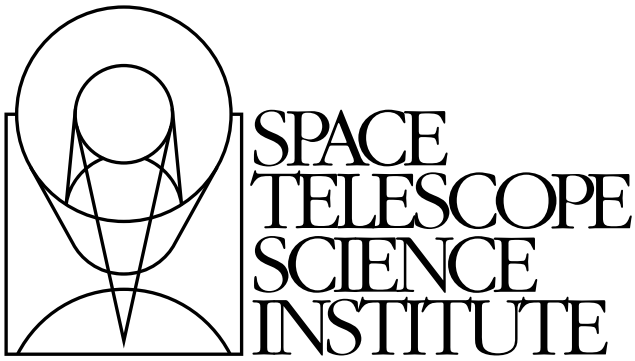


Spherical Geometry Toolkit Documentation

Release 0.2

Michael Droettboom and Pey Lian Lim, STScI

April 18, 2016

CONTENTS

Contents:

CHAPTER

ONE

USER DOCUMENTATION

The `stsci.sphere` library is a pure Python package for handling spherical polygons that represent arbitrary regions of the sky.

1.1 Requirements

- Python 2.7
- Numpy 1.4 or later
- astropy 0.3 or later

1.2 Coordinate representation

Coordinates in world space are traditionally represented by right ascension and declination (*ra* and *dec*), or longitude and latitude. While these representations are convenient, they have discontinuities at the poles, making operations on them trickier at arbitrary locations on the sky sphere. Therefore, all internal operations of this library are done in 3D vector space, where coordinates are represented as (*x*, *y*, *z*) vectors. The *stsci.sphere.vector* module contains functions to convert between (*ra*, *dec*) and (*x*, *y*, *z*) representations.

While any (*x*, *y*, *z*) triple represents a vector and therefore a location on the sky sphere, a distinction must be made between normalized coordinates that fall exactly on the unit sphere, and unnormalized coordinates which do not. A normalized coordinate is defined as a vector whose length is 1, i.e.:

$$\sqrt{x^2 + y^2 + z^2} = 1$$

To prevent unnecessary recomputation, many methods in this library assume that the vectors passed in are already normalized. If this is not the case, *stsci.sphere.vector.normalize_vector* can be used to normalize an array of vectors.

When not working in Cartesian vectors, the library allows the user to work in either degrees or radians. All methods that require or return an angular value have a `degrees` keyword argument. When `degrees` is `True`, these measurements are in degrees, otherwise they are in radians.

Warning: Due to constraints in the precision of intersection calculations, points on the sphere that are closer than 2^{-32} along a Cartesian axis are automatically merged into a single point. This prevents intersections from being missed due to floating point rounding error. There is currently no implemented solution to deal with points that need to be closer together.

1.3 Spherical polygons

Spherical polygons are arbitrary areas on the sky sphere enclosed by great circle arcs. They are represented by the *SphericalPolygon* class.

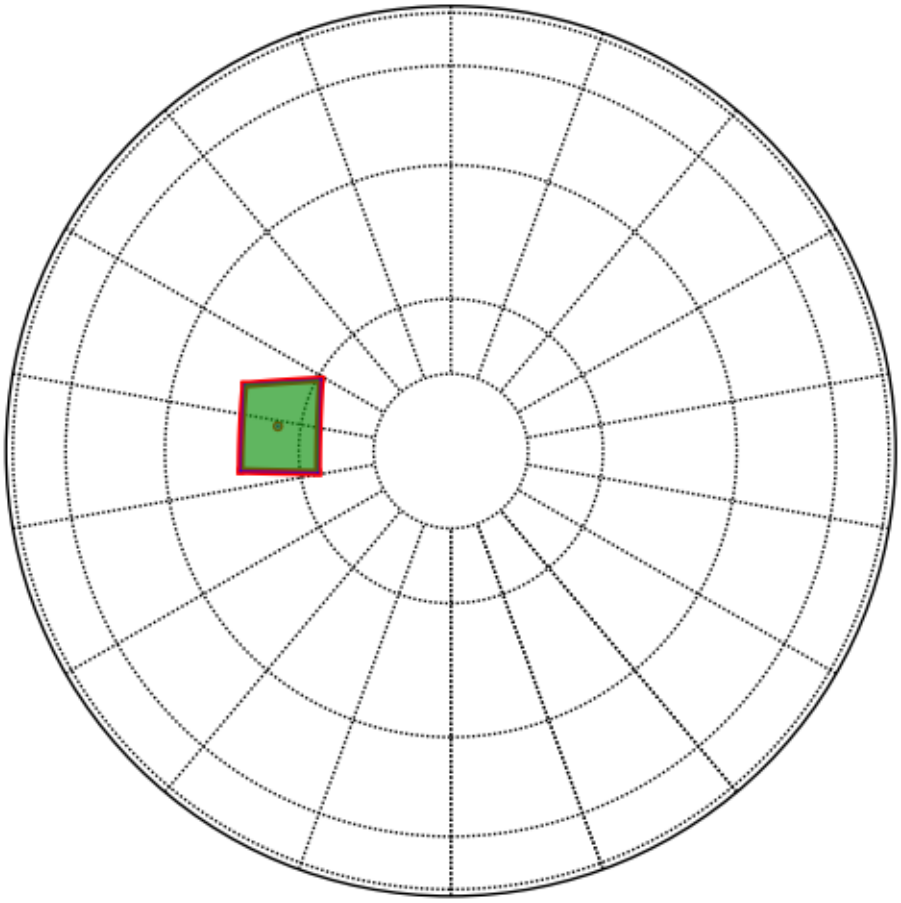
1.3.1 Representation

The points defining the polygon are available from the *points* property. It is a Nx3 array where each row is an (x, y, z) vector, normalized. The polygon points are explicitly closed, i.e., the first and last points are the same.

Where is the inside?

The edges of a polygon serve to separate the "inside" from the "outside" area. On a traditional 2D planar surface, the "inside" is defined as the finite area and the "outside" is the infinite area. However, since the surface of a sphere is cyclical, i.e., it wraps around on itself, the a spherical polygon actually defines two finite areas. To specify which should be considered the "inside" vs. the "outside", the definition of the polygon also has an "inside point" which is just any point that should be considered inside of the polygon.

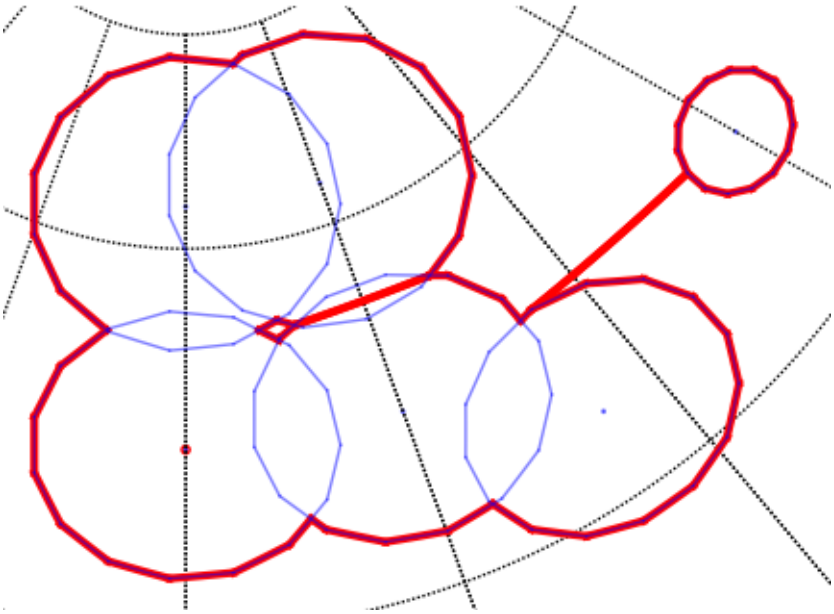
In the following image, the inside point (marked with the red dot) declares that the area of the polygon is the green region, and not the white region.



The inside point of the the polygon can be obtained from the *inside* property.

Cut lines

If the polygon represents two disjoint areas or the polygon has holes, those areas will be connected by cut lines. The following image shows a polygon made from the union of a number of cone areas which has both a hole and a disjoint region connected by cut lines.



1.3.2 Creating spherical polygons

`SphericalPolygon` objects have 4 different constructors:

- `SphericalPolygon`: Takes an array of (*x*, *y*, *z*) points and an inside point.
- `SphericalPolygon.from_radec`: Takes an array of (*ra*, *dec*) points and an inside point.
- `SphericalPolygon.from_cone`: Creates a polygon from a cone on the sky shere. Takes (*ra*, *dec*, *radius*).
- `SphericalPolygon.from_wcs`: Creates a polygon from the footprint of a FITS image using its WCS header keywords. Takes a FITS filename or a `astropy.io.fits.Header` object.

1.3.3 Operations on Spherical Polygons

Once one has a `SphericalPolygon` object, there are a number of operations available:

- `contains_point`: Determines if the given point is inside the polygon.
- `intersects_poly`: Determines if one polygon intersects with another.
- `area`: Determine the area of a polygon.
- `union` and `multi_union`: Return a new polygon that is the union of two or more polygons.
- `intersection` and `multi_intersection`: Return a new polygon that is the intersection of two or more polygons.
- `overlap`: Determine how much a given polygon overlaps another.
- `to_radec`: Convert (*x*, *y*, *z*) points in the polygon to (*ra*, *dec*) points.

- `same_points_as`: Determines if one polygon has the same points as another. When only sorted unique points are considered (default behavior), polygons with same points might not be the same polygons because the order of the points matter.
- *`draw`*: Plots the polygon using matplotlib's Basemap toolkit. This feature is rather bare and intended primarily for debugging purposes.

1.4 Great circle arcs

As seen above, great circle arcs are used to define the edges of the polygon. The *`stsci.sphere.great_circle_arc`* module contains a number of functions that are useful for dealing with them.

- *`length`*: Returns the angular distance between two points on the sphere.
- *`intersection`*: Returns the intersection point between two great circle arcs.
- *`intersects`*: Determines if two great circle arcs intersect.
- *`intersects_point`*: Determines if a point is along the great circle arc.
- *`angle`*: Calculate the angle between two great circle arcs.
- *`midpoint`*: Calculate the midpoint along a great circle arc.

CHAPTER

TWO

API DOCUMENTATION

2.1 Vectors

The `sphere.vector` module contains the basic operations for handling vectors and converting them to and from other representations.

`stsci.sphere.vector.`**`radec_to_vector`**(*ra*, *dec*, *degrees=True*)

Converts a location on the unit sphere from right-ascension and declination to an *x*, *y*, *z* vector.

Parameters

ra, dec : scalars or 1-D arrays

degrees : bool, optional

If `True`, (default) *ra* and *dec* are in decimal degrees, otherwise in radians.

Returns

x, y, z : tuple of scalars or 1-D arrays of the same length

Notes

Where right-ascension is α and declination is δ:

$$x = \cos\alpha \cos\delta$$
$$y = \sin\alpha \cos\delta$$
$$z = \sin\delta$$

`stsci.sphere.vector.`**`vector_to_radec`**(*x*, *y*, *z*, *degrees=True*)

Converts a vector to right-ascension and declination.

Parameters

x, y, z : scalars or 1-D arrays

The input vectors

degrees : bool, optional

If `True` (default) the result is returned in decimal degrees, otherwise radians.

Returns

ra, dec : tuple of scalars or arrays of the same length

Notes

Where right-ascension is α and declination is δ:

$$\alpha = \arctan 2(y, x)$$
$$\delta = \arctan 2(z, \sqrt{x^2 + y^2})$$

stsci.sphere.vector.**normalize_vector**(*xyz*, *output=None*)

Normalizes a vector so it falls on the unit sphere.

Parameters

xyz : Nx3 array of vectors

The input vectors

output : Nx3 array of vectors, optional

The array to store the results in. If `None`, a new array will be created and returned.

Returns

output : Nx3 array of vectors

stsci.sphere.vector.**rotate_around**(*x*, *y*, *z*, *u*, *v*, *w*, *theta*, *degrees=True*)

Rotates the vector (*x*, *y*, *z*) around the arbitrary axis defined by vector (*u*, *v*, *w*) by *theta*.

It is assumed that both (*x*, *y*, *z*) and (*u*, *v*, *w*) are already normalized.

Parameters

x, y, z : doubles

The normalized vector to rotate

u, v, w : doubles

The normalized vector to rotate around

theta : double, or array of doubles

The amount to rotate

degrees : bool, optional

When `True`, *theta* is given in degrees, otherwise radians.

Returns

X, Y, Z : doubles

The rotated vector

2.2 Great circle arcs

The `sphere.great_circle_arc` module contains functions for computing the length, intersection, angle and midpoint of great circle arcs.

Great circles are circles on the unit sphere whose center is coincident with the center of the sphere. Great circle arcs are the section of those circles between two points on the unit sphere.

stsci.sphere.great_circle_arc.**angle**(*A*, *B*, *C*, *degrees=True*)

Returns the angle at *B* between *AB* and *BC*.

Parameters

A, B, C : (*x*, *y*, *z*) triples or Nx3 arrays of triples

Points on sphere.

degrees : bool, optional

If `True` (default) the result is returned in decimal degrees, otherwise radians.

Returns

angle : float or array of floats

The angle at *B* between *AB* and *BC*.

References

[R1]

`stsci.sphere.great_circle_arc.`**`intersection`**(*A*, *B*, *C*, *D*)

Returns the point of intersection between two great circle arcs. The arcs are defined between the points *AB* and *CD*. Either *A* and *B* or *C* and *D* may be arrays of points, but not both.

Parameters

A, B : (*x*, *y*, *z*) triples or Nx3 arrays of triples

Endpoints of the first great circle arc.

C, D : (*x*, *y*, *z*) triples or Nx3 arrays of triples

Endpoints of the second great circle arc.

Returns

T : (*x*, *y*, *z*) triples or Nx3 arrays of triples

If the given arcs intersect, the intersection is returned. If the arcs do not intersect, the triple is set to all NaNs.

Notes

The basic intersection is computed using linear algebra as follows *[R2]*:

$$T = \|(AB)(CD)\|$$

To determine the correct sign (i.e. hemisphere) of the intersection, the following four values are computed:

$$s_1 = ((AB)A) \cdot T$$
$$s_2 = (B(AB)) \cdot T$$
$$s_3 = ((CD)C) \cdot T$$
$$s_4 = (D(CD)) \cdot T$$

For s_n, if all positive T is returned as-is. If all negative, T is multiplied by -1. Otherwise the intersection does not exist and is undefined.

References

http://www.mathworks.com/matlabcentral/newsreader/view_thread/276271

[R2]

`stsci.sphere.great_circle_arc.`**`intersects`**(*A*, *B*, *C*, *D*)

Returns `True` if the great circle arcs between *AB* and *CD* intersect. Either *A* and *B* or *C* and *D* may be arrays of points, but not both.

Parameters

A, B : (*x*, *y*, *z*) triples or Nx3 arrays of triples

Endpoints of the first great circle arc.

C, D : (x, y, z) triples or Nx3 arrays of triples

Endpoints of the second great circle arc.

Returns

intersects : bool or array of bool

If the given arcs intersect, the intersection is returned as `True`.

`stsci.sphere.great_circle_arc.`**`intersects_point`**(*A*, *B*, *C*)

Returns True if point C is along the great circle arc *AB*.

`stsci.sphere.great_circle_arc.`**`length`**(*A*, *B*, *degrees=True*)

Returns the angular distance between two points (in vector space) on the unit sphere.

Parameters

A, B : (x, y, z) triples or Nx3 arrays of triples

The endpoints of the great circle arc, in vector space.

degrees : bool, optional

If `True` (default) the result is returned in decimal degrees, otherwise radians.

Returns

length : scalar or array of scalars

The angular length of the great circle arc.

Notes

The length is computed using the following:

$$\Delta = \arccos(A \cdot B)$$

`stsci.sphere.great_circle_arc.`**`midpoint`**(*A*, *B*)

Returns the midpoint on the great circle arc between *A* and *B*.

Parameters

A, B : (x, y, z) triples or Nx3 arrays of triples

The endpoints of the great circle arc. It is assumed that these points are already normalized.

Returns

midpoint : (x, y, z) triple or Nx3 arrays of triples

The midpoint between *A* and *B*, normalized on the unit sphere.

`stsci.sphere.great_circle_arc.`**`interpolate`**(*A*, *B*, *steps=50*)

Interpolate along the great circle arc.

Parameters

A, B : (x, y, z) triples or Nx3 arrays of triples

The endpoints of the great circle arc. It is assumed thats these points are already normalized.

steps : int

The number of interpolation steps

Returns

array : (x, y, z) triples

The points interpolated along the great circle arc

Notes

This uses Slerp interpolation where Ω is the angle subtended by the arc, and t is the parameter 0 <= t <= 1.

$$\frac{\sin((1-t)\Omega)}{\sin\Omega}A + \frac{\sin(t\Omega)}{\sin\Omega}B$$

2.3 Spherical polygons

The `polygon` module defines the *SphericalPolygon* class for managing polygons on the unit sphere.

class `stsci.sphere.polygon.`**SphericalPolygon**(*init*, *inside=None*)

Polygons are represented by both a set of points (in Cartesian (x, y, z) normalized on the unit sphere), and an inside point. The inside point is necessary, because both the inside and outside of the polygon are finite areas on the great sphere, and therefore we need a way of specifying which is which.

This class contains a list of disjoint closed polygons.

Parameters

init : object

May be either:

- A list of disjoint *SphericalPolygon* objects.
- An Nx3 array of (x, y, z) triples in Cartesian space. These points define the boundary of the polygon. It must be "closed", i.e., the last point is the same as the first.

  It may contain zero points, in which it defines the null polygon. It may not contain one, two or three points. Four points are needed to define a triangle, since the polygon must be closed.

inside : An (x, y, z) triple, optional

If *init* is an array of points, this point must be inside the polygon. If not provided, the mean of the points will be used.

area()

Returns the area of the polygon on the unit sphere in steradians.

The area is computed using a generalization of Girard's Theorem.

if θ is the sum of the internal angles of the polygon, and n is the number of vertices, the area is:

$$S = \theta - (n-2)\pi$$

contains_arc(a, b)

Returns `True` if the polygon fully encloses the arc given by a and b.

contains_point(*point*)

Determines if this *SphericalPolygon* contains a given point.

Parameters

point : an (x, y, z) triple

The point to test.

Returns

contains : bool

Returns `True` if the polygon contains the given *point*.

draw(*m*, ***plot_args*)
Draws the polygon in a matplotlib.Basemap axes.

Parameters
m : Basemap axes object

****plot_args** : Any plot arguments to pass to basemap

classmethod from_cone(*ra*, *dec*, *radius*, *degrees=True*, *steps=16.0*)
Create a new *SphericalPolygon* from a cone (otherwise known as a "small circle") defined using (*ra*, *dec*, *radius*).

The cone is not represented as an ideal circle on the sphere, but as a series of great circle arcs. The resolution of this conversion can be controlled using the *steps* parameter.

Parameters
ra, dec : float scalars

This defines the center of the cone

radius : float scalar

The radius of the cone

degrees : bool, optional

If `True`, (default) *ra*, *dec* and *radius* are in decimal degrees, otherwise in radians.

steps : int, optional

The number of steps to use when converting the small circle to a polygon.

Returns
polygon : *SphericalPolygon* object

classmethod from_radec(*ra*, *dec*, *center=None*, *degrees=True*)
Create a new *SphericalPolygon* from a list of (*ra*, *dec*) points.

Parameters
ra, dec : 1-D arrays of the same length

The vertices of the polygon in right-ascension and declination. It must be "closed", i.e., that is, the last point is the same as the first.

center : (*ra*, *dec*) pair, optional

A point inside of the polygon to define its inside. If no *center* point is provided, the mean of the polygon's points in vector space will be used. That approach may not work for concave polygons.

degrees : bool, optional

If `True`, (default) *ra* and *dec* are in decimal degrees, otherwise in radians.

Returns
polygon : *SphericalPolygon* object

classmethod from_wcs(*fitspath*, *steps=1*, *crval=None*)
Create a new *SphericalPolygon* from the footprint of a FITS WCS specification.

This method requires having astropy installed.

Parameters
fitspath : path to a FITS file, `astropy.io.fits.Header`, or `astropy.wcs.WCS`

Refers to a FITS header containing a WCS specification.

steps : int, optional

The number of steps along each edge to convert into polygon edges.

Returns

polygon : *SphericalPolygon* object

intersection(*other*)

Return a new *SphericalPolygon* that is the intersection of *self* and *other*.

If the intersection is empty, a *SphericalPolygon* with zero subpolygons will be returned.

Parameters

other : *SphericalPolygon*

Returns

polygon : *SphericalPolygon* object

Notes

For implementation details, see the `graph` module.

intersects_arc(*a*, *b*)

Determines if this *SphericalPolygon* intersects or contains the given arc.

intersects_poly(*other*)

Determines if this *SphericalPolygon* intersects another *SphericalPolygon*.

This method is much faster than actually computing the intersection region between two polygons.

Parameters

other : *SphericalPolygon*

Returns

intersects : bool

Returns `True` if this polygon intersects the *other* polygon.

iter_polygons_flat()

Iterate over all base polygons that make up this multi-polygon set.

classmethod multi_intersection(*polygons*, *method=u'parallel'*)

Return a new *SphericalPolygon* that is the intersection of all of the polygons in *polygons*.

Parameters

polygons : sequence of *SphericalPolygon*

method : 'parallel' or 'serial', optional

Specifies the method that is used to perform the intersections:

- 'parallel' (default): A graph is built using all of the polygons, and the intersection operation is computed on the entire thing globally.
- 'serial': The polygon is built in steps by adding one polygon at a time and computing the intersection at each step.

This option is provided because one may be faster than the other depending on context, but it primarily exposed for testing reasons. Both modes should theoretically provide equivalent results.

Returns

polygon : *SphericalPolygon* object

classmethod multi_union(*polygons*)
Return a new *SphericalPolygon* that is the union of all of the polygons in *polygons*.

Parameters
polygons : sequence of *SphericalPolygon*

Returns
polygon : *SphericalPolygon* object

See also:

union

overlap(*other*)
Returns the fraction of *self* that is overlapped by *other*.

Let *self* be *a* and *other* be *b*, then the overlap is defined as:

$$\frac{S_a}{S_{a \cap b}}$$

Parameters
other : *SphericalPolygon*

Returns
frac : float

The fraction of *self* that is overlapped by *other*.

to_radec()
Convert the *SphericalPolygon* footprint to RA and DEC coordinates.

Returns
polyons : iterator

Each element in the iterator is a tuple of the form (*ra*, *dec*), where each is an array of points.

union(*other*)
Return a new *SphericalPolygon* that is the union of *self* and *other*.

Parameters
other : *SphericalPolygon*

Returns
polygon : *SphericalPolygon* object

See also:

multi_union

Notes

For implementation details, see the `graph` module.

inside
Iterate over the inside point of each of the polygons.

points
The points defining the polygons. It is an iterator over disjoint closed polygons, where each element is an Nx3 array of (*x*, *y*, *z*) vectors. Each polygon is explicitly closed, i.e., the first and last points are the same.

polygons
Get a sequence of all of the subpolygons. Each subpolygon may itself have subpolygons. To get a flattened sequence of all base polygons, use *iter_polygons_flat*.

2.4 Graph operations on polygons

This contains the code that does the actual unioning of regions.

class `stsci.sphere.graph.`**Graph** (*polygons*)
A graph of nodes connected by edges. The graph is used to build unions between polygons.

Note: This class is not meant to be used directly. Instead, use `union` and `intersection`.

Parameters
polygons : sequence of `SphericalPolygon` instances

Build a graph from this initial set of polygons.

class **Edge** (*A*, *B*, *source_polygons=[]*)
An *Edge* represents a connection between exactly two *Node* objects. This *Edge* class has no direction.

Parameters
A, B : *Node* instances

source_polygon : `SphericalPolygon` instance, optional

The polygon this edge came from. Used for bookkeeping.

equals (*other*)
Returns `True` if the other edge is between the same two nodes.
Parameters
other : *Edge* instance
Returns
equals : bool

follow (*node*)
Follow along the edge from the given *node* to the other node.
Parameters
node : *Node* instance
Returns
other : *Node* instance

class `Graph.`**Node** (*point*, *source_polygons=[]*)
A *Node* represents a single point, connected by an arbitrary number of *Edge* objects to other *Node* objects.

Parameters
point : 3-sequence (*x*, *y*, *z*) coordinate

source_polygon : `SphericalPolygon` instance, optional

The polygon(s) this node came from. Used for bookkeeping.

equals (*other*, *thres=2e-08*)
Returns `True` if the location of this and the *other* *Node* are the same.
Parameters
other : *Node* instance
The other node.
thres : float
If difference is smaller than this, points are equal. The default value of 2e-8 radians is set based on empirical test cases. Relative threshold based on the actual sizes of polygons is not implemented.

Graph.**add_edge**(*A*, *B*, *source_polygons=[]*)
Add an edge between two nodes.

Note: It is assumed both nodes already belong to the graph.

Parameters
A, B : *Node* instances

source_polygons : SphericalPolygon instance, optional

The polygon(s) this edge came from. Used for bookkeeping.

Returns
edge : *Edge* instance

The new edge

Graph.**add_node**(*point*, *source_polygons=[]*)
Add a node to the graph. It will be disconnected until used in a call to *add_edge*.

Parameters
point : 3-sequence (*x*, *y*, *z*) coordinate

source_polygon : SphericalPolygon instance, optional

The polygon this node came from. Used for bookkeeping.

Returns
node : *Node* instance

The new node

Graph.**add_polygon**(*polygon*)
Add a single polygon to the graph.

Note: Must be called before *union* or *intersection*.

Parameters
polygon : SphericalPolygon instance

Polygon to add to the graph

Graph.**add_polygons**(*polygons*)
Add more polygons to the graph.

Note: Must be called before *union* or *intersection*.

Parameters
polygons : sequence of SphericalPolygon instances

Set of polygons to add to the graph

Graph.**intersection**()
Once all of the polygons have been added to the graph, calculate the intersection.

Returns

points : Nx3 array of (*x*, *y*, *z*) points

This is a list of points outlining the intersection of the polygons that were given to the constructor. If the resulting polygons are disjunct or contain holes, cut lines will be included in the output.

`Graph.`**`remove_edge`**`(`*edge*`)`

Remove an edge from the graph. The nodes it points to remain intact.

Note: It is assumed that *edge* is already a part of the graph.

Parameters

edge : *Edge* instance

`Graph.`**`remove_node`**`(`*node*`)`

Removes a node and all of the edges that touch it.

Note: It is assumed that *Node* is already a part of the graph.

Parameters

node : *Node* instance

`Graph.`**`split_edge`**`(`*edge*, *node*`)`

Splits an *Edge* *edge* at *Node* *node*, removing *edge* and replacing it with two new *Edge* instances. It is intended that *E* is along the original edge, but that is not enforced.

Parameters

edge : *Edge* instance

The edge to split

node : *Node* instance

The node to insert

Returns

edgeA, edgeB : *Edge* instances

The two new edges on either side of *node*.

`Graph.`**`union`**`()`

Once all of the polygons have been added to the graph, join the polygons together.

Returns

points : Nx3 array of (*x*, *y*, *z*) points

This is a list of points outlining the union of the polygons that were given to the constructor. If the original polygons are disjunct or contain holes, cut lines will be included in the output.

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

[R1] Miller, Robert D. Computing the area of a spherical polygon. Graphics Gems IV. 1994. Academic Press.

[R2] Method explained in an e-mail by Roger Stafford.

S

S

T

U

V