scenzgrid-dggs Documentation

Release 0.3

Alexander Raichev

January 29, 2013

CONTENTS

CHAPTER

ONE

INTRODUCTION

SCENZ-Grid is a geographic analysis system currently under development by Landcare Research, New Zealand. scenzgrid-dggs is a collection of Python modules that implements SCENZ-Grid's discrete global grid system (DGGS), which is an instance of the rHEALPix DGGS.

This documentation assumes you are familiar with the rHEALPix DGGS as described in [GRS2013] and familiar with basic Python 2.7.x usage as described in, say, The Python Tutorial.

1.1 Dependencies

- Python 2.7
- numpy and scipy Third-party Python modules for scientific computation.
- Sage (Optional) Third-party Python package for symbolic mathematics. Needed only for a few optional graphics methods.

1.2 Installation

The modules of scenzgrid-dggs are open source and available for download at Landcare's git repository http://code.scenzgrid.org/index.php/p/scenzgrid-py/ and can be cloned via the command `git clone git@code.scenzgrid.org:scenzgrid-py.git`.

1.3 Usage

To use the scenzgrid-dggs modules, start a Python session in the directory where you downloaded the modules and import the modules. Here are some examples. For a list of all methods available, see the application programming interface (API) in the following chapters.

1.3.1 Using `projections.py`

The `projections` module implements the HEALPix and rHEALPix map projections on oblate ellipsoids of revolution whose authalic sphere is the unit sphere. Because of this restriction, it is not so useful by itself. Still, here are a few examples.

Import all the classes, methods, and constants from the module

```
>>> from projections import *
```

Project some points of an oblate ellipsoid of revolution using the HEALPix and (1, 2)-rHEALPix projections.

```
>>> e = 0.2  # Eccentricity of ellipsoid
>>> p = (0, 60)
>>> q = healpix_ellipsoid(*p, e=e, degrees=True); print q
(0.27646508139326409, 1.0618632447907124)
>>> print healpix_ellipsoid(*q, e=e, degrees=True, inverse=True), p
(6.3611093629270335e-15, 59.999997971669899) (0, 60)
>>> q = rhealpix_ellipsoid(0, 60, e=e, north=1, south=2, degrees=True); print q
(-0.27646508139326409, 1.0618632447907124)
>>> print rhealpix_ellipsoid(*q, e=e, north=1, south=2, degrees=True, inverse=True), p
(6.3611093629270335e-15, 59.999997971669899) (0, 60)
```

1.3.2 Using `grids.py`

The `grids` module implements the rHEALPix DGGS and various operations thereupon. It depends upon the `projections` module.

Import all the classes, methods, and constants from the module

```
>>> from grids import *
```

Create the (0, 0)-rHEALPix DGGS based upon the WGS84 ellipsoid. Use degrees for angular measurements

```
>>> E = Earth(ellps='WGS84', degrees=True, south=0, north=0)
>>> print E
____Earth model____
lengths measured in meters and angles measured in degrees
ellipsoid: WGS84
    major radius: 6378137
    flattening factor: 0.00335281066475
    minor radius: 6356752.31425
    eccentricity: 0.0818191908426
    authalic sphere radius: 6371007.18092
central meridian: 0
north pole square position: 0
south pole square position: 0
max resolution: 1.0
max cell level: 15
```

Pick a (longitude-latitude) point on the ellipsoid and find the level 1 cell that contains it

```
>>> p = (0, 15)
>>> c = E.cell_from_point(1, p, surface='ellipsoid'); print c
Q0
```

Find the planar (edge) neighbors of this cell

```
>>> for (direction, cell) in c.neighbors('plane').items():
...     print direction, cell
down Q3
right Q1
up N2
left P2
```

Compute the ellipsoidal shape and ellipsoidal nuclei of these cells

```
>>> cells = c.neighbors().values()
>>> for cell in cells:
...     print cell, cell.ellipsoidal_shape(), cell.nucleus_and_vertices(surface='ellipsoid')[0]
Q3 rectangle (14.999999999999998, 0.0)
Q1 rectangle (45.0, 26.490118738229611)
N2 dart (0.0, 58.528017480415983)
P2 rectangle (-14.999999999999998, 26.490118738229611)
```

CHAPTER

TWO

THE PROJECTIONS MODULE

The Python 2.7 module implements the cartographic projections used in SCENZ-Grid.

CHANGELOG:

- Alexander Raichev (AR), 2011-09-27: Split projection functions and pixelation functions into separate files.
- AR, 2011-10-04: Tidied a little. Improved variable names.
- AR, 2011-10-06: Added more examples and more default arguments.
- AR, 2011-10-23: Added more error-checking, such as the function in_image().
- AR, 2012-03-07: Changed 'rot' to 'rotate'.
- AR, 2012-03-23: Fixed a conceptual bug in combine_caps() and get_cap(). Now polar caps actually assemble on top of the caps numbered north and south.
- AR, 2012-03-26: Changed in_image() to accept a radius argument.
- AR, 2012-04-16: Imported NumPy instead of SciPy.
- AR, 2012-06-13: Simplified get_cap() to cap() and simplified combine_caps().
- AR, 2012-07-10: Corrected a subtle longitude rounding error in healpix_sphere(inverse=True).
- AR, 2012-08-31: Redefined the projections for only the unit sphere and the ellipsoid with eccentricity e and authalic sphere equal to the unit sphere. Will handle scaling of spheres/ellipsoids in grids.py instead. This simplifies the code and puts it more in line with proj4 design, which also deals with scaling as a pre-/post-processing step outside of the projection definitions.
- AR, 2012-09-03: Corrected WGS84_f, the WGS84 ellipsoid flattening factor.

NOTE:

All lengths are measured in meters and all angles are measured in radians unless indicated otherwise.

These projections work only for ellipsoids and not for general (triaxial) ellipsoids.

Except when manipulating positive integers, I avoid the modulo function '%' and insted write everything in terms of 'floor()'. This is because Python interprets the sign of '%' differently than Java or C, and I don't want to confuse people who are translating this code to those languages.

`projections.`**`auth_lat`**(*phi*, *e*, *inverse=False*)

Given a point of geographic latitude *phi* on an ellipse of eccentricity *e*, return the authalic latitude of the point. If *inverse=True*, then compute its inverse approximately.

EXAMPLES:

```
>>> beta = auth_lat(pi/4, WGS84_E); beta
0.78315895611802222
>>> auth_lat(beta, WGS84_E, inverse=True); pi/4
0.78539816331575041
0.7853981633974483
```

projections.**cap**(*x*, *y*, *north=0*, *south=0*, *inverse=False*)

Return the number of the polar cap and region that *(x, y)* lies in. If *inverse=False*, then assume *(x,y)* lies in the image of the HEALPix projection of the unit sphere. If *inverse=True*, then assume *(x,y)* lies in the image of the *(north, south)*-rHEALPix projection of the unit sphere.

INPUT:

- *x, y* - Coordinates in the HEALPix or rHEALPix (if *inverse=True*) projection of the unit sphere.
- *north, south* - Integers between 0 and 3 indicating the positions of the north pole square and south pole square respectively. See rhealpix_sphere() docstring for a diagram.
- *inverse* - (Optional; default = False) Boolean. If False, then compute forward function. If True, then compute inverse function.

OUTPUT:

The pair (cap_number, region). Here region equals 'north' (polar), 'south' (polar), or 'equatorial', indicating where *(x, y)* lies. If region = 'equatorial', then cap_number = None. Suppose now that region != 'equatorial'. If *inverse=False*, then cap_number is the number (= 0, 1, 2, or 3) of the HEALPix polar cap Z that *(x, y)* lies in. If *inverse=True*, then cap_number is the number (= 0, 1, 2, or 3) of the HEALPix polar cap that *(x, y)* will get moved into.

EXAMPLES:

```
>>> cap(-pi/4, pi/4 + 0.1)
(1, 'north')
>>> cap(-3*pi/4 + 0.1, pi/2, inverse=True)
(1, 'north')
```

NOTE:

In the HEALPix projection, the polar caps are labeled 0–3 from east to west like this:

```
    *       *       *       *
  * 0 *   * 1 *   * 2 *   * 3 *
*-------*-------*-------*-------*
|       |       |       |       |
|       |       |       |       |
|       |       |       |       |
*-------*-------*-------*-------*
  * 0 *   * 1 *   * 2 *   * 3 *
    *       *       *       *
```

In the rHEALPix projection these polar caps get rearranged into a square with the caps numbered *north* and *south* remaining fixed. For example, if *north* = 1 and *south* = 3, then the caps get rearranged this way:

```
North polar square:     *-------*
                        | * 3 * |
                        | 0 * 2 |
                        | * 1 * |
                    ----*-------*----

South polar square: ----*-------*----
                        | * 3 * |
                        | 2 * 0 |
                        | * 1 * |
                        *-------*
```

projections.**combine_caps**(*x*, *y*, *north=0*, *south=0*, *inverse=False*)

Rearrange point *(x, y)* in the HEALPix projection by combining the polar caps into two polar squares. Put the north polar square in position *north* and the south polar square in position *south*. If *inverse=True*, uncombine the polar caps.

INPUT:

- *x, y* - Coordinates in the HEALPix projection of the unit sphere.

- *north, south* - Integers between 0 and 3 indicating the positions of the north polar square and south polar square respectively. See rhealpix_sphere() docstring for a diagram.
- *inverse* - (Optional; default = False) Boolean. If False, then compute forward function. If True, then compute inverse function.

EXAMPLES:

```
>>> u, v = -pi/4, pi/3
>>> x, y = combine_caps(u, v); x, y
(-1.8325957145940459, 1.5707963267948966)
>>> combine_caps(x, y, inverse=True); u, v
(-0.78539816339744828, 1.0471975511965976)
(-0.7853981633974483, 1.0471975511965976)
```

projections.**ellipsoid_parameters**(*a*, *f*)

Given the major radius *a* and flattening factor *f* of a ellipsoid, return its minor radius, eccentricity, and authalic sphere radius.

EXAMPLES:

```
>>> ellipsoid_parameters(WGS84_A, WGS84_F)
(6356752.314245179, 0.081819190842621486, 6371007.1809184756)
```

projections.**healpix_ellipsoid**(*u*, *v*, *e*, *lon0=0*, *degrees=False*, *inverse=False*)

Compute the forward and inverse signature functions of the HEALPix projection of an oblate ellipsoid with eccentricity *e* whose authalic sphere is the unit sphere with central meridian *lon0*.

INPUT:

- *u, v* - If *inverse=False*, then these are geographic coordinates. Any input angles are accepted and get standardized to lie in the intervals -pi <= *u* < pi and -pi/2 <= *v* <= pi/2 via wrap_longitude() and wrap_latitude(). If *inverse=True*, then these are planar coordinates in the image of the HEALPix projection of a sphere of radius 1.
- *e* - Eccentricity.
- *lon0* - (Optional; default = 0) Central meridian. Any angle is accepted and gets standardized to lie in the interval -pi <= *lon0* < pi.
- *degrees* - (Optional; default = False) Boolean. If True, then use degrees for input/output instead of radians.
- *inverse* - (Optional; default = False) Boolean. If False, then compute forward function. If True, then compute inverse function.

EXAMPLES:

```
>>> u, v = 82,-45
>>> x, y = healpix_ellipsoid(u, v, e=WGS84_E, degrees=True)
>>> healpix_ellipsoid(x, y,e=WGS84_E, degrees=True, inverse=True); u, v
(81.999999999999986, -44.999999995319058)
(82, -45)
```

projections.**healpix_sphere**(*u*, *v*, *lon0=0*, *degrees=False*, *inverse=False*)

Compute the forward and inverse signature functions of the HEALPix projection of the unit sphere with central meridian *lon0*.

INPUT:

- *u, v* - If *inverse=False*, then these are geographic coordinates. Any input angles are accepted and get standardized to lie in the intervals -pi <= *u* < pi and -pi/2 <= *v* <= pi/2 via wrap_longitude() and wrap_latitude(). If *inverse=True*, then these are planar coordinates in the image of the HEALPix projection of the unit sphere.
- *lon0* - (Optional; default = 0) Central meridian. Any angle is accepted and gets standardized to lie in the interval -pi <= *lon0* < pi.

- *degrees* - (Optional; default = False) Boolean. If True, then use degrees for input/output instead of radians.
- *inverse* - (Optional; default = False) Boolean. If False, then compute forward function. If True, then compute inverse function.

EXAMPLES:

```
>>> healpix_sphere(0, 0, lon0=-45, degrees=True); pi/4
(0.7853981633974483, 0.0)
0.7853981633974483
>>> healpix_sphere(-45, 0, lon0=-45, degrees=True)
(0.0, 0.0)
>>> healpix_sphere(180, 0, lon0=-45, degrees=True)
(-2.3561944901923448, 0.0)
>>> healpix_sphere(-180, 0, lon0=-45, degrees=True)
(-2.3561944901923448, 0.0)
>>> u, v = -50, 82
>>> x, y = healpix_sphere(u, v, degrees=True)
>>> healpix_sphere(x, y, degrees=True, inverse=True); u, v
(-49.999999999999993, 82.000000000000028)
(-50, 82)
```

projections.**in_image**(*x*, *y*, *proj='healpix_sphere'*, *north=0*, *south=0*)

Return True if *(x, y)* lies in the image of the projection *proj* of the unit sphere, where *proj* is either 'healpix_sphere' or 'rhealpix_sphere' and *north* and *south* indicate the positions of the polar squares in case *proj* = 'rhealpix_sphere'. Return False otherwise.

EXAMPLES:

```
>>> eps = 0      # Test boundary points.
>>> hp = [
... (-pi - eps, pi/4),
... (-3*pi/4, pi/2 + eps),
... (-pi/2, pi/4 + eps),
... (-pi/4, pi/2 + eps),
... (0, pi/4 + eps),
... (pi/4, pi/2 + eps),
... (pi/2, pi/4 + eps),
... (3*pi/4, pi/2 + eps),
... (pi + eps, pi/4),
... (pi + eps,-pi/4),
... (3*pi/4,-pi/2 - eps),
... (pi/2,-pi/4 - eps),
... (pi/4,-pi/2 - eps),
... (0,-pi/4 - eps),
... (-pi/4,-pi/2 - eps),
... (-pi/2,-pi/4 - eps),
... (-3*pi/4,-pi/2 - eps),
... (-pi - eps,-pi/4)
... ]
>>> for p in hp:
...     if not in_image(*p):
...             print 'Fail'
...
>>> in_image(0, 0)
True
>>> in_image(0, pi/4 + 0.1)
False
>>> eps = 0      # Test boundary points.
>>> north, south = 0, 0
>>> rhp = [
... (-pi - eps, pi/4 + eps),
... (-pi + north*pi/2 - eps, pi/4 + eps),
```

```
... (-pi + north*pi/2 - eps, 3*pi/4 + eps),
... (-pi + (north + 1)*pi/2 + eps, 3*pi/4 + eps),
... (-pi + (north + 1)*pi/2 + eps, pi/4 + eps),
... (pi + eps, pi/4 + eps),
... (pi + eps,-pi/4 - eps),
... (-pi + (south + 1)*pi/2 + eps,-pi/4 - eps),
... (-pi + (south + 1)*pi/2 + eps,-3*pi/4 - eps),
... (-pi + south*pi/2 - eps,-3*pi/4 - eps),
... (-pi + south*pi/2 -eps,-pi/4 - eps),
... (-pi - eps,-pi/4 - eps)
... ]
>>> for p in rhp:
...     if not in_image(*p, proj='rhealpix_sphere'):
...             print 'Fail'
...
>>> in_image(0, 0,proj='rhealpix_sphere')
True
>>> in_image(0, pi/4 + 0.1, proj='rhealpix_sphere')
False
```

projections.**para_lat**(*phi*, *e*, *inverse=False*)

Given a point at geographic latitude *phi* on an ellipse of eccentricity *e*, return the parametric latitude of the point. If *inverse=True*, then compute its inverse.

EXAMPLES:

```
>>> eta = para_lat(pi/3, WGS84_E); eta
1.0457420826841251
>>> para_lat(eta, WGS84_E, inverse=True); pi/3
1.0471975511965976
1.0471975511965976
```

projections.**rhealpix_ellipsoid**(*u*, *v*, *e*, *lon0=0*, *north=0*, *south=0*, *degrees=False*, *inverse=False*)

Compute the forward and inverse signature functions of the rHEALPix projection of an oblate ellipsoid with eccentricity *e* whose authalic sphere is the unit sphere with central meridian *lon0*. The north pole square is put in position *north*, and the south pole square is put in position *south*.

INPUT:

- *u, v* - If *inverse=False*, then these are geographic coordinates. Any input angles are accepted and get standardized to lie in the intervals -pi <= *u* < pi and -pi/2 <= *v* <= pi/2 via wrap_longitude() and wrap_latitude(). If *inverse= True*, then these are planar coordinates in the image of the rHEALPix projection of the unit sphere.
- *e* - Eccentricity.
- *lon0* - (Optional; default = 0) Central meridian. Any angle is accepted and gets standardized to lie in the interval -pi <= *lon0* < pi.
- *north, south* - (Optional; defaults = 0, 0) Integers between 0 and 3 indicating positions of north polar and south polar squares, respectively. See rhealpix_sphere() docstring for a diagram.
- *degrees* - (Optional; default = False) Boolean. If True, then use degrees for input/output instead of radians.
- *inverse* - (Optional; default = False) Boolean. If False, then compute forward function. If True, then compute inverse function.

EXAMPLES:

```
>>> u, v = 90, 72
>>> x, y = healpix_ellipsoid(u, v, e=WGS84_E, degrees=True)
>>> healpix_ellipsoid(x, y,e=WGS84_E, degrees=True, inverse=True); u, v
(90.000000000000028, 71.99999999914445)
(90, 72)
```

projections.**rhealpix_sphere**(*u*, *v*, *lon0=0*, *north=0*, *south=0*, *degrees=False*, *inverse=False*)

Compute the forward and inverse signature functions of the rHEALPix projection of the unit sphere with central meridian *lon0*. The north pole square is put in position *north*, and the south pole square is put in position *south*.

INPUT:

- *u, v* - If *inverse* = False, then these are geographic coordinates. Any input angles are accepted and get standardized to lie in the intervals -pi <= *u* < pi and -pi/2 <= *v* <= pi/2 via wrap_longitude() and wrap_latitude(). If *inverse* = True, then these are planar coordinates in the image of the rHEALPix projection of the unit sphere.
- *lon0* - (Optional; default = 0) Central meridian. Any angle is accepted and gets standardized to lie in the interval -pi <= *lon0* < pi.
- *north, south* - (Optional; defaults = 0, 0) Integers between 0 and 3 indicating positions of north polar and south polar squares, respectively.
- *degrees* - (Optional; default = False) Boolean. If True, then use degrees for input/output instead of radians.
- *inverse* - (Optional; default = False) Boolean. If False, then compute forward function. If True, then compute inverse function.

EXAMPLES:

```
>>> u, v = -50, 82
>>> x, y = rhealpix_sphere(u, v, north=0, south=0, degrees=True)
>>> rhealpix_sphere(x, y, degrees=True, inverse=True); u, v
(-50.000000000000036, 82.000000000000028)
(-50, 82)
```

NOTE:

The polar squares are labeled 0, 1, 2, 3 from east to west like this:

```
east    lon0    west
*---*---*---*---*
| 0 | 1 | 2 | 3 |
*---*---*---*---*
|   |   |   |   |
*---*---*---*---*
| 0 | 1 | 2 | 3 |
*---*---*---*---*
```

projections.**wrap_latitude**(*phi*, *degrees=False*)

Given a point p on the unit circle at angle *phi* from the positive x-axis, if p lies in the right half of the circle, then return its angle that lies in the interval [-pi/2, pi/2]. If p lies in the left half of the circle, then reflect it through the origin, and return the angle of the reflected point that lies in the interval [-pi/2, pi/2]. If *radians* = True, then *phi* and the output are given in radians. Otherwise, they are given in degrees.

EXAMPLES:

```
>>> wrap_latitude(45, degrees=True)
45.0
>>> wrap_latitude(-45, degrees=True)
-45.0
>>> wrap_latitude(90, degrees=True)
90.0
>>> wrap_latitude(-90, degrees=True)
-90.0
>>> wrap_latitude(135, degrees=True)
-45.0
>>> wrap_latitude(-135, degrees=True)
45.0
```

projections.**wrap_longitude**(*lam*, *degrees=False*)

Given a point p on the unit circle at angle *lam* from the positive x-axis, return its angle theta in the range -pi <= theta < pi. If *radians* = True, then *lam* and the output are given in radians. Otherwise, they are given in degrees.

EXAMPLES:

```
>>> wrap_longitude(2*pi + pi)
-3.1415926535897931
>>> wrap_longitude(-185, degrees=True)
175.0
>>> wrap_longitude(-180, degrees=True)
-180.0
>>> wrap_longitude(185, degrees=True)
-175.0
```

CHAPTER

THREE

THE GRIDS MODULE

This Python 2.7 module implements the SCENZ-Grid discrete global grid system (DGGS), which is an instance of the rHEALPix DGGS.

CHANGELOG:

- Alexander Raichev (AR), 2011-07-11: Initial version
- AR, 2011-07-26: Added new functions. Improved suids.
- AR, 2011-08-05: Extended function capababilities to the WGS84 ellipsoid. Improved suids.
- AR, 2011-08-31: Added optional central meridian shift in HEALPix and rHEALPix projections. Added optional polar squares shift in rHEALPix projection. Added Earth class and restructured and tidied everything. Fixed rounding bug in HEALPix projection.
- AR, 2011-09-01: Added option for entering angles in degrees in projections and set degrees as default mode in Earth class. Fixed a rounding bug in auth_lat().
- AR, 2011-09-27: Split projection functions and grid functions into separate files.
- AR, 2011-10-07: Improved variable and method suids. Tested more. Documented more.
- AR, 2012-03-06: Refactored to update attribute and method suids and redo cell ordering.
- AR, 2012-03-12: Refactored some more and added CellFamily class.
- AR, 2012-03-28: Improved CellFamily methods.
- AR, 2012-04-05: Reformatted code to comform to standard Python style. Fixed minimize(), union(), and intersect() to handle empty cell families properly.
- AR, 2012-04-11: Improved index() and it's inverse.
- AR, 2012-04-13: Rewrote CellFamily to inherit from collections.MutableSequence. Added and optimized CellFamily methods.
- AR, 2012-04-16: Removed wildcard imports to avoid suidspace pollution.
- AR, 2012-04-23: Added a *filter_level* option to cell family intersectionand union operations.
- AR, 2012-04-25: Changed level 0 cell suids to 'N', 'O', ..., 'S' labeled left to right and top to bottom and changed subcell suids to 0, 1, ..., 8 labeled left to right and top to bottom. Consequently had to change the code that converts between cell suid and cell location.
- AR, 2012-05-09: Changed 'name' to 'suid' and improved intersect_all() slightly.
- AR, 2012-06-08: Changed 'suid_xy' to 'suid_colrow' and cleaned up location(), cell_from_point(), and cell_from_region().
- AR, 2012-07-04: Simplified minimize() and eliminated preprocess().
- AR, 2012-07-05: Simplified total ordering definition in Cell class.
- AR, 2012-07-06: Added a width() method to Cell class.
- AR, 2012-09-17: Fixed rounding errors in nucleus_and_vertices() which occurred when interpolation > 0.

- AR, 2012-10-15: Changed names. Changed 'location' to 'ul_vertex', 'center' to 'nucleus', 'corners' to 'vertices'. Introduced the 'surface' keyword for cell operations that depend on the distinction between planar and ellipsoidal cells. - AR, 2012-10-24: Added ellipsoidal_shape() and centroid().

EXAMPLES:

Create a WGS84 ellipsoid model of the Earth:

```
>>> E = Earth(ellps='WGS84', south=1, north=2)
>>> print E
____Earth model____
lengths measured in meters and angles measured in degrees
ellipsoid: WGS84
    major radius: 6378137
    flattening factor: 0.00335281066475
    minor radius: 6356752.31425
    eccentricity: 0.0818191908426
    authalic sphere radius: 6371007.18092
central meridian: 0
north pole square position: 2
south pole square position: 1
max areal resolution: 1.0
max cell level: 15
```

Use the HEALPix projection for this Earth to project the (longitude, latitude) point (45, 60) onto a plane and back again:

```
>>> p = E.healpix(45, 60); p
(5003777.338885325, 6823798.3004998406)
>>> E.healpix(*p, inverse=True)
(45.0, 59.999999998490921)
```

Do the same but use the rHEALPix projection:

```
>>> p = E.rhealpix(45, 60); p
(5003777.338885325, 6823798.3004998406)
>>> E.rhealpix(*p, inverse=True)
(45.0, 59.999999998490921)
```

Create the 'Q3' cell of this Earth and find its four neighbors:

```
>>> c = E.cell('Q3')
>>> for k, v in c.neighbors().items():
...     print k, v
...
down Q6
right Q4
up Q0
left P5
```

Find the nucleus and vertices of this cell and the level 2 cell that contains the nucleus:

```
>>> nucleus = c.nucleus_and_vertices()[0]; nucleus
(1667925.7796284417, 0.0)
>>> print E.cell_from_point(3, nucleus)
Q344
```

NOTES:

All lengths are measured in meters and all angles are measured in radians unless indicated otherwise. Points lying on the plane are given in rectangular (horizontal, vertical) coordinates, and points lying on the ellipsoid are given in geodetic (longitude, latitude) coordinates unless indicated otherwise. Below, GGS abbreviates global grid system.

Except when manipulating positive integers, I avoid the modulo function '%' and insted write everything in terms of 'floor()'. This is because Python interprets the sign of '%' differently than Java or C, and I don't want to confuse

people who are translating this code to those languages.

class grids.**Cell**(*earth*, *suid=None*, *level_order_index=None*, *post_order_index=None*)
Bases: object

Represents a cell of the planar or ellipsoidal global grid system (GGS) relative to a given Earth model. Cell identifiers are of the form (p_0, p_1,...,p_l), where p_0 is one of the characters 'A', 'B', 'C', 'D', 'E', 'F' and p_i for i > 0 is one of the characters '0', '1',..., '8'.

area(*surface='plane'*)
Return the area of this cell.

static atomic_rotate(*quarter_turns*, *x*)
Return the function g that represents the table

```
0 1 2
3 4 5
6 7 8
```

rotated anticlockwise by *quarter_turns* quarter turns. The table is then read from left to right and top to bottom to give the values g(0), g(1), ..., g(8).

INPUT:

- *quarter_turns* - 0, 1, 2, or 3.
- *x* - 0, 1, 2, ..., 8, Earth.CELLS0[5], Earth.CELLS0[1], ..., Earth.CELLS0[0].

EXAMPLES:

```
>>> Cell.atomic_rotate(0, 0)
0
>>> Cell.atomic_rotate(1, 0)
2
>>> Cell.atomic_rotate(2, 0)
8
>>> Cell.atomic_rotate(3, 0)
6
>>> Cell.atomic_rotate(4, 0)
0
```

centroid(*surface='plane'*)
Return the centroid of this planar or ellipsoidal cell.

ellipsoidal_shape()
Return the shape of this cell ('rectangle', 'cap', 'dart', or 'trapezoid') when viewed on the ellipsoid.

index(*order='level'*)
Return the index of *self* when it's ordered according to *order*. Here *order* can be 'level' (default) or 'post'. Indices start at 0. The empty cell has index None.

The ordering comes from the way of traversing the tree T of all cells defined as follows. The root of T is a non-cell place holder. The children of the root are the cells A < B < ... < F. The children of a cell in T with suid s are s0 < s1 < ... < s8.

The level order index of a nonempty cell is its position (starting from 0) in the level order traversal of T starting at cell A.

The post order index of a nonempty cell is its position (starting from 0) in the post order traversal of T.

EXAMPLES:

```
>>> E = Earth(ellps='unit_sphere')
>>> c = Cell(E, 'N2')
>>> print c.index(order='level')
8
>>> print c.index(order='post')
2
```

neighbor(*direction*)

Return the neighboring cell of this cell in the direction *direction*. The direction is relative to this planar neighbor diagram, where *self* is the middle cell

```
              up
            *-----*
            |     |
            |     |
            |     |
      *-----*-----*-----*
      |     | 012 |     |
left  |     | 345 |     | right
      |     | 678 |     |
      *-----*-----*-----*
            |     |
            |     |
            |     |
            *-----*
             down
```

The tricky part is that the neighbor relationships of the six level 0 cells is determined by the positions of those cells on the surface of a cube, one on each face (and not on a plane). So sometimes rotating cells in needed to compute neighbors.

INPUT:

- *direction* - One of the strings 'up', 'right', 'down', or 'left'.

EXAMPLES:

```
>>> c = Cell(Earth(), 'N0')
>>> print c.neighbor('down')
N3
```

neighbors()

Return a dictionary of the left, right, down, and up neighbors of this cell. Direction is relative to this planar neighbor diagram, where *self* is the middle cell

```
                    up
                  *-----*
                  |     |
                  |     |
                  |     |
            *-----*-----*-----*
            |     | 012 |     |
      left  |     | 345 |     | right
            |     | 678 |     |
            *-----*-----*-----*
                  |     |
                  |     |
                  |     |
                  *-----*
                   down
```

```
The tricky part is that the neighbor relationships of the
six level 0 cells is determined by the positions of those cells
on the surface of a cube, one on each face (and not a plane).
So sometimes rotating cells in needed to compute neighbors.
```

EXAMPLES:

```
>>> c = Cell(Earth(), 'N0')
>>> for k, v in c.neighbors().items():
...     print k, v
...
```

```
down N3
right N1
up Q2
left R0
```

nucleus_and_vertices (*surface='plane'*, *interpolation=0*)

Return the nucleus of this planar or ellipsoidal cell along with its corner points interpolated by *3**(interpolation)* points along each edge. The nucleus is the first point in the output list and the remaining points are the boundary points enumerated from the upleft corner to the downleft corner to the downright corner to the upright corner to the upleft corner (but not including the downright corner in the list again). The output for ellipsoidal cells is the projection onto the ellipsoid of the output for planar cells. In particular, while the nucleus of a planar cell is its centroid, the nucleus of an ellipsoidal cell is not its centroid. To compute the centroid of a cell, use centroid() below.

NOTE:

The interpolation option is intended mostly for drawing the boundary of *self* with varying degrees of smoothness.

EXAMPLES:

```
>>> E = Earth('unit_sphere')
>>> c = E.cell('N')
>>> c.nucleus_and_vertices()
[(-2.3561944901923448, 1.5707963267948966), (-3.141592653589793, 2.356194490192345), (-3.
```

predecessor (*level=None*)

Return the greatest level *level* cell less than *self*. Note: *self* need not be a level *level* cell.

EXAMPLES:

```
>>> c = Cell(Earth(), 'N08')
>>> print str(c.predecessor())
N07
>>> print str(c.predecessor(0))
None
>>> print str(c.predecessor(1))
None
>>> print str(c.predecessor(3))
N088
```

region ()

Return the region, 'equatorial' or 'polar', of this cell.

rotate (*quarter_turns*)

Return the cell that is the result of rotating this cell *quarter_turns* quarter_turns. Used in neighbor().

EXAMPLES:

```
>>> c = Cell(Earth(), 'N0')
>>> print c.rotate(0)
N0
>>> print c.rotate(1)
N2
>>> print c.rotate(2)
N8
>>> print c.rotate(3)
N6
>>> print c.rotate(4)
N0
```

str9 ()

Return the suid of *self* as a base 9 numeral. Just need to convert the leading character to a digit between 0 and 8.

EXAMPLES:

```
>>> E = Earth()
>>> c = Cell(E, 'N130534')
>>> print c.str9()
0130534
```

subcell (*other*)
Subcell (subset) relation on cells.

EXAMPLES:

```
>>> a = Cell(Earth(), 'N1')
>>> b = Cell(Earth(), 'N')
>>> print a.subcell(b)
True
>>> print b.subcell(a)
False
```

subcells (*level=None*)
Generator function for the set of level *level* subcells of *self*. If *level=None*, then return a generator function for the children of *self*.

EXAMPLES:

```
>>> c = Cell(Earth(), 'N')
>>> print [str(z) for z in c.subcells(2)]
['N00', 'N01', 'N02', 'N03', 'N04', 'N05', 'N06', 'N07', 'N08', 'N10', 'N11', 'N12', 'N13
```

successor (*level=None*)
Return the least level *level* cell greater than *self*. Note: *self* need not be a level *level* cell.

EXAMPLES:

```
>>> c = Cell(Earth(), 'N82')
>>> print str(c.successor())
N83
>>> print str(c.successor(0))
O
>>> print str(c.successor(1))
O0
>>> print str(c.successor(3))
N830
```

suid_colrow ()
Return the pair of row- and column-suids of *self*, each as tuples.

EXAMPLES:

```
>>> E = Earth()
>>> c = Cell(E, 'N73')
>>> xn, yn = c.suid_colrow()
>>> print xn == ('N', 1, 0)
True
>>> print yn == ('N', 2, 1)
True
```

static suid_from_index (*earth*, *index*, *order='level'*)
Return the suid of a cell from its index. The index is according to the cell ordering *order*, which can be 'level' (default) or 'post'. See the *index()* docstring for more details on orderings. For internal use.

ul_vertex (*surface='plane'*)
Return the upper left (northwest) vertex of this planar or ellipsoidal cell.

WARNING: The upper left vertex of a cell might not lie in the cell, because not all cells contain their boundary.

EXAMPLES:

```
>>> c = Cell(Earth('unit_sphere'), 'N0')
>>> print c.ul_vertex() == (-pi, 3*pi/4)
True
```

width(*surface='plane'*)

Return the width of this cell. If *surface='ellipsoid'*, then return None, because ellipsoidal cells don't have a fixed width.

EXAMPLES:

```
>>> c = Cell(Earth('unit_sphere'), 'N8')
>>> print c
N8
>>> c.width() == pi/2*3**(-1)
True
```

class grids.**CellFamily**(*cells=None*, *min_level=None*, *max_level=None*, *sort=True*, *eliminate_subcells=True*)

Bases: _abcoll.MutableSequence

A subcell-free ordered list of cells along with the minimum level of the cells in family the maximum level of the cells in family.

add(*key*)

Add *key* to *self* if *key* is not a subcell of any cell of *self*. Uses bisection to find the insertion point in O(log(len(self))) comparisons.

EXAMPLES:

```
>>> E = Earth()
>>> f = CellFamily(list(E.grid(0)))
>>> del f[2]; print f
['N', 'O', 'Q', 'R', 'S']
>>> c = E.cell('P3')
>>> f.add(c)
>>> print f
['N', 'O', 'P3', 'Q', 'R', 'S']
>>> c = E.cell('O3')
>>> f.add(c)    # Should do nothing.
>>> print f
['N', 'O', 'P3', 'Q', 'R', 'S']
```

insert(*index*, *value*)

Insert the *value* into *self* at index *index*. Warning: for internal use only; does not maintain sorted order. Use *add()* to add cells to *self*.

intersect(*other*, *filter_level=None*)

Return the minimal cell family v for the intersection of the regions of *self* and *other*. If a nonnegative integer *filter_level* is given, then return the cell family comprised of the cell's of v at levels <= *filter_level*.

Uses O(max(m, n)) cell operations, where m = *len(self)* and n = *len(other)*.

EXAMPLES:

```
>>> a = Cell(Earth(), 'N')
>>> b = Cell(Earth(), 'P')
>>> f = CellFamily([b] + list(a.subcells(1))[:4])
>>> print f
['N0', 'N1', 'N2', 'N3', 'P']
>>> g = CellFamily([b] + list(a.subcells(1))[3:])
>>> print g
['N3', 'N4', 'N5', 'N6', 'N7', 'N8', 'P']
>>> print f.intersect(g)
['N3', 'P']
```

```
>>> print f.intersect(g, filter_level=0)
['P']
```

intersect_all (*others*, *filter_level=None*)

Return the minimal cell family v for the intersection of the regions of *self* and the cell families in the list *others*. If a nonnegative integer *filter_level* is given, then return the cell family comprised of the cell's of v at levels <= *filter_level*.

Uses O(n) cell operations, where n is the maximum of *len(self)* and the lengths of the cell families in *others*.

EXAMPLES:

```
>>> E = Earth()
>>> cell_families = []
>>> for c in E.grid(0):
...     cell_families.append(CellFamily([c]))
...
>>> f = cell_families.pop(0)
>>> print f.intersect_all(cell_families)
[]
```

minimize ()

Return the minimal cell family that has the same region as *self*.

Uses O(len(self)) cell operations.

EXAMPLES:

```
>>> E = Earth()
>>> c = Cell(E, 'N')
>>> f = CellFamily(list(c.subcells(1)))
>>> print f
['N0', 'N1', 'N2', 'N3', 'N4', 'N5', 'N6', 'N7', 'N8']
>>> print f.minimize()
['N']
```

union (*other*, *filter_level=None*)

Return the cell family u that is the minimizeed union of *self* and *other*, that is, the smallest cell family whose region is the union of the regions of *self* and *other*. If a nonnegative integer *filter_level* is given, then return the cell family comprised of the cell's of u at levels <= *filter_level*. Do not set *minimize_first=False* unless you know that the input cell families are already minimizeed.

Uses O(max(m, n)) cell operations, where m = *len(self)* and n = *len(other)*.

EXAMPLES:

```
>>> a = Cell(Earth(), 'N')
>>> b1 = Cell(Earth(), 'P1')
>>> f = CellFamily([b1] + list(a.subcells(1))[:4])
>>> print f
['N0', 'N1', 'N2', 'N3', 'P1']
>>> g = CellFamily(list(a.subcells(1))[3:])
>>> print g
['N3', 'N4', 'N5', 'N6', 'N7', 'N8']
>>> print f.union(g)
['N', 'P1']
>>> print f.union(g, filter_level=0)
['N']
```

union_all (*others*, *filter_level=None*, *minimize_first=True*)

Return the cell family u that is the union of the regions of this cell family and cell families in the list *others*. If a nonnegative integer *filter_level* is given, then return the cell family comprised of the cell's of u at levels <= *filter_level*.

Uses O(n) cell operations, where n is the maximum of *len(self)* and the lengths of the cell families in *others*.

EXAMPLES:

```
>>> E = Earth()
>>> cell_families = []
>>> for c in E.grid(1):
...     cell_families.append(CellFamily([c]))
>>> print len(cell_families)
54
>>> f = cell_families.pop(0)
>>> print f
['N0']
>>> print f.union_all(cell_families)
['N', 'O', 'P', 'Q', 'R', 'S']
```

class grids.**Earth**(*ellps='WGS84'*, *degrees=True*, *lon0=0*, *north=0*, *south=0*, *max_areal_resolution=1.0*)

Bases: object

Represents a model of the Earth its planar and ellipsoidal GGSs.

CELLS0 = ['N', 'O', 'P', 'Q', 'R', 'S']

WGS84 = {'a': 6378137, 'b': 6356752.314245179, 'e': 0.081819190842621486, 'figure': 'ellipsoid', 'f': 0.003352810664

WGS84_asphere = {'R': 6371007.1809184756, 'figure': 'sphere'}

cell(*suid=None*, *level_order_index=None*, *post_order_index=None*)

Return a cell of this Earth model either from its SUID or from its level and index. The cell can be interpreted as lying on the plane or on the ellipsoid, depending on the *surface* option given in the instance methods below.

EXAMPLES:

```
>>> E = Earth()
>>> c = E.cell('N45')
>>> print isinstance(c, Cell)
True
>>> print c
N45
```

cell_area(*level*, *surface='plane'*)

For this Earth model, return the area of a planar or ellipsoiadl cell at level *level*.

cell_from_point(*level*, *p*, *surface='plane'*)

For this Earth model, return the planar or ellipsoidal cell at level *level* that contains the point *p*.

INPUT:

- *level* - Cell level.
- *p* - A point on the plane or ellipsoid given in rectangular or geodetic coordinates, respectively.
- *surface* - (Optional; default='plane') One of 'plane' or 'ellipsoid', indicating whether the given point and returned cell both lie on the plane or on the ellipsoid.

EXAMPLES:

```
>>> E = Earth('WGS84')
>>> p = (0, 0)
>>> c = E.cell_from_point(2, p)
>>> print c
Q33
```

cell_from_region(*ul*, *dr*, *surface='plane'*)

Return the smallest planar or ellipsoidal cell wholly containing the region bounded by the axis-aligned

rectangle with upper left and lower right vertices given by the the points *ul* and *dr*, respectively. If such as cell does not exist, then return None. If *surface='plane'*, then *ul* and *dr* and the returned cell are interpreted as lying on the plane. If *surface='ellipsoid'*, then they are interpreted as lying on the ellipsoid and the bounding rectangle is interpreted as being longitude-latitude aligned. To specify an ellipsoidal cap region (whose centroid is the north or south pole and whose boundary is a parallel of latitude), set ul = (-pi, pi/2) or (-pi, -pi/2), indicating a north or south pole, respectively, and set dr = (-pi, phi), where phi is the latitude of the boundary. Works on defunct (point and line) planar rectangles too.

EXAMPLES:

```
>>> E = Earth('unit_sphere')
>>> p = (0, 0)
>>> q = (pi/4, pi/4)
>>> c = E.cell_from_region(p, q)
>>> print c
Q
```

cell_width(*level*, *surface='plane'*)

For this Earth model, return the width of a planar cell at level *level*. If *surface != 'plane'*, then return None, because ellipsoidal cells don't have constant width.

EXAMPLES:

```
>>> E = Earth('unit_sphere')
>>> E.cell_width(0)
1.5707963267948966
>>> E.cell_width(1)
0.5235987755982988
```

child_order = {0: (0, 0), (0, 1): 3, 2: (2, 0), (0, 0): 0, 4: (1, 1), 5: (2, 1), 6: (0, 2), 1: (1, 0), 8: (2, 2), (1, 2): 7, (2, 1): 5,

col = 2

combine_caps(*u*, *v*, *inverse=False*)

Return the combine_caps() transformation of the point *(u, v)* (or its inverse if *inverse=True*) appropriate to this Earth. It maps the HEALPix projection to the rHEALPix projection.

EXAMPLES:

```
>>> E = Earth('unit_sphere')
>>> p = (0, 0)
>>> q = (-pi / 4, pi / 2)
>>> print E.combine_caps(*p) == (0, 0)
True
>>> print E.combine_caps(*q) == (-3 * pi / 4, pi / 2)
True
```

ellipsoids = {'unit_sphere': {'R': 1, 'figure': 'sphere'}, 'WGS84_asphere': {'R': 6371007.1809184756, 'figure': 'sp

grid(*level*)

Generator function for all the cells at level *level*.

EXAMPLES:

```
>>> E = Earth()
>>> g = E.grid(0)
>>> print [str(x) for x in g]
['N', 'O', 'P', 'Q', 'R', 'S']
```

healpix(*u*, *v*, *inverse=False*)

Return the HEALPix projection of point *(u, v)* (or its inverse if *inverse=True*) for this Earth model.

EXAMPLES:

```
>>> E = Earth('unit_sphere')
>>> p = (0, 0)
>>> q = (-180, 90)
>>> print E.healpix(*p) == (0, 0)
True
>>> print E.healpix(*q) == (-3 * pi / 4, pi / 2)
True
```

interval (*a*, *b*)

Generator function for all the level *max(a.level, a.level)* cells between cell *a* and cell *b* (inclusive and with respect to the postorder ordering on cells). Note that *a* and *b* don't have to lie at the same level.

EXAMPLES:

```
>>> E = Earth()
>>> a = E.cell('N10')
>>> b = E.cell('N3')
>>> print [str(z) for z in E.interval(a, b)]
['N10', 'N11', 'N12', 'N13', 'N14', 'N15', 'N16', 'N17', 'N18', 'N20', 'N21', 'N22', 'N23
```

num_cells (*level_a*, *level_b=None*, *subcells=False*)

Return the number of cells at levels *level_a* to *level_b* (inclusive). Assume *level_a* <= *level_b*. If *subcells=True*, then return the number of subcells at levels *level_a* to *level_b* (inclusive) of a cell at level *level_a*. If *level_b=None* and *subcells=False, then return the number of cells at level 'level_a*. If *level_b=None* and *subcells=True*, then return the number of subcells from level *level_a* to level *self.max_level*.

EXAMPLES:

```
>>> E = Earth()
>>> E.num_cells(0)
6
>>> E.num_cells(0, 1)
60
>>> E.num_cells(0, subcells=True)
231627523606480
>>> E.num_cells(0, 1, subcells=True)
10
>>> E.num_cells(5, 6, subcells=True)
10
```

order = 8

project (*p*, *source='lonlat'*, *target='rhealpix'*)

Transform a point *p* from the image of the *source* projection to the image of the *target* projection.

INPUT:

- *p* - A point in geographic space, in the HEALPix projection, or in the rHEALPix projection.
- *source, target* - One of 'lonlat', 'healpix', or 'rhealpix'.

EXAMPLES:

```
>>> from numpy import rad2deg, arcsin
>>> E = Earth('unit_sphere')
>>> p = (-pi / 2, 3 * pi / 4)
>>> phi0 = rad2deg(arcsin(2.0 / 3))
>>> print phi0
41.8103148958
>>> print E.project(p, source='rhealpix', target='lonlat') == (0, phi0)
True
```

random_cell (*level=None*)

Return a cell chosen of level *level* chosen uniformly at random among all level *level* cells. If *level=None*, then *level* is first chosen uniformly randomly in [0,..,self.max_level]

rhealpix(*u*, *v*, *inverse=False*)
Return the rHEALPix projection of the point *(u, v)* (or its inverse if *inverse=True*) appropriate to this Earth.

EXAMPLES:

```
>>> E = Earth('unit_sphere')
>>> p = (0, 0)
>>> q = (-180, -90)
>>> print E.rhealpix(*p) == (0, 0)
True
>>> print E.rhealpix(*q) == (-3 * pi / 4, -pi / 2)
True
```

row = 2

sample(*cells*, *k*)
Return a *k* length list of unique elements chosen at random without replacement from the cell population sequence *cells*.

unit_sphere = {'R': 1, 'figure': 'sphere'}

grids.**scale**(*x*, *factor*, *inverse=False*)
Multiply (all the elements of) *x* by the number *factor* or by 1/*factor* if *inverse=True*. Here *x* can be a number or a tuple of numbers.

EXAMPLES:

```
>>> scale(5, 2)
10
>>> scale(5, 2,inverse=True)
2.5
>>> scale((5, 3.1),2)
(10, 6.2)
>>> scale((5, 3.1),2, inverse=True)
(2.5, 1.55)
```

CHAPTER

FOUR

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

BIBLIOGRAPHY

[GRS2013] Robert Gibb, Alexander Raichev, Michael Speth, The rHEALPix discrete global grid system, in preparation, 2013.

PYTHON MODULE INDEX

INDEX