: returns two points; the center and a box corner. Circle: returns two points; the center and a point on the circumference. CmpRegion: no points returned; to get points that define a compound region, call this method on each of the component regions via the method "decompose". Ellipse: three points: 1) center, 2) end of one axis, 3) end of the other axis Interval: two points: 1) lower bounds position, 2) upper bounds position (reversed when interval is an excluded interval) NullRegion: no points PointList: positions that the list was constructed with Polygon: vertex positions used to construct the polygon Prism: no points (see CmpRegion)

NOTE: points returned reflect the current coordinate system and may be different from the initial construction

**Return type** `ndarray`

**Returns** NumPy array of coordinate points in degrees, shape (ncoord,2), e.g. [[ra1,dec1], [ra2, dec2], …, [ra_n, dec_n]]

**property radius: astropy.units.quantity.Quantity**

The radius of this circle region in degrees.

**Return type** `Quantity`

**Returns** The radius as a geodesic distance in the associated coordinate system as an `astropy.units.Quantity` object in degrees.

**regionWithMapping**(*map=None*, *frame=None*)

Returns a new ASTRegion with the coordinate system from the supplied frame.

Corresponds to the `astMapRegion` C function (`starlink.Ast.mapregion`).

**Parameters**

- **map** – A mapping that can convert coordinates from the system of the current region to that of the supplied frame.
- **frame** – A frame containing the coordinate system for the new region.

**Return type** *ASTRegion*

**Returns** new ASTRegion with a new coordinate system

**setLabelForAxis**(*label=None*)

Set the label for the specified axis.

**setUnitForAxis**(*axis=None*, *unit=None*)

Set the unit as a string value for the specified axis.

**property system**

String which identifies the coordinate system represented by the Frame. The system is `Cartesian` by default, but can have other values for subclasses of Frame, e.g. `FK4`, `Galactic`.

**property title: str**

Returns the frame title, a string describing the coordinate system which the frame represents.

**Return type** `str`

**toPolygon**(*npoints=200*, *maxerr=<Quantity 1. arcsec>*)

Returns a new polygon region that approximates this circle in the same frame.

The algorithm used in this method leads to the new polygon being fully inscribed by the originating circle; all points generated are on the circle's circumference. Although the default number of points is 200, typically a much smaller number (e.g. 20) is sufficient.

**Parameters**

- **npoints** – number of points to sample from the circle for the resulting polygon
- **maxerr** (`Quantity`) –

**transform**(*points=None*)

Transform the coordinates of a set of points provided according the mapping defined by this object.

**Parameters**

- **in** – input list of coordinates as numpy.ndarray, any iterable list accepted 2-dimensional array with shape (nin,npoint)
- **out** – output coordinates

Format of points:

```
[ [ values on axis 1 ], [ values on axis 2 ], ... ]
```

e.g. sky to pixel:

```
[ [ra1, ra2, ...], [dec1, dec2, ...] ]
```

**Return type** `ndarray`

**property uncertainty**

Uncertainties associated with the boundary of the Box.

The uncertainty in any point on the boundary of the Box is found by shifting the supplied "uncertainty" Region so that it is centered at the boundary point being considered. The area covered by the shifted uncertainty Region then represents the uncertainty in the boundary position. The uncertainty is assumed to be the same for all points.

**unit**(*axis=None*)

Return the unit for the specified axis.

### 2.1.4 ASTCompoundRegion

class cornish.**ASTCompoundRegion**(*ast_object=None*, *regions=None*, *operation=None*)

Bases: *cornish.region.region.ASTRegion*

A region that is created as the composite of multiple regions.

Regions are composited two at a time in the order they are supplied, e.g. if regions=[r1, r2, r3, r4] the result would be

region = compound( compound( compound(r1, r2), r3 ) r4 )

all using the same operator as specified.

**Parameters**

- **regions** (`Optional[Iterable[Union[`*ASTRegion*`, Region]]]`) – a list of regions to compound
- **operation** (`Optional[int]`) – one of *starlink.Ast.AND, `starlink.Ast.OR*, *starlink.Ast.XOR*

**angle**(*vertex=None*, *points=None*)

Calculate the angle in this frame between two line segments connected by a point.

Let A = point1, C = point2, and B = the vertex point. This method calculates the angle between the line segments AB and CB.

If the frame is a sky frame, lines are drawn on great circles. Units are assumed to be degrees if not provided with units, e.g. as an astropy.coordinates.SkyCoord or astropy.units.Quantity values.

**Parameters**

- **vertex** (`Optional[Iterable]`) – a two element list/tuple/Numpy array or a SkyCoord of the vertex
- **points** (`Optional[Container[Union[SkyCoord, Iterable]]]`) – a two element list/tuple/etc. containing two points in this frame

**Return type** `Quantity`

**property area**
The area of the compound region on the sphere. [Not yet implemented.]

**property astString**
Return the AST serialization of this object.

**ast_description()**
A string description of this object, customized for each subclass of `ASTObject`.

**boundaryPointMesh**(*npoints=None*)
Returns an array of evenly distributed points that cover the boundary of the region. For example, if the region is a box, it will generate a list of points that trace the edges of the box.

The default value of 'npoints' is 200 for 2D regions and 2000 for three or more dimensions.

**Parameters** **npoints** (`Optional[int]`) – the approximate number of points to generate in the mesh

**Return type** `ndarray`

**Returns** list of points in degrees

**boundingBox()**
Returns an ASTBox region that bounds this region where the box sides align with RA, dec.

**Return type** *ASTBox*

**boundingCircle()**
Returns the smallest circle (*`ASTCircle`*) that bounds this region.

It is up to the caller to know that this is a 2D region (only minimal checks are made). :raises cornish.exc.NotA2DRegion: raised when attempting to get a bounding circle for a region that is not 2D

**Return type** *ASTCircle*

**property bounds: Tuple**
Returns lower and upper coordinate points that bound this region.

**Return type** `Tuple`

**containsPoint**(*point=None*)
Returns `True` if the provided point lies inside this region, `False` otherwise.

This method is a direct synonym for *`pointInRegion()`*. The name "containsPoint" is more appropriate from an object perspective, but the `pointInRegion` method is present for consistency with the AST library.

**Parameters** **point** (`Union[Iterable, SkyCoord, None]`) – a coordinate point in the same frame as this region

**Return type** `bool`

**distance**(*point1*, *point2*)
Distance between two points in this frame.

**Parameters**

- **point1** (`Union[Iterable, SkyCoord]`) – a two element list/tuple/Numpy array or a SkyCoord of the first point coordinates

- **point2** (Union[Iterable, SkyCoord]) – a two element list/tuple/Numpy array or a SkyCoord of the second point coordinates

**Return type** Quantity

**property domain: str**

The physical domain of the coordinate system (string value). The Domain attribute also controls how Frames align with each other. If the Domain value in a Frame is set, then only Frames with the same Domain value can be aligned with it.

Example values: GRID, FRACTION, PIXEL, AXIS, SKY, SPECTRUM, CURPIC, NDC, BASEPIC, GRAPHICS

Frames created by the user (for instance, using WCSADD) can have any Domain value, but the standard domain names should be avoided unless the standard meanings are appropriate for the Frame being created.

**Return type** str

**property fillFactor**

<Fraction of the Region which is of interest>

**frame**()

Returns a copy of the frame encapsulated by this region.

Note that the frame is not directly accessible; both this method and the underlying starlink-pyast function returns a copy.

**Return type** *ASTFrame*

static **frameFromAstObject**(*ast_object=None*)

Factory method that returns the appropriate Cornish frame object (e.g. ASTSkyFrame) for a given frame.

**Parameters** **ast_object** (Optional[Frame]) – an Ast.Frame object

static **frameFromFITSHeader**(*header*)

Factory method that returns a new ASTFrame from the provided FITS header.

**frameSet**()

Returns a copy of the frameset encapsulated by this region.

From AST docs:

```
The base Frame is the Frame in which the box was originally
defined, and the current Frame is the Frame into which the
Region is currently mapped (i.e. it will be the same as the
Frame returned by astGetRegionFrame).
```

**Return type** *ASTFrameSet*

**framesetWithMappingTo**(*template_frame=None*)

Search this frame (or set) to identify a frame that shares the same coordinate system as the provided template frame.

For example, this method can be used to see if this frame (or frame set) contains a sky frame.

Returns None if no mapping can be found.

**Parameters** **template_frame** (Optional[*ASTFrame*]) – an instance of the type of frame being searched for

**Return type** Optional[*ASTFrame*]

**Returns** a frame that matches the template `classmethod fromFITSHeader`(*fits_header=None*, *uncertainty=4.848e-06*)

Factory method to create a region from the provided FITS header; the returned object will be as specific as possible (but probably an *ASTPolygon*).

The frame is determined from the FITS header.

**Parameters**

- **fits_header** – a FITS header (Astropy, fitsio, an array of cards)
- **uncertainty** (`float`) – defaults to 4.848e-6, an uncertainty of 1 arcsec

`fullyEncloses`(*region*)

Returns 'True' if this region fully encloses the provided region.

**Return type** `bool`

`property id: str`

String which may be used to identify this object.

NOTE: Unlike most other attributes, the value of the ID attribute is not transferred when an Object is copied. Instead, its value is undefined (and therefore defaults to an empty string) in any copy. However, it is retained in any external representation of an Object produced by the astWrite function.

Not sure how to handle the above in this class.

**Return type** `str`

**Returns** string identifier that can be used to uniquely identify this object

`interiorPointMesh`(*npoints=None*)

Returns an array of evenly distributed points that cover the surface of the region. For example, if the region is a box, it will generate a list of points that fill the interior area of the box.

The default value of 'npoints' is 200 for 2D regions and 2000 for three or more dimensions.

**Parameters** **npoints** (`Optional[int]`) – the approximate number of points to generate in the mesh

**Returns** array of points in degrees

`inverseMapping`()

Returns a new mapping object that is the inverse of this mapping.

For example, if the forward transformation of this mapping is pixel to sky, then the forward transformation of the returned mapping will be sky to pixel.

`property isAdaptive`

Boolean attribute that indicates whether the area adapt to changes in the coordinate system.

`property isBounded: bool`

Boolean attribute that indicates whether the region is bounded.

**Return type** `bool`

`property isClosed: bool`

Boolean attribute that indicates whether the boundary be considered to be inside the region.

**Return type** `bool`

`isFullyWithin`(*region*)

Returns 'True' if this region is fully within the provided region.

**Return type** `bool`

`isIdenticalTo`(*region*)
Returns 'True' if this region is identical (to within their uncertainties) to the provided region, 'False' otherwise.

**Return type** `bool`

`property isLinear: bool`
Returns True if the mapping is linear

**Return type** `bool`

`property isNegated`
Boolean attribute that indicates whether the original region has been negated.

`isNegationOf`(*region*)
Returns 'True' if this region is the exact negation of the provided region.

`property isSimple: bool`
Returns True if the mapping has been simplified.

**Return type** `bool`

`property isSkyFrame: bool`
Returns `True` if this is a SkyFrame, `False` otherwise.

**Return type** `bool`

`label`(*axis=None*)
Return the label for the specified axis.

**Return type** `str`

`mapRegionMesh`(*mapping=None*, *frame=None*)
Returns a new ASTRegion that is the same as this one but with the specified coordinate system.

**Parameters**

- `mapping` (*~cornish.mapping.ASTMapping* class) – The mapping to transform positions from the current ASTRegion to those specified by the given frame.
- `frame` (*~cornish.frame.ASTFrame* class) – Coordinate system frame to convert the current ASTRegion to.

**Returns** **region** – A new region object covering the same area but in the frame specified in *frame*.

**Return type** *ASTRegion*

**Raises** `Exception` – An exception is raised for missing parameters.

`maskOnto`(*image=None*, *mapping=None*, *fits_coordinates=True*, *lower_bounds=None*, *mask_inside=True*, *mask_value=nan*)
Apply this region as a mask on top of the provided image; note: the image values are overwritten!

**Parameters**

- `image` – numpy.ndarray of pixel values (or other array of values)
- `mapping` – mapping from this region to the pixel coordinates of the provided image
- `fits_coordinates` (`bool`) – use the pixel coordinates of a FITS file (i.e. origin = [0.5, 0.5] for 2D)
- `lower_bounds` – lower bounds of provided image, only specify if not using FITS coordinates
- `mask_inside` – True: mask the inside of this region; False: mask outside of this region

- `mask_value` – the value to set the masked image pixels to

**Returns** number of pixels in image masked

`property meshSize: int`
Number of points used to create a mesh covering the region.

**Return type** `int`

`property naxes: int`
Returns the number of axes for the frame.

**Return type** `int`

`negate()`
Negate the region, i.e. points inside the region will be outside, and vice versa.

`property numberOfInputCoordinates`
Number of dimensions of the space in which the Mapping's input points reside. This property gives the number of coordinate values required to specify an input point for a Mapping.

**Returns** number of input dimensions described by this mapper

`property numberOfOutputCoordinates`
Number of dimensions of the space in which the Mapping's output points reside. This property gives the number of coordinate values required to specify an output point for a Mapping.

**Returns** number of output dimensions described by this mapper

`offsetAlongGeodesicCurve`(*point1*, *point2*, *offset*)
Coordinates and offset value should be in the units of the frame, e.g. pixels, degrees.

In a sky frame, the line will be curved. In a basic frame, the line will be straight.

**Parameters**

- `point1` (`Iterable`) – a two element list/tuple/NumPy array of the first point coordinates
- `point2` (`Iterable`) – a two element list/tuple/NumPy array of the second point coordinates
- `offset` (`Quantity`) – a distance along the geodesic sphere connecting the two points

`overlaps`(*region*)
Return `True` if this region overlaps with the provided one.

**Return type** `bool`

`pointInRegion`(*point*)
Returns `True` if the provided point lies inside this region, `False` otherwise.

If no units are specified degrees are assumed.

**Parameters** `point` (`Union[Iterable, SkyCoord, ndarray]`) – a coordinate point in the same frame as this region

**Return type** `bool`

`property points: numpy.ndarray`
The array of points that define the region. The interpretation of the points is dependent on the type of shape in question.

Box: returns two points; the center and a box corner. Circle: returns two points; the center and a point on the circumference. CmpRegion: no points returned; to get points that define a compound region, call this method on each of the component regions via the method "decompose". Ellipse: three points: 1) center, 2) end of one axis, 3) end of the other axis Interval: two points: 1) lower bounds position, 2) upper bounds position (reversed when interval is an excluded interval) NullRegion: no points PointList: positions that the list was constructed with Polygon: vertex positions used to construct the polygon Prism: no points (see CmpRegion)

NOTE: points returned reflect the current coordinate system and may be different from the initial construction

**Return type** `ndarray`

**Returns** NumPy array of coordinate points in degrees, shape (ncoord,2), e.g. [[ra1,dec1], [ra2, dec2], . . . , [ra_n, dec_n]]

`regionWithMapping`(*map=None*, *frame=None*)

Returns a new ASTRegion with the coordinate system from the supplied frame.

Corresponds to the `astMapRegion` C function (`starlink.Ast.mapregion`).

**Parameters**

- **`map`** – A mapping that can convert coordinates from the system of the current region to that of the supplied frame.
- **`frame`** – A frame containing the coordinate system for the new region.

**Return type** *ASTRegion*

**Returns** new ASTRegion with a new coordinate system

`setLabelForAxis`(*label=None*)

Set the label for the specified axis.

`setUnitForAxis`(*axis=None*, *unit=None*)

Set the unit as a string value for the specified axis.

`property system`

String which identifies the coordinate system represented by the Frame. The system is `Cartesian` by default, but can have other values for subclasses of Frame, e.g. `FK4`, `Galactic`.

`property title: str`

Returns the frame title, a string describing the coordinate system which the frame represents.

**Return type** `str`

`abstract toPolygon`(*npoints=200*, *maxerr=<Quantity 1. arcsec>*)

Method that guarantees returning a polygon that describes or approximates this region.

This method provides a common interface to create polygons from different region types. Calling this on an ASTPolygon will return itself; calling it on an ASTCircle will return a polygon that approximates the circle. The parameters 'npoints' and 'maxerr' will be used only when appropriate.

**Parameters**

- **`npoints`** – number of points to sample from the region's boundary for the resulting polygon
- **`maxerr`** (*`astropy.units.Quantity`*) –

**Return type** *ASTPolygon*

`transform`(*points=None*)

Transform the coordinates of a set of points provided according the mapping defined by this object.

**Parameters**

- **`in`** – input list of coordinates as numpy.ndarray, any iterable list accepted 2-dimensional array with shape (nin,npoint)

- **out** – output coordinates

Format of points:

```
[ [ values on axis 1 ], [ values on axis 2 ], ... ]
```

e.g. sky to pixel:

```
[ [ra1, ra2, ...], [dec1, dec2, ...] ]
```

**Return type** ndarray

**property uncertainty**
Uncertainties associated with the boundary of the Box.

The uncertainty in any point on the boundary of the Box is found by shifting the supplied "uncertainty" Region so that it is centered at the boundary point being considered. The area covered by the shifted uncertainty Region then represents the uncertainty in the boundary position. The uncertainty is assumed to be the same for all points.

**unit**(*axis=None*)
Return the unit for the specified axis.

## 2.1.5 ASTPolygon

**class** cornish.**ASTPolygon**(*ast_object=None*, *frame=None*, *points=None*, *fits_header=None*)
Bases: *cornish.region.region.ASTRegion*

ASTPolygon is an ASTRegion that represents a polygon, a collection of vertices on a sphere in a 2D plane.

Accepted signatures for creating an ASTPolygon:

```
p = ASTPolygon(frame, points)
p = ASTPolygon(fits_header, points)  # get the frame from the FITS header provided
p = ASTPolygon(ast_object)           # where ast_object is a starlink.Ast.Polygon␣
↪object
```

Points may be provided as a list of coordinate points, e.g.

```
[(x1, y1), (x2, y2), ... , (xn, yn)]
```

or as two parallel arrays, e.g.

```
[[x1, x2, x3, ..., xn], [y1, y2, y3, ..., yn]]
```

A string format that can be parsed as above is also accepted, e.g.:

```
"((131.758,5.366),(131.759,3.766),(132.561,3.767),(133.363,3.766),(133.364,5.366),
↪(132.577,5.367))"
```

**Parameters**

- **ast_object** (Optional[Polygon]) – create a new ASTPolygon from an existing starlink.Ast.Polygon object
- **frame** (Union[*ASTFrame*, Frame, *ASTRegion*, Region, None]) – the frame the provided points lie in, accepts either ASTFrame or starlink.Ast.Frame objects

- **`points`** – points in degrees that describe the polygon, may be a list of pairs of points or two parallel arrays of axis points

**Returns** Returns a new `ASTPolygon` object.

**`angle`**(*vertex=None*, *points=None*)
Calculate the angle in this frame between two line segments connected by a point.

Let A = point1, C = point2, and B = the vertex point. This method calculates the angle between the line segments AB and CB.

If the frame is a sky frame, lines are drawn on great circles. Units are assumed to be degrees if not provided with units, e.g. as an astropy.coordinates.SkyCoord or astropy.units.Quantity values.

**Parameters**

- **`vertex`** (`Optional[Iterable]`) – a two element list/tuple/Numpy array or a SkyCoord of the vertex
- **`points`** (`Optional[Container[Union[SkyCoord, Iterable]]]`) – a two element list/tuple/etc. containing two points in this frame

**Return type** `Quantity`

**`property area: astropy.units.quantity.Quantity`**
Returns the area of the polygon as an `astropy.units.quantity.Quantity`. [Not yet implemented for non-sky frames.]

**Return type** `Quantity`

**`property astString`**
Return the AST serialization of this object.

**`ast_description`**()
A string description of this object, customized for each subclass of `ASTObject`.

**`boundaryPointMesh`**(*npoints=None*)
Returns an array of evenly distributed points that cover the boundary of the region. For example, if the region is a box, it will generate a list of points that trace the edges of the box.

The default value of 'npoints' is 200 for 2D regions and 2000 for three or more dimensions.

**Parameters** **`npoints`** (`Optional[int]`) – the approximate number of points to generate in the mesh

**Return type** `ndarray`

**Returns** list of points in degrees

**`boundingBox`**()
Returns an ASTBox region that bounds this region where the box sides align with RA, dec.

**Return type** *ASTBox*

**`boundingCircle`**()
Returns the smallest circle (*`ASTCircle`*) that bounds this region.

It is up to the caller to know that this is a 2D region (only minimal checks are made). :raises cornish.exc.NotA2DRegion: raised when attempting to get a bounding circle for a region that is not 2D

**Return type** *ASTCircle*

**`property bounds: Tuple`**
Returns lower and upper coordinate points that bound this region.

**Return type** `Tuple`

`containsPoint`(*point=None*)

Returns `True` if the provided point lies inside this region, `False` otherwise.

This method is a direct synonym for *`pointInRegion()`*. The name "containsPoint" is more appropriate from an object perspective, but the `pointInRegion` method is present for consistency with the AST library.

**Parameters** **`point`** (`Union[Iterable, SkyCoord, None]`) – a coordinate point in the same frame as this region

**Return type** `bool`

`distance`(*point1*, *point2*)

Distance between two points in this frame.

**Parameters**

- **`point1`** (`Union[Iterable, SkyCoord]`) – a two element list/tuple/Numpy array or a SkyCoord of the first point coordinates
- **`point2`** (`Union[Iterable, SkyCoord]`) – a two element list/tuple/Numpy array or a SkyCoord of the second point coordinates

**Return type** `Quantity`

`property domain: str`

The physical domain of the coordinate system (string value). The Domain attribute also controls how Frames align with each other. If the Domain value in a Frame is set, then only Frames with the same Domain value can be aligned with it.

Example values: `GRID`, `FRACTION`, `PIXEL`, `AXIS`, `SKY`, `SPECTRUM`, `CURPIC`, `NDC`, `BASEPIC`, `GRAPHICS`

Frames created by the user (for instance, using WCSADD) can have any Domain value, but the standard domain names should be avoided unless the standard meanings are appropriate for the Frame being created.

**Return type** `str`

`downsize`(*maxerr=None*, *maxvert=0*)

Returns a new ASTPolygon that contains a subset of the vertices of this polygon.

The subset is chosen so that the returned polygon is a good approximation of this polygon, within the limits specified. The density of points in the new polygon is greater where the curvature of the boundary is greater.

The 'maxerr' parameter set the maximum allowed discrepancy between the original and new polygons as a geodesic distance within the polygon's coordinate frame. Setting this to zero returns a new polygon with the number of vertices set in "maxvert".

The 'maxvert' parameter set the maximum number of vertices the new polygon can have. If this is less than 3, the number of vertices in the returned polygon will be the minimum needed to achieve the maximum discrepancy specified by "maxerr". The unadorned value is in radians, but accepts Astropy unit objects.

**Parameters**

- **`maxerr`** – maximum allowed discrepancy in radians between the original and new polygons as a geodesic distance within the polygon's coordinate frame
- **`maxvert`** – maximum allowed number of vertices in the returned polygon

**Returns** a new ASTPolygon.

`property fillFactor`

<Fraction of the Region which is of interest>

**frame**()

Returns a copy of the frame encapsulated by this region.

Note that the frame is not directly accessible; both this method and the underlying `starlink-pyast` function returns a copy.

**Return type** *ASTFrame*

static **frameFromAstObject**(*ast_object=None*)

Factory method that returns the appropriate Cornish frame object (e.g. `ASTSkyFrame`) for a given frame.

**Parameters** **ast_object** (`Optional[Frame]`) – an `Ast.Frame` object

static **frameFromFITSHeader**(*header*)

Factory method that returns a new ASTFrame from the provided FITS header.

**frameSet**()

Returns a copy of the frameset encapsulated by this region.

From AST docs:

```
The base Frame is the Frame in which the box was originally
defined, and the current Frame is the Frame into which the
Region is currently mapped (i.e. it will be the same as the
Frame returned by astGetRegionFrame).
```

**Return type** *ASTFrameSet*

**framesetWithMappingTo**(*template_frame=None*)

Search this frame (or set) to identify a frame that shares the same coordinate system as the provided template frame.

For example, this method can be used to see if this frame (or frame set) contains a sky frame.

Returns `None` if no mapping can be found.

**Parameters** **template_frame** (`Optional[`*ASTFrame*`]`) – an instance of the type of frame being searched for

**Return type** `Optional[`*ASTFrame*`]`

**Returns** a frame that matches the template

static **fromFITSFilepath**(*path=None*, *hdu=1*)

Create a polygon bounding the region of a 2D image.

**Parameters**

- **path** (`Union[str, PathLike, None]`) – the path to the FITS file
- **hdu** (`int`) – the HDU to open, first HDU = 1

static **fromFITSHeader**(*header=None*, *uncertainty=4.848e-06*)

Creates an ASTPolygon in a sky frame from a FITS header. Header of HDU must be a 2D image and contain WCS information.

**Parameters**

- **header** – a FITS header
- **uncertainty** – TODO: parameter not yet used

**static fromPointsOnSkyFrame**(*frame=None*, *points=None*, *expand_by=<Quantity 20. pix>*)

Create an `ASTPolygon` specifically in a sky frame from an array of points.

Points can be provided in degrees either as an array or coordinate pairs, e.g.

```
np.array([[1,2], [3,4], [5,6]])
```

or as parallel arrays of ra,dec:

```
np.array([[1,3,5], [2,4,6]])
```

**Parameters**

- **points** (`Optional[ndarray]`) – coordinate points, either as a list of coordinate pairs or two parallel ra,dec arrays
- **frame** (`Optional[`*ASTSkyFrame*`]`) – the frame the points lie in, specified as an `ASTSkyFrame` object
- **expand_by** (`Quantity`) – number of pixels to extend the polygon beyond the provided points

**Returns** new `ASTPolygon` object

**fullyEncloses**(*region*)

Returns 'True' if this region fully encloses the provided region.

**Return type** `bool`

**property id: str**

String which may be used to identify this object.

NOTE: Unlike most other attributes, the value of the ID attribute is not transferred when an Object is copied. Instead, its value is undefined (and therefore defaults to an empty string) in any copy. However, it is retained in any external representation of an Object produced by the astWrite function.

Not sure how to handle the above in this class.

**Return type** `str`

**Returns** string identifier that can be used to uniquely identify this object

**interiorPointMesh**(*npoints=None*)

Returns an array of evenly distributed points that cover the surface of the region. For example, if the region is a box, it will generate a list of points that fill the interior area of the box.

The default value of 'npoints' is 200 for 2D regions and 2000 for three or more dimensions.

**Parameters** **npoints** (`Optional[int]`) – the approximate number of points to generate in the mesh

**Returns** array of points in degrees

**inverseMapping**()

Returns a new mapping object that is the inverse of this mapping.

For example, if the forward transformation of this mapping is pixel to sky, then the forward transformation of the returned mapping will be sky to pixel.

**property isAdaptive**

Boolean attribute that indicates whether the area adapt to changes in the coordinate system.

`property isBounded: bool`
Boolean attribute that indicates whether the region is bounded.

**Return type** `bool`

`property isClosed: bool`
Boolean attribute that indicates whether the boundary be considered to be inside the region.

**Return type** `bool`

`isFullyWithin`(*region*)
Returns 'True' if this region is fully within the provided region.

**Return type** `bool`

`isIdenticalTo`(*region*)
Returns 'True' if this region is identical (to within their uncertainties) to the provided region, 'False' otherwise.

**Return type** `bool`

`property isLinear: bool`
Returns True if the mapping is linear

**Return type** `bool`

`property isNegated`
Boolean attribute that indicates whether the original region has been negated.

`isNegationOf`(*region*)
Returns 'True' if this region is the exact negation of the provided region.

`property isSimple: bool`
Returns True if the mapping has been simplified.

**Return type** `bool`

`property isSkyFrame: bool`
Returns `True` if this is a SkyFrame, `False` otherwise.

**Return type** `bool`

`label`(*axis=None*)
Return the label for the specified axis.

**Return type** `str`

`mapRegionMesh`(*mapping=None*, *frame=None*)
Returns a new ASTRegion that is the same as this one but with the specified coordinate system.

**Parameters**

- `mapping` (*~cornish.mapping.ASTMapping* class) – The mapping to transform positions from the current ASTRegion to those specified by the given frame.
- `frame` (*~cornish.frame.ASTFrame* class) – Coordinate system frame to convert the current ASTRegion to.

**Returns** **region** – A new region object covering the same area but in the frame specified in *frame*.

**Return type** *ASTRegion*

**Raises** `Exception` – An exception is raised for missing parameters.

`maskOnto`(*image=None*, *mapping=None*, *fits_coordinates=True*, *lower_bounds=None*, *mask_inside=True*, *mask_value=nan*)

Apply this region as a mask on top of the provided image; note: the image values are overwritten!

**Parameters**

- **image** – numpy.ndarray of pixel values (or other array of values)
- **mapping** – mapping from this region to the pixel coordinates of the provided image
- **fits_coordinates** (`bool`) – use the pixel coordinates of a FITS file (i.e. origin = [0.5, 0.5] for 2D)
- **lower_bounds** – lower bounds of provided image, only specify if not using FITS coordinates
- **mask_inside** – True: mask the inside of this region; False: mask outside of this region
- **mask_value** – the value to set the masked image pixels to

**Returns** number of pixels in image masked

`property meshSize: int`

Number of points used to create a mesh covering the region.

**Return type** `int`

`property naxes: int`

Returns the number of axes for the frame.

**Return type** `int`

`negate()`

Negate the region, i.e. points inside the region will be outside, and vice versa.

`property numberOfInputCoordinates`

Number of dimensions of the space in which the Mapping's input points reside. This property gives the number of coordinate values required to specify an input point for a Mapping.

**Returns** number of input dimensions described by this mapper

`property numberOfOutputCoordinates`

Number of dimensions of the space in which the Mapping's output points reside. This property gives the number of coordinate values required to specify an output point for a Mapping.

**Returns** number of output dimensions described by this mapper

`offsetAlongGeodesicCurve`(*point1*, *point2*, *offset*)

Coordinates and offset value should be in the units of the frame, e.g. pixels, degrees.

In a sky frame, the line will be curved. In a basic frame, the line will be straight.

**Parameters**

- **point1** (`Iterable`) – a two element list/tuple/NumPy array of the first point coordinates
- **point2** (`Iterable`) – a two element list/tuple/NumPy array of the second point coordinates
- **offset** (`Quantity`) – a distance along the geodesic sphere connecting the two points

`overlaps`(*region*)

Return `True` if this region overlaps with the provided one.

**Return type** `bool`

`pointInRegion`(*point*)

Returns `True` if the provided point lies inside this region, `False` otherwise.

If no units are specified degrees are assumed.

**Parameters** **point** (`Union[Iterable, SkyCoord, ndarray]`) – a coordinate point in the same frame as this region

**Return type** `bool`

**property points: numpy.ndarray**

The array of points that define the region. The interpretation of the points is dependent on the type of shape in question.

Box: returns two points; the center and a box corner. Circle: returns two points; the center and a point on the circumference. CmpRegion: no points returned; to get points that define a compound region, call this method on each of the component regions via the method “decompose”. Ellipse: three points: 1) center, 2) end of one axis, 3) end of the other axis Interval: two points: 1) lower bounds position, 2) upper bounds position (reversed when interval is an excluded interval) NullRegion: no points PointList: positions that the list was constructed with Polygon: vertex positions used to construct the polygon Prism: no points (see CmpRegion)

NOTE: points returned reflect the current coordinate system and may be different from the initial construction

**Return type** `ndarray`

**Returns** NumPy array of coordinate points in degrees, shape (ncoord,2), e.g. [[ra1,dec1], [ra2, dec2], . . . , [ra_n, dec_n]]

`regionWithMapping`(*map=None*, *frame=None*)

Returns a new ASTRegion with the coordinate system from the supplied frame.

Corresponds to the `astMapRegion` C function (`starlink.Ast.mapregion`).

**Parameters**

- **map** – A mapping that can convert coordinates from the system of the current region to that of the supplied frame.
- **frame** – A frame containing the coordinate system for the new region.

**Return type** *`ASTRegion`*

**Returns** new ASTRegion with a new coordinate system

`setLabelForAxis`(*label=None*)

Set the label for the specified axis.

`setUnitForAxis`(*axis=None*, *unit=None*)

Set the unit as a string value for the specified axis.

**property system**

String which identifies the coordinate system represented by the Frame. The system is `Cartesian` by default, but can have other values for subclasses of Frame, e.g. `FK4`, `Galactic`.

**property title: str**

Returns the frame title, a string describing the coordinate system which the frame represents.

**Return type** `str`

`toPolygon`(*npoints=200*, *maxerr=<Quantity 1. arcsec>*)

Common interface to return a polygon from a region; here ‘self’ is returned.

The parameters ‘npoints’ and ‘maxerr’ are ignored.

**Return type** *ASTPolygon*

`transform`(*points=None*)
Transform the coordinates of a set of points provided according the mapping defined by this object.

**Parameters**

- **in** – input list of coordinates as numpy.ndarray, any iterable list accepted 2-dimensional array with shape (nin,npoint)
- **out** – output coordinates

Format of points:

```
[ [ values on axis 1 ], [ values on axis 2 ], ... ]
```

e.g. sky to pixel:

```
[ [ra1, ra2, ...], [dec1, dec2, ...] ]
```

**Return type** `ndarray`

**property uncertainty**
Uncertainties associated with the boundary of the Box.

The uncertainty in any point on the boundary of the Box is found by shifting the supplied "uncertainty" Region so that it is centered at the boundary point being considered. The area covered by the shifted uncertainty Region then represents the uncertainty in the boundary position. The uncertainty is assumed to be the same for all points.

`unit`(*axis=None*)
Return the unit for the specified axis.

## 2.2 Mappings and Frames APIs

Classes that describe frames and frame mappings are documented here.

### 2.2.1 ASTMapping

**class** `cornish.mapping.ASTMapping`(*ast_object=None*)
Bases: `cornish.ast_object.ASTObject`

**Parameters** **ast_object** – an existing `starlink.Ast.Mapping` object

**property astString**
Return the AST serialization of this object.

`ast_description`()
A string description of this object, customized for each subclass of `ASTObject`.

**property id: str**
String which may be used to identify this object.

NOTE: Unlike most other attributes, the value of the ID attribute is not transferred when an Object is copied. Instead, its value is undefined (and therefore defaults to an empty string) in any copy. However, it is retained in any external representation of an Object produced by the astWrite function.

Not sure how to handle the above in this class.

**Return type** `str`

**Returns** string identifier that can be used to uniquely identify this object

`inverseMapping()`

Returns a new mapping object that is the inverse of this mapping.

For example, if the forward transformation of this mapping is pixel to sky, then the forward transformation of the returned mapping will be sky to pixel.

`property isLinear: bool`

Returns True if the mapping is linear

**Return type** `bool`

`property isSimple: bool`

Returns True if the mapping has been simplified.

**Return type** `bool`

`property numberOfInputCoordinates`

Number of dimensions of the space in which the Mapping's input points reside. This property gives the number of coordinate values required to specify an input point for a Mapping.

**Returns** number of input dimensions described by this mapper

`property numberOfOutputCoordinates`

Number of dimensions of the space in which the Mapping's output points reside. This property gives the number of coordinate values required to specify an output point for a Mapping.

**Returns** number of output dimensions described by this mapper

`transform`(*points=None*)

Transform the coordinates of a set of points provided according the mapping defined by this object.

**Parameters**

- **`in`** – input list of coordinates as numpy.ndarray, any iterable list accepted 2-dimensional array with shape (nin,npoint)
- **`out`** – output coordinates

Format of points:

```
[ [ values on axis 1 ], [ values on axis 2 ], ... ]
```

e.g. sky to pixel:

```
[ [ra1, ra2, ...], [dec1, dec2, ...] ]
```

**Return type** `ndarray`

### 2.2.2 ASTFrame

`class cornish.mapping.ASTFrame`(*ast_object=None*, *naxes=None*)

Bases: `cornish.mapping.mapping.ASTMapping`

A Frame is a representation of a coordinate system, e.g. Cartesian, RA/dec. It contains information about the labels which appear on the axes, the axis units, a title, knowledge of how to format the coordinate values on each axis, etc.

List and description of `starlink.Ast.Frame` attributes in documentation: Section 7.5.

Ref: http://www.starlink.rl.ac.uk/docs/sun95.htx/sun95se27.html http://www.strw.leidenuniv.nl/docs/starlink/sun210.htx/node71.html

**Parameters** **ast_object** (`Optional[Frame]`) – an existing `starlink.Ast.Frame` object

**angle**(*vertex=None*, *points=None*)

Calculate the angle in this frame between two line segments connected by a point.

Let A = point1, C = point2, and B = the vertex point. This method calculates the angle between the line segments AB and CB.

If the frame is a sky frame, lines are drawn on great circles. Units are assumed to be degrees if not provided with units, e.g. as an astropy.coordinates.SkyCoord or astropy.units.Quantity values.

**Parameters**

- **vertex** (`Optional[Iterable]`) – a two element list/tuple/Numpy array or a SkyCoord of the vertex
- **points** (`Optional[Container[Union[SkyCoord, Iterable]]]`) – a two element list/tuple/etc. containing two points in this frame

**Return type** `Quantity`

**property astString**

Return the AST serialization of this object.

**ast_description**()

A string description of this object, customized for each subclass of `ASTObject`.

**distance**(*point1*, *point2*)

Distance between two points in this frame.

**Parameters**

- **point1** (`Union[Iterable, SkyCoord]`) – a two element list/tuple/Numpy array or a SkyCoord of the first point coordinates
- **point2** (`Union[Iterable, SkyCoord]`) – a two element list/tuple/Numpy array or a SkyCoord of the second point coordinates

**Return type** `Quantity`

**property domain: str**

The physical domain of the coordinate system (string value). The Domain attribute also controls how Frames align with each other. If the Domain value in a Frame is set, then only Frames with the same Domain value can be aligned with it.

Example values: `GRID`, `FRACTION`, `PIXEL`, `AXIS`, `SKY`, `SPECTRUM`, `CURPIC`, `NDC`, `BASEPIC`, `GRAPHICS`

Frames created by the user (for instance, using WCSADD) can have any Domain value, but the standard domain names should be avoided unless the standard meanings are appropriate for the Frame being created.

**Return type** `str`

`static frameFromAstObject`(*ast_object=None*)
: Factory method that returns the appropriate Cornish frame object (e.g. *ASTSkyFrame*) for a given frame.

    **Parameters** **ast_object** (`Optional[Frame]`) – an `Ast.Frame` object

`static frameFromFITSHeader`(*header*)
: Factory method that returns a new ASTFrame from the provided FITS header.

`framesetWithMappingTo`(*template_frame=None*)
: Search this frame (or set) to identify a frame that shares the same coordinate system as the provided template frame.

    For example, this method can be used to see if this frame (or frame set) contains a sky frame.

    Returns `None` if no mapping can be found.

    **Parameters** **template_frame** (`Optional[`*ASTFrame*`]`) – an instance of the type of frame being searched for

    **Return type** `Optional[`*ASTFrame*`]`

    **Returns** a frame that matches the template

`property id: str`
: String which may be used to identify this object.

    NOTE: Unlike most other attributes, the value of the ID attribute is not transferred when an Object is copied. Instead, its value is undefined (and therefore defaults to an empty string) in any copy. However, it is retained in any external representation of an Object produced by the astWrite function.

    Not sure how to handle the above in this class.

    **Return type** `str`

    **Returns** string identifier that can be used to uniquely identify this object

`inverseMapping()`
: Returns a new mapping object that is the inverse of this mapping.

    For example, if the forward transformation of this mapping is pixel to sky, then the forward transformation of the returned mapping will be sky to pixel.

`property isLinear: bool`
: Returns True if the mapping is linear

    **Return type** `bool`

`property isSimple: bool`
: Returns True if the mapping has been simplified.

    **Return type** `bool`

`property isSkyFrame: bool`
: Returns `True` if this is a SkyFrame, `False` otherwise.

    **Return type** `bool`

`label`(*axis=None*)
: Return the label for the specified axis.

    **Return type** `str`

`property naxes: int`
: Returns the number of axes for the frame.

    **Return type** `int`

**property numberOfInputCoordinates**

Number of dimensions of the space in which the Mapping's input points reside. This property gives the number of coordinate values required to specify an input point for a Mapping.

**Returns** number of input dimensions described by this mapper

**property numberOfOutputCoordinates**

Number of dimensions of the space in which the Mapping's output points reside. This property gives the number of coordinate values required to specify an output point for a Mapping.

**Returns** number of output dimensions described by this mapper

**offsetAlongGeodesicCurve**(*point1*, *point2*, *offset*)

Coordinates and offset value should be in the units of the frame, e.g. pixels, degrees.

In a sky frame, the line will be curved. In a basic frame, the line will be straight.

**Parameters**

- **point1** (`Iterable`) – a two element list/tuple/NumPy array of the first point coordinates
- **point2** (`Iterable`) – a two element list/tuple/NumPy array of the second point coordinates
- **offset** (`Quantity`) – a distance along the geodesic sphere connecting the two points

**setLabelForAxis**(*label=None*)

Set the label for the specified axis.

**setUnitForAxis**(*axis=None*, *unit=None*)

Set the unit as a string value for the specified axis.

**property system**

String which identifies the coordinate system represented by the Frame. The system is `Cartesian` by default, but can have other values for subclasses of Frame, e.g. `FK4`, `Galactic`.

**property title: str**

Returns the frame title, a string describing the coordinate system which the frame represents.

**Return type** `str`

**transform**(*points=None*)

Transform the coordinates of a set of points provided according the mapping defined by this object.

**Parameters**

- **in** – input list of coordinates as numpy.ndarray, any iterable list accepted 2-dimensional array with shape (nin,npoint)
- **out** – output coordinates

Format of points:

```
[ [ values on axis 1 ], [ values on axis 2 ], ... ]
```

e.g. sky to pixel:

```
[ [ra1, ra2, ...], [dec1, dec2, ...] ]
```

**Return type** `ndarray`

**unit**(*axis=None*)

Return the unit for the specified axis.

### 2.2.3 ASTFrameSet

**class** cornish.mapping.**ASTFrameSet**(*ast_object=None*, *base_frame=None*)

Bases: *cornish.mapping.frame.frame.ASTFrame*

Create a new AST frame set. Object can be created from an `starlink.Ast.FrameSet` "primitive" (e.g. returned by another object).

A set of inter-related coordinate systems made up of existing mapping's and frames. A FrameSet may be extended by adding a new Frame and associated Mapping.

A FrameSet must have a "base" frame which represents the "native" coordinate system (for example, the pixel coordinates of an image). Similarly, one Frame is termed the current Frame and represents the "currently-selected" coordinates. It might typically be a celestial or spectral coordinate system and would be used during interactions with a user, as when plotting axes on a graph or producing a table of results. Other Frames within the FrameSet represent a library of alternative coordinate systems which a software user can select by making them current.

Accepted signatures for creating an `ASTFrameSet`:

**Parameters**

- **ast_object** (`Optional[FrameSet]`) – an `Ast.astFrame` object from the *starlink-pyast* library
- **base_frame** (`Union[FrameSet,` *ASTFrame*, `None]`) – base frame to create the FrameSet from

**addToBaseFrame**(*frame=None*)

Add a new frame to this frame set's base frame.

**angle**(*vertex=None*, *points=None*)

Calculate the angle in this frame between two line segments connected by a point.

Let A = point1, C = point2, and B = the vertex point. This method calculates the angle between the line segments AB and CB.

If the frame is a sky frame, lines are drawn on great circles. Units are assumed to be degrees if not provided with units, e.g. as an astropy.coordinates.SkyCoord or astropy.units.Quantity values.

**Parameters**

- **vertex** (`Optional[Iterable]`) – a two element list/tuple/Numpy array or a SkyCoord of the vertex
- **points** (`Optional[Container[Union[SkyCoord, Iterable]]]`) – a two element list/tuple/etc. containing two points in this frame

**Return type** `Quantity`

**property astString**

Return the AST serialization of this object.

**ast_description**()

A string description of this object, customized for each subclass of `ASTObject`.

**property baseFrame**

Return the base frame.

**centerCoordinates**()

Returns the coordinates at the center of the frame.

**convert**(*points*, *forward=True*)

**property currentFrame**
Returns the current frame.

**distance**(*point1*, *point2*)
Distance between two points in this frame.

**Parameters**

- **point1** (Union[Iterable, SkyCoord]) – a two element list/tuple/Numpy array or a SkyCoord of the first point coordinates
- **point2** (Union[Iterable, SkyCoord]) – a two element list/tuple/Numpy array or a SkyCoord of the second point coordinates

**Return type** `Quantity`

**property domain: str**
The physical domain of the coordinate system (string value). The Domain attribute also controls how Frames align with each other. If the Domain value in a Frame is set, then only Frames with the same Domain value can be aligned with it.

Example values: `GRID`, `FRACTION`, `PIXEL`, `AXIS`, `SKY`, `SPECTRUM`, `CURPIC`, `NDC`, `BASEPIC`, `GRAPHICS`

Frames created by the user (for instance, using WCSADD) can have any Domain value, but the standard domain names should be avoided unless the standard meanings are appropriate for the Frame being created.

**Return type** `str`

**frameAtIndex**(*frame_index*)
Return the frame at the specified index within this frame set.

**Return type** *ASTFrame*

**static frameFromAstObject**(*ast_object=None*)
Factory method that returns the appropriate Cornish frame object (e.g. *ASTSkyFrame*) for a given frame.

**Parameters** **ast_object** (Optional[Frame]) – an `Ast.Frame` object

**static frameFromFITSHeader**(*header*)
Factory method that returns a new ASTFrame from the provided FITS header.

**framesetWithMappingTo**(*template_frame=None*)
Search this frame (or set) to identify a frame that shares the same coordinate system as the provided template frame.

For example, this method can be used to see if this frame (or frame set) contains a sky frame.

Returns `None` if no mapping can be found.

**Parameters** **template_frame** (Optional[*ASTFrame*]) – an instance of the type of frame being searched for

**Return type** Optional[*ASTFrame*]

**Returns** a frame that matches the template

**static fromFITSHeader**(*fits_header=None*)
Static method that returns a FrameSet object read from the provided FITS header.

**static fromFrames**(*frame1*, *frame2*)
Static method to create a frame set from two existing frames.

A frame set is a mapping between two frames that can convert coordinates from one frame (the “base” frame) to the other frame (the “current” frame).

**Parameters**

- **frame1** (Union[*ASTFrame*, Frame]) – the "base" frame (frame to convert coordinates from)
- **frame2** (Union[*ASTFrame*, Frame]) – the "current" frame (frame to convert coordinates to)

**property id: str**

String which may be used to identify this object.

NOTE: Unlike most other attributes, the value of the ID attribute is not transferred when an Object is copied. Instead, its value is undefined (and therefore defaults to an empty string) in any copy. However, it is retained in any external representation of an Object produced by the astWrite function.

Not sure how to handle the above in this class.

**Return type** `str`

**Returns** string identifier that can be used to uniquely identify this object

**inverseMapping**()

Returns a new mapping object that is the inverse of this mapping.

For example, if the forward transformation of this mapping is pixel to sky, then the forward transformation of the returned mapping will be sky to pixel.

**property isLinear: bool**

Returns True if the mapping is linear

**Return type** `bool`

**property isSimple: bool**

Returns True if the mapping has been simplified.

**Return type** `bool`

**property isSkyFrame: bool**

Returns `True` if this is a SkyFrame, `False` otherwise.

**Return type** `bool`

**label**(*axis=None*)

Return the label for the specified axis.

**Return type** `str`

**property mapping**

Return an object that maps points from the base frame to the current frame of this frame set.

**property naxes: int**

Returns the number of axes for the frame.

**Return type** `int`

**property numberOfInputCoordinates**

Number of dimensions of the space in which the Mapping's input points reside. This property gives the number of coordinate values required to specify an input point for a Mapping.

**Returns** number of input dimensions described by this mapper

**property numberOfOutputCoordinates**

Number of dimensions of the space in which the Mapping's output points reside. This property gives the number of coordinate values required to specify an output point for a Mapping.

**Returns** number of output dimensions described by this mapper

**offsetAlongGeodesicCurve**(*point1*, *point2*, *offset*)

Coordinates and offset value should be in the units of the frame, e.g. pixels, degrees.

In a sky frame, the line will be curved. In a basic frame, the line will be straight.

**Parameters**

- **point1** (Iterable) – a two element list/tuple/NumPy array of the first point coordinates
- **point2** (Iterable) – a two element list/tuple/NumPy array of the second point coordinates
- **offset** (Quantity) – a distance along the geodesic sphere connecting the two points

**pix2world**(*points*)

Convert provided coordinates from a world frame to a pixel frame.

This method will throw a `cornish.exc.FrameNotAvailable` exception if the frame set does not contain both a pixel and world frame.

Format of points:

```
[ [ values on axis 1 ], [ values on axis 2 ], ... ]
```

e.g. pixel to sky:

```
[ [x1, x2, ...], [y1, y2, ...] ]
```

A single point may also be specified alone, e.g. `[a,b]` or `np.array([a,b])`.

**Parameters** **points** (Iterable) – input list of coordinates as numpy.ndarray, 2-dimensional array with shape (2,npoint)

**Return type** ndarray

**removeCurrentFrame**()

Remove the current frame from the frame set.

**setLabelForAxis**(*label=None*)

Set the label for the specified axis.

**setUnitForAxis**(*axis=None*, *unit=None*)

Set the unit as a string value for the specified axis.

**property system**

String which identifies the coordinate system represented by the Frame. The system is `Cartesian` by default, but can have other values for subclasses of Frame, e.g. `FK4`, `Galactic`.

**property title: str**

Returns the frame title, a string describing the coordinate system which the frame represents.

**Return type** str

**transform**(*points=None*)

Transform the coordinates of a set of points provided according the mapping defined by this object.

**Parameters**

- **in** – input list of coordinates as numpy.ndarray, any iterable list accepted 2-dimensional array with shape (nin,npoint)
- **out** – output coordinates

Format of points:

```
[ [ values on axis 1 ], [ values on axis 2 ], ... ]
```

e.g. sky to pixel:

```
[ [ra1, ra2, ...], [dec1, dec2, ...] ]
```

**Return type** `ndarray`

**`unit`**(*axis=None*)
Return the unit for the specified axis.

**`world2pix`**(*points*)
Convert provided coordinates from a world frame to a pixel frame.

This method will throw a `cornish.exc.FrameNotAvailable` exception if the frame set does not contain both a pixel and world frame.

Points must have the shape (2,n), e.g.:

```
[ [ra1, ra2, ...], [dec1, dec2, ...] ]
```

A single point may also be specified alone, e.g. `[a,b]` or `np.array([a,b])`.

**Parameters** **`points`** (`Union[Iterable, SkyCoord]`) – input list of coordinates as numpy.ndarray, 2-dimensional array with shape (2,npoints); units are assumed to be degrees if not specified via `astropy.units.Quantity`

**Return type** `ndarray`

### 2.2.4 ASTSkyFrame

`class cornish.mapping.`**`ASTSkyFrame`**(*ast_object=None*, *equinox=None*, *system=None*, *epoch=None*)
Bases: *`cornish.mapping.frame.frame.ASTFrame`*

A SkyFrame is a specialised form of Frame which describes celestial longitude/latitude coordinate systems.

Systems available in AST:

`ICRS`, `J2000`, `AZEL`, `ECLIPTIC`, `FK4`, `FK4-NO-E`, `FK4_NO_E`, `FK5`, `EQUATORIAL`, `GALACTIC`, `GAPPT`, `GEOCENTRIC`, `APPARENT`, `HELIOECLIPTIC`, `SUPERGALACTIC`

**Parameters**

- **`ast_object`** (`Optional[SkyFrame]`) – an existing `starlink.Ast.SkyFrame` object
- **`equinox`** (`Optional[str]`) – frame equinox, default value `2000.0`
- **`system`** (`Optional[str]`) – coordinate system used to describe positions within the domain, see AST System documentation, default value = `ICRS`
- **`epoch`** (`Optional[str]`) – epoch of the mean equinox as a string value, e.g. `J2000.0`, `B1950.0`, default = `2000.0`

**`angle`**(*vertex=None*, *points=None*)
Calculate the angle in this frame between two line segments connected by a point.

Let A = point1, C = point2, and B = the vertex point. This method calculates the angle between the line segments AB and CB.

If the frame is a sky frame, lines are drawn on great circles. Units are assumed to be degrees if not provided with units, e.g. as an astropy.coordinates.SkyCoord or astropy.units.Quantity values.

**Parameters**

- **`vertex`** (`Optional[Iterable]`) – a two element list/tuple/Numpy array or a SkyCoord of the vertex
- **`points`** (`Optional[Container[Union[SkyCoord, Iterable]]]`) – a two element list/tuple/etc. containing two points in this frame

**Return type** `Quantity`

**`property astString`**
Return the AST serialization of this object.

**`ast_description()`**
A string description of this object, customized for each subclass of `ASTObject`.

**`distance`**(*point1*, *point2*)
Distance between two points in this frame.

**Parameters**

- **`point1`** (`Union[Iterable, SkyCoord]`) – a two element list/tuple/Numpy array or a SkyCoord of the first point coordinates
- **`point2`** (`Union[Iterable, SkyCoord]`) – a two element list/tuple/Numpy array or a SkyCoord of the second point coordinates

**Return type** `Quantity`

**`property domain: str`**
The physical domain of the coordinate system (string value). The Domain attribute also controls how Frames align with each other. If the Domain value in a Frame is set, then only Frames with the same Domain value can be aligned with it.

Example values: `GRID`, `FRACTION`, `PIXEL`, `AXIS`, `SKY`, `SPECTRUM`, `CURPIC`, `NDC`, `BASEPIC`, `GRAPHICS`

Frames created by the user (for instance, using WCSADD) can have any Domain value, but the standard domain names should be avoided unless the standard meanings are appropriate for the Frame being created.

**Return type** `str`

**`property epoch`**

**`property equinox`**

**Todo:** how to evaluate a valid equinox string?

**`static frameFromAstObject`**(*ast_object=None*)
Factory method that returns the appropriate Cornish frame object (e.g. *`ASTSkyFrame`*) for a given frame.

**Parameters** **`ast_object`** (`Optional[Frame]`) – an `Ast.Frame` object

**`static frameFromFITSHeader`**(*header*)
Factory method that returns a new ASTFrame from the provided FITS header.

**`framesetWithMappingTo`**(*template_frame=None*)
Search this frame (or set) to identify a frame that shares the same coordinate system as the provided template frame.

For example, this method can be used to see if this frame (or frame set) contains a sky frame.

Returns `None` if no mapping can be found.

**Parameters** **`template_frame`** (`Optional[`*`ASTFrame`*`]`) – an instance of the type of frame being searched for

**Return type** `Optional[`*`ASTFrame`*`]`

**Returns** a frame that matches the template

**`classmethod fromFITSHeader`**(*header*)

Returns a SkyFrame built from the WCS in the provided header, if found. Creates a sky frame from the provided FITS header. :raises: exc.NoWCSFound: if no sky frame WCS found

**Return type** *`ASTSkyFrame`*

**`property id: str`**

String which may be used to identify this object.

NOTE: Unlike most other attributes, the value of the ID attribute is not transferred when an Object is copied. Instead, its value is undefined (and therefore defaults to an empty string) in any copy. However, it is retained in any external representation of an Object produced by the astWrite function.

Not sure how to handle the above in this class.

**Return type** `str`

**Returns** string identifier that can be used to uniquely identify this object

**`inverseMapping`**()

Returns a new mapping object that is the inverse of this mapping.

For example, if the forward transformation of this mapping is pixel to sky, then the forward transformation of the returned mapping will be sky to pixel.

**`property isLinear: bool`**

Returns True if the mapping is linear

**Return type** `bool`

**`property isSimple: bool`**

Returns True if the mapping has been simplified.

**Return type** `bool`

**`property isSkyFrame: bool`**

Returns `True` if this is a SkyFrame, `False` otherwise.

**Return type** `bool`

**`label`**(*axis=None*)

Return the label for the specified axis.

**Return type** `str`

**`property naxes: int`**

Returns the number of axes for the frame.

**Return type** `int`

**`property numberOfInputCoordinates`**

Number of dimensions of the space in which the Mapping's input points reside. This property gives the number of coordinate values required to specify an input point for a Mapping.

**Returns** number of input dimensions described by this mapper

**property numberOfOutputCoordinates**

Number of dimensions of the space in which the Mapping's output points reside. This property gives the number of coordinate values required to specify an output point for a Mapping.

**Returns** number of output dimensions described by this mapper

**offsetAlongGeodesicCurve**(*point1*, *point2*, *offset*)

Coordinates and offset value should be in the units of the frame, e.g. pixels, degrees.

In a sky frame, the line will be curved. In a basic frame, the line will be straight.

**Parameters**

- **point1** (`Iterable`) – a two element list/tuple/NumPy array of the first point coordinates
- **point2** (`Iterable`) – a two element list/tuple/NumPy array of the second point coordinates
- **offset** (`Quantity`) – a distance along the geodesic sphere connecting the two points

**setLabelForAxis**(*label=None*)

Set the label for the specified axis.

**setUnitForAxis**(*axis=None*, *unit=None*)

Set the unit as a string value for the specified axis.

**property system**

String which identifies the coordinate system represented by the Frame. The system is `Cartesian` by default, but can have other values for subclasses of Frame, e.g. `FK4`, `Galactic`.

**property title: str**

Returns the frame title, a string describing the coordinate system which the frame represents.

**Return type** `str`

**transform**(*points=None*)

Transform the coordinates of a set of points provided according the mapping defined by this object.

**Parameters**

- **in** – input list of coordinates as numpy.ndarray, any iterable list accepted 2-dimensional array with shape (nin,npoint)
- **out** – output coordinates

Format of points:

```
[ [ values on axis 1 ], [ values on axis 2 ], ... ]
```

e.g. sky to pixel:

```
[ [ra1, ra2, ...], [dec1, dec2, ...] ]
```

**Return type** `ndarray`

**unit**(*axis=None*)

Return the unit for the specified axis.

### 2.2.5 ASTICRSFrame

**class** `cornish.mapping.`**`ASTICRSFrame`**(*equinox='2000.0'*, *epoch='2000.0'*)

Bases: *`cornish.mapping.frame.sky_frame.ASTSkyFrame`*

Factory class that returns an *`ASTSkyFrame`* automatically set to `System=ICRS`, `equinox=2000.0`, `epoch=2000.0`.

**`angle`**(*vertex=None*, *points=None*)

Calculate the angle in this frame between two line segments connected by a point.

Let A = point1, C = point2, and B = the vertex point. This method calculates the angle between the line segments AB and CB.

If the frame is a sky frame, lines are drawn on great circles. Units are assumed to be degrees if not provided with units, e.g. as an astropy.coordinates.SkyCoord or astropy.units.Quantity values.

**Parameters**

- **vertex** (`Optional[Iterable]`) – a two element list/tuple/Numpy array or a SkyCoord of the vertex
- **points** (`Optional[Container[Union[SkyCoord, Iterable]]]`) – a two element list/tuple/etc. containing two points in this frame

**Return type** `Quantity`

**property astString**

Return the AST serialization of this object.

**`ast_description`**()

A string description of this object, customized for each subclass of `ASTObject`.

**`distance`**(*point1*, *point2*)

Distance between two points in this frame.

**Parameters**

- **point1** (`Union[Iterable, SkyCoord]`) – a two element list/tuple/Numpy array or a SkyCoord of the first point coordinates
- **point2** (`Union[Iterable, SkyCoord]`) – a two element list/tuple/Numpy array or a SkyCoord of the second point coordinates

**Return type** `Quantity`

**property domain: str**

The physical domain of the coordinate system (string value). The Domain attribute also controls how Frames align with each other. If the Domain value in a Frame is set, then only Frames with the same Domain value can be aligned with it.

Example values: `GRID`, `FRACTION`, `PIXEL`, `AXIS`, `SKY`, `SPECTRUM`, `CURPIC`, `NDC`, `BASEPIC`, `GRAPHICS`

Frames created by the user (for instance, using WCSADD) can have any Domain value, but the standard domain names should be avoided unless the standard meanings are appropriate for the Frame being created.

**Return type** `str`

**property epoch**

**property equinox**

**Todo:** how to evaluate a valid equinox string?

`static frameFromAstObject(ast_object=None)`
Factory method that returns the appropriate Cornish frame object (e.g. `ASTSkyFrame`) for a given frame.

**Parameters** **ast_object** (`Optional[Frame]`) – an `Ast.Frame` object

`static frameFromFITSHeader(header)`
Factory method that returns a new ASTFrame from the provided FITS header.

`framesetWithMappingTo(template_frame=None)`
Search this frame (or set) to identify a frame that shares the same coordinate system as the provided template frame.

For example, this method can be used to see if this frame (or frame set) contains a sky frame.

Returns `None` if no mapping can be found.

**Parameters** **template_frame** (`Optional[ASTFrame]`) – an instance of the type of frame being searched for

**Return type** `Optional[ASTFrame]`

**Returns** a frame that matches the template

`classmethod fromFITSHeader(header)`
Returns a SkyFrame built from the WCS in the provided header, if found. Creates a sky frame from the provided FITS header. :raises: exc.NoWCSFound: if no sky frame WCS found

**Return type** `ASTSkyFrame`

`property id: str`
String which may be used to identify this object.

NOTE: Unlike most other attributes, the value of the ID attribute is not transferred when an Object is copied. Instead, its value is undefined (and therefore defaults to an empty string) in any copy. However, it is retained in any external representation of an Object produced by the astWrite function.

Not sure how to handle the above in this class.

**Return type** `str`

**Returns** string identifier that can be used to uniquely identify this object

`inverseMapping()`
Returns a new mapping object that is the inverse of this mapping.

For example, if the forward transformation of this mapping is pixel to sky, then the forward transformation of the returned mapping will be sky to pixel.

`property isLinear: bool`
Returns True if the mapping is linear

**Return type** `bool`

`property isSimple: bool`
Returns True if the mapping has been simplified.

**Return type** `bool`

`property isSkyFrame: bool`
Returns `True` if this is a SkyFrame, `False` otherwise.

**Return type** `bool`

`label`(*axis=None*)
: Return the label for the specified axis.

  **Return type** `str`

**property naxes: int**
: Returns the number of axes for the frame.

  **Return type** `int`

**property numberOfInputCoordinates**
: Number of dimensions of the space in which the Mapping's input points reside. This property gives the number of coordinate values required to specify an input point for a Mapping.

  **Returns** number of input dimensions described by this mapper

**property numberOfOutputCoordinates**
: Number of dimensions of the space in which the Mapping's output points reside. This property gives the number of coordinate values required to specify an output point for a Mapping.

  **Returns** number of output dimensions described by this mapper

`offsetAlongGeodesicCurve`(*point1*, *point2*, *offset*)
: Coordinates and offset value should be in the units of the frame, e.g. pixels, degrees.

  In a sky frame, the line will be curved. In a basic frame, the line will be straight.

  **Parameters**

  - **point1** (`Iterable`) – a two element list/tuple/NumPy array of the first point coordinates
  - **point2** (`Iterable`) – a two element list/tuple/NumPy array of the second point coordinates
  - **offset** (`Quantity`) – a distance along the geodesic sphere connecting the two points

`setLabelForAxis`(*label=None*)
: Set the label for the specified axis.

`setUnitForAxis`(*axis=None*, *unit=None*)
: Set the unit as a string value for the specified axis.

**property system**
: String which identifies the coordinate system represented by the Frame. The system is `Cartesian` by default, but can have other values for subclasses of Frame, e.g. `FK4`, `Galactic`.

**property title: str**
: Returns the frame title, a string describing the coordinate system which the frame represents.

  **Return type** `str`

`transform`(*points=None*)
: Transform the coordinates of a set of points provided according the mapping defined by this object.

  **Parameters**

  - **in** – input list of coordinates as numpy.ndarray, any iterable list accepted 2-dimensional array with shape (nin,npoint)
  - **out** – output coordinates

  Format of points:

```
[ [ values on axis 1 ], [ values on axis 2 ], ... ]
```

  e.g. sky to pixel:

```
[ [ra1, ra2, ...], [dec1, dec2, ...] ]
```

**Return type** `ndarray`

**unit**(*axis=None*)
Return the unit for the specified axis.

## 2.2.6 ASTCompoundFrame

class cornish.mapping.**ASTCompoundFrame**(*ast_object=None*, *naxes=None*)
Bases: *cornish.mapping.frame.frame.ASTFrame*

A compound frame is the merging of two existing *ASTFrame* objects.

For example, a compound frame could have celestial coordinates on two axes and an unrelated coordinate (wavelength, perhaps) on a third. Knowledge of the relationships between the axes is preserved internally by the process of constructing the frames which represents them.

**angle**(*vertex=None*, *points=None*)
Calculate the angle in this frame between two line segments connected by a point.

Let A = point1, C = point2, and B = the vertex point. This method calculates the angle between the line segments AB and CB.

If the frame is a sky frame, lines are drawn on great circles. Units are assumed to be degrees if not provided with units, e.g. as an astropy.coordinates.SkyCoord or astropy.units.Quantity values.

**Parameters**

- **vertex** (`Optional[Iterable]`) – a two element list/tuple/Numpy array or a SkyCoord of the vertex
- **points** (`Optional[Container[Union[SkyCoord, Iterable]]]`) – a two element list/tuple/etc. containing two points in this frame

**Return type** `Quantity`

**property astString**
Return the AST serialization of this object.

**ast_description**()
A string description of this object, customized for each subclass of `ASTObject`.

**distance**(*point1*, *point2*)
Distance between two points in this frame.

**Parameters**

- **point1** (`Union[Iterable, SkyCoord]`) – a two element list/tuple/Numpy array or a SkyCoord of the first point coordinates
- **point2** (`Union[Iterable, SkyCoord]`) – a two element list/tuple/Numpy array or a SkyCoord of the second point coordinates

**Return type** `Quantity`

**property domain: str**
The physical domain of the coordinate system (string value). The Domain attribute also controls how Frames align with each other. If the Domain value in a Frame is set, then only Frames with the same Domain value can be aligned with it.

Example values: `GRID`, `FRACTION`, `PIXEL`, `AXIS`, `SKY`, `SPECTRUM`, `CURPIC`, `NDC`, `BASEPIC`, `GRAPHICS`

Frames created by the user (for instance, using WCSADD) can have any Domain value, but the standard domain names should be avoided unless the standard meanings are appropriate for the Frame being created.

**Return type** `str`

`static frameFromAstObject(`*ast_object=None*`)`
Factory method that returns the appropriate Cornish frame object (e.g. *ASTSkyFrame*) for a given frame.

**Parameters** **`ast_object`** (`Optional[Frame]`) – an `Ast.Frame` object

`static frameFromFITSHeader(`*header*`)`
Factory method that returns a new ASTFrame from the provided FITS header.

`framesetWithMappingTo(`*template_frame=None*`)`
Search this frame (or set) to identify a frame that shares the same coordinate system as the provided template frame.

For example, this method can be used to see if this frame (or frame set) contains a sky frame.

Returns `None` if no mapping can be found.

**Parameters** **`template_frame`** (`Optional[`*ASTFrame*`]`) – an instance of the type of frame being searched for

**Return type** `Optional[`*ASTFrame*`]`

**Returns** a frame that matches the template

**`property id: str`**
String which may be used to identify this object.

NOTE: Unlike most other attributes, the value of the ID attribute is not transferred when an Object is copied. Instead, its value is undefined (and therefore defaults to an empty string) in any copy. However, it is retained in any external representation of an Object produced by the astWrite function.

Not sure how to handle the above in this class.

**Return type** `str`

**Returns** string identifier that can be used to uniquely identify this object

`inverseMapping()`
Returns a new mapping object that is the inverse of this mapping.

For example, if the forward transformation of this mapping is pixel to sky, then the forward transformation of the returned mapping will be sky to pixel.

**`property isLinear: bool`**
Returns True if the mapping is linear

**Return type** `bool`

**`property isSimple: bool`**
Returns True if the mapping has been simplified.

**Return type** `bool`

**`property isSkyFrame: bool`**
Returns `True` if this is a SkyFrame, `False` otherwise.

**Return type** `bool`

`label(`*axis=None*`)`
Return the label for the specified axis.

**Return type** `str`

**`property naxes: int`**

Returns the number of axes for the frame.

**Return type** `int`

**`property numberOfInputCoordinates`**

Number of dimensions of the space in which the Mapping's input points reside. This property gives the number of coordinate values required to specify an input point for a Mapping.

**Returns** number of input dimensions described by this mapper

**`property numberOfOutputCoordinates`**

Number of dimensions of the space in which the Mapping's output points reside. This property gives the number of coordinate values required to specify an output point for a Mapping.

**Returns** number of output dimensions described by this mapper

**`offsetAlongGeodesicCurve`**(*point1*, *point2*, *offset*)

Coordinates and offset value should be in the units of the frame, e.g. pixels, degrees.

In a sky frame, the line will be curved. In a basic frame, the line will be straight.

**Parameters**

- **point1** (`Iterable`) – a two element list/tuple/NumPy array of the first point coordinates
- **point2** (`Iterable`) – a two element list/tuple/NumPy array of the second point coordinates
- **offset** (`Quantity`) – a distance along the geodesic sphere connecting the two points

**`setLabelForAxis`**(*label=None*)

Set the label for the specified axis.

**`setUnitForAxis`**(*axis=None*, *unit=None*)

Set the unit as a string value for the specified axis.

**`property system`**

String which identifies the coordinate system represented by the Frame. The system is `Cartesian` by default, but can have other values for subclasses of Frame, e.g. `FK4`, `Galactic`.

**`property title: str`**

Returns the frame title, a string describing the coordinate system which the frame represents.

**Return type** `str`

**`transform`**(*points=None*)

Transform the coordinates of a set of points provided according the mapping defined by this object.

**Parameters**

- **in** – input list of coordinates as numpy.ndarray, any iterable list accepted 2-dimensional array with shape (nin,npoint)
- **out** – output coordinates

Format of points:

```
[ [ values on axis 1 ], [ values on axis 2 ], ... ]
```

e.g. sky to pixel:

```
[ [ra1, ra2, ...], [dec1, dec2, ...] ]
```

**Return type** `ndarray`

**`unit`**(*axis=None*)
Return the unit for the specified axis.

## 2.3 Plotting APIs

See *Plotting Examples* for examples.

### 2.3.1 Matplotlib Interface

**class** `cornish.plot.matplotlib.`**`SkyPlot`**(*extent=None*, *figsize=(12.0, 12.0)*)
Bases: `cornish.plot.cornish.CornishPlot`

A convenience class providing a high level interface for creating sky plots in Matplotlib.

**Parameters**

- **`extent`** (`Optional[`*`ASTRegion`*`]`) – an ASTRegion that encompasses the full area to plot
- **`figsize`** (`Tuple[float, float]`) – width,height of the plot figure in inches (parameter passed directly to `matplotlib.figure.Figure`)

**`addPoints`**(*ra=None*, *dec=None*, *points=None*, *style=1*, *size=None*, *colour=None*, *color=None*)

Draw a point onto an existing plot.

**:widths 20 25 25**

**header-rows** 1

- – marker_style value
  – **style**
    * string equivalent
- – 1
  – **small circle**
    * circle
- – 2
- **cross**
  – cross
- – 3
- **star**
  – star

- – 4
- **larger circle**
  - circle
- – 5
- **x**
  - x
- – 6
- **pixel dot**
  - dot
- – 7
- **triangle pointing up**
  - triangle
- – 8
- **triangle pointing down**
  - triangle down
- – 9
- **triangle pointing left**
  - triangle left
- – 10
- **triangle pointing right**
  - triangle right
- – 11
- …

  **param points** point should be in degrees (e.g. list or numpy.ndarray, or a pair (list/tuple) of astropy.units.Quantity values, or a SkyCoord, or a container of these (all in the same form)

  **param style** an integer corresponding to one of the built-in marker styles

  **param colour** marker plot colour

  **param color** synonym for 'colour'

  **param size** scale point size by this value

`addRegionOutline`(*region*, *colour='#4a7f7b'*, *color=None*, *style=1*)
Overlay the outline of the provided region to the plot.

**Parameters**

- **region** (Union[*ASTRegion*, Region]) – the region to draw
- **colour** (str) – a color name (e.g. black) or hex code (e.g. #4a7f7b)
- **color** – synonym for 'colour'
- **style** (int) – line style: 1=solid, 2=solid, 3=dashes, 4=short dashes, 5=long dashes

**figure**()

Return the matplotlib.figure.Figure object for plot customization outside of this API.

**show**()

Display the plot (passthrough for matplotlib.pyplot.show()).

CHAPTER

THREE

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

# INDEX