healpy Documentation

Release 1.10.3

July 13, 2017

Contents

CHAPTER 1

Tutorial

Healpy tutorial

Creating and manipulating maps

Maps are simply numpy arrays, where each array element refers to a location in the sky as defined by the Healpix pixelization schemes (see the healpix website).

Note: Running the code below in a regular Python session will not display the maps; it's recommended to use IPython:

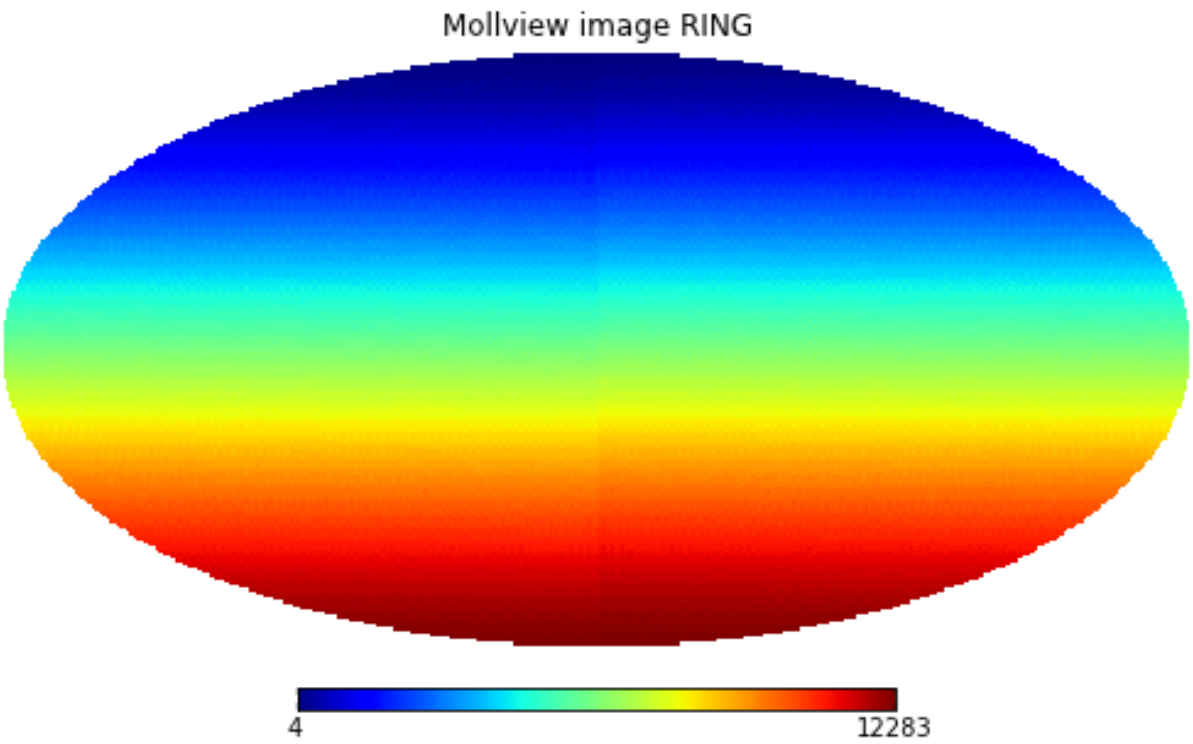
```
% ipython
```

...then select the appropriate backend display:

```
>>> %matplotlib inline # for IPython notebook
>>> %matplotlib qt     # using Qt (e.g. Windows)
>>> %matplotlib osx    # on Macs
>>> %matplotlib gtk    # GTK
```

The resolution of the map is defined by the *NSIDE* parameter. The *nside2npix()* function gives the number of pixel *NPIX* of the map:

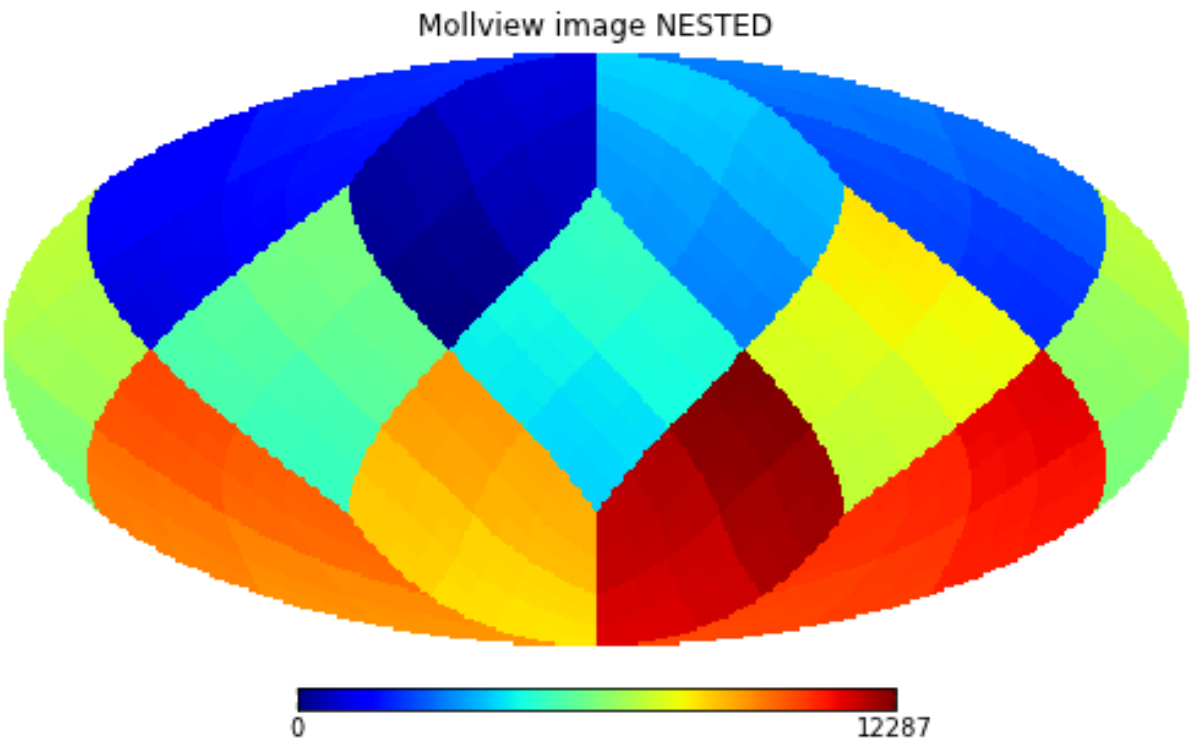
```
>>> import numpy as np
>>> import healpy as hp
>>> NSIDE = 32
>>> m = np.arange(hp.nside2npix(NSIDE))
>>> hp.mollview(m, title="Mollview image RING")
```



Healpix supports two different ordering schemes, *RING* or *NESTED*. By default, healpy maps are in *RING* ordering.

In order to work with *NESTED* ordering, all map related functions support the *nest* keyword, for example:

```
>>> hp.mollview(m, nest=True, title="Mollview image NESTED")
```



Reading and writing maps to file

Maps are read with the *read_map()* function:

```
>>> wmap_map_I = hp.read_map('../healpy/test/data/wmap_band_imap_r9_7yr_W_v4.fits')
```

By default, input maps are converted to *RING* ordering, if they are in *NESTED* ordering. You can otherwise specify *nest=True* to retrieve a map is NESTED ordering, or *nest=None* to keep the ordering unchanged.

By default, *read_map()* loads the first column, for reading other columns you can specify the *field* keyword.

write_map() writes a map to disk in FITS format, if the input map is a list of 3 maps, they are written to a single file as I,Q,U polarization components:

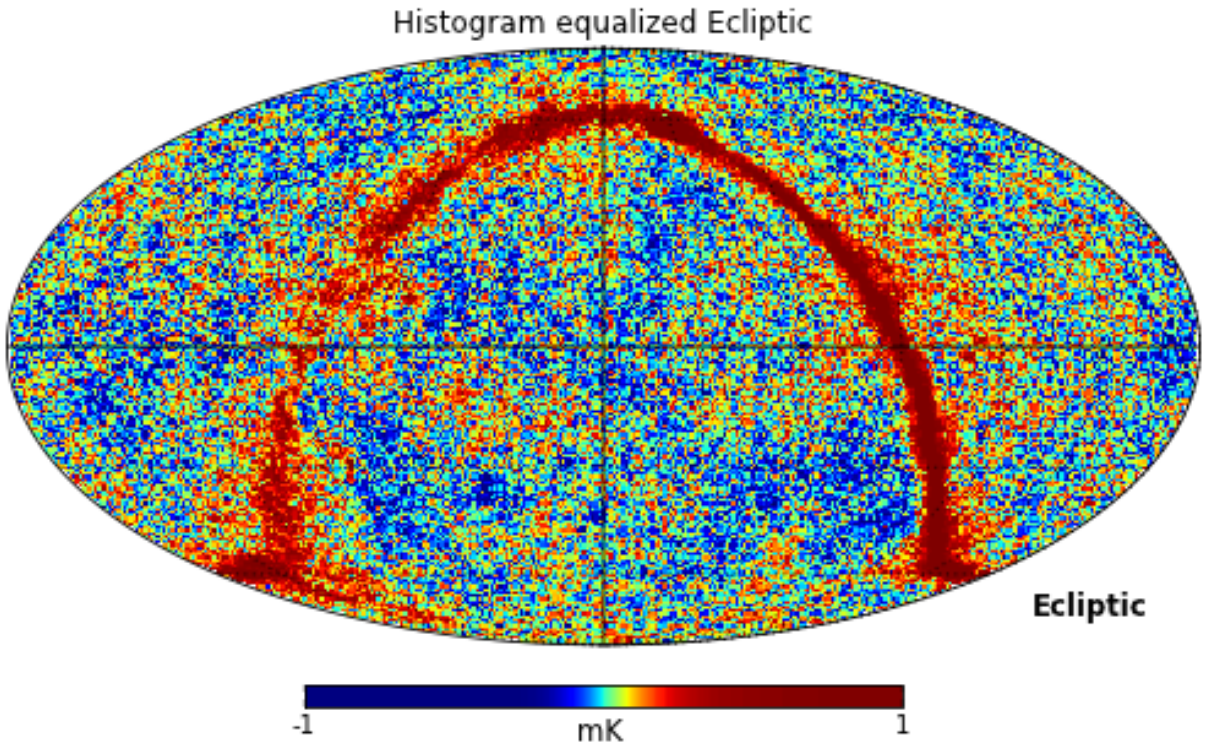
```
>>> hp.write_map("my_map.fits", wmap_map_I)
```

Visualization

Mollweide projection with *mollview()* is the most common visualization tool for HEALPIX maps. It also supports coordinate transformation:

```
>>> hp.mollview(wmap_map_I, coord=['G','E'], title='Histogram equalized Ecliptic',␣
↪unit='mK', norm='hist', min=-1,max=1, xsize=2000)
>>> hp.graticule()
```

coord does galactic to ecliptic coordinate transformation, *norm='hist'* sets a histogram equalized color scale and *xsize* increases the size of the image. *graticule()* adds meridians and parallels.



gnomview() instead provides gnomonic projection around a position specified by *rot*:

```
>>> hp.gnomview(wmap_map_I, rot=[0,0.3], title='GnomView', unit='mK', format='%.2g')
```

shows a projection of the galactic center, *xsize* and *ysize* change the dimension of the sky patch.

mollzoom() is a powerful tool for interactive inspection of a map, it provides a mollweide projection where you can click to set the center of the adjacent gnomview panel.

Masked map, partial maps

By convention, HEALPIX uses -1.6375e+30 to mark invalid or unseen pixels. This is stored in healpy as the constant *UNSEEN()*.

All healpy functions automatically deal with maps with UNSEEN pixels, for example *mollview()* marks in grey that sections of a map.

There is an alternative way of dealing with UNSEEN pixel based on the numpy MaskedArray class, *ma()* loads a map as a masked array:

```
>>> mask = hp.read_map('../healpy/test/data/wmap_temperature_analysis_mask_r9_7yr_v4.
↪fits').astype(np.bool)
>>> wmap_map_I_masked = hp.ma(wmap_map_I)
>>> wmap_map_I_masked.mask = np.logical_not(mask)
```

By convention the mask is 0 where the data are masked, while numpy defines data masked when the mask is True, so it is necessary to flip the mask.

```
>>> hp.mollview(wmap_map_I_masked.filled())
```

filling a masked array fills in the *UNSEEN* value and return a standard array that can be used by *mollview*. *compressed()* instead removes all the masked pixels and returns a standard array that can be used for examples by the matplotlib *hist()* function:

```
>>> import matplotlib.pyplot as plt
>>> plt.hist(wmap_map_I_masked.compressed(), bins = 1000)
```

Spherical harmonic transforms

healpy provides bindings to the C++ HEALPIX library for performing spherical harmonic transforms. *anafast()* computes the angular power spectrum of a map:

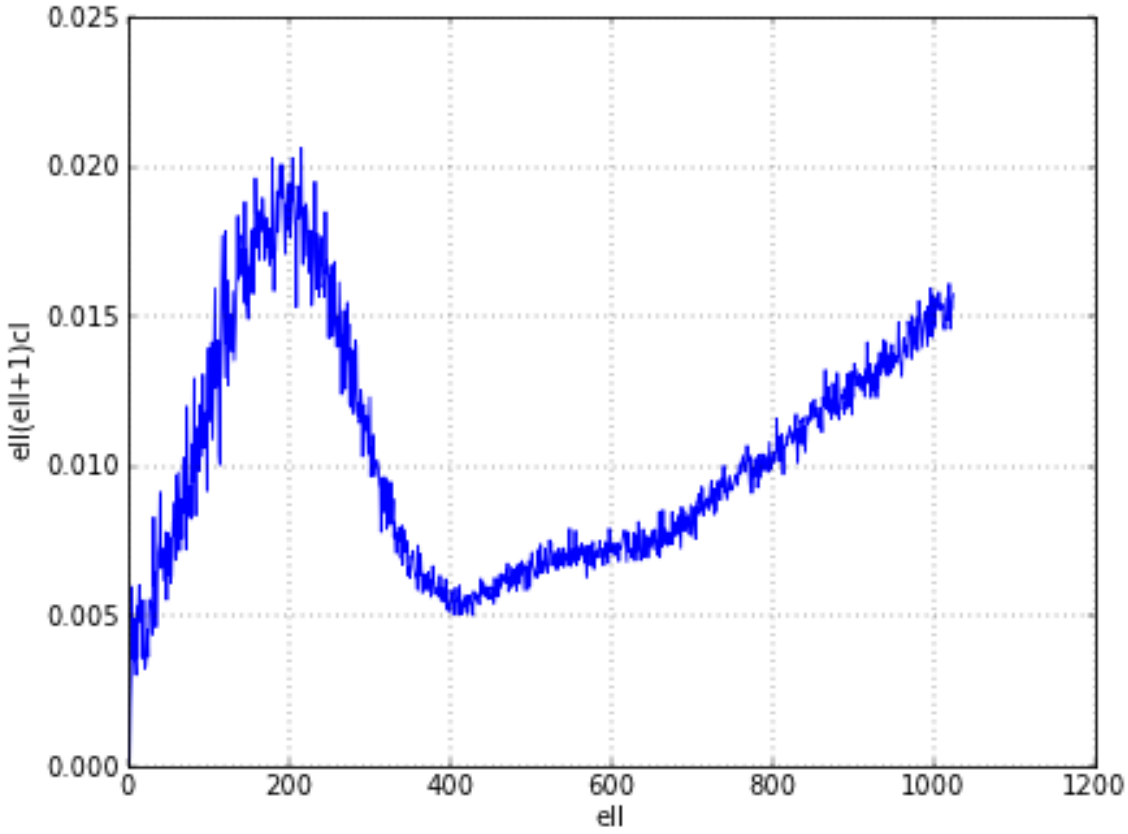
```
>>> LMAX = 1024
>>> cl = hp.anafast(wmap_map_I_masked.filled(), lmax=LMAX)
```

the relative *ell* array is just:

```
>>> ell = np.arange(len(cl))
```

therefore we can plot a normalized CMB spectrum and write it to disk:

```
>>> plt.figure()
>>> plt.plot(ell, ell * (ell+1) * cl)
>>> plt.xlabel('ell'); plt.ylabel('ell(ell+1)cl'); plt.grid()
>>> hp.write_cl('cl.fits', cl)
```



Gaussian beam map smoothing is provided by *smoothing()*:

```
>>> wmap_map_I_smoothed = hp.smoothing(wmap_map_I, fwhm=np.radians(1.))
>>> hp.mollview(wmap_map_I_smoothed, min=-1, max=1, title='Map smoothed 1 deg')
```

CHAPTER 2

Installation

Installation procedure for Healpy

Requirements

Healpy depends on the HEALPix C++ and cfitsio C libraries. Source code for both is included with Healpy and is built automatically, so you do not need to install them yourself. Only Linux and MAC OS X are supported, not Windows.

Binary installation with conda (RECOMMENDED)

Conda forge provides a conda channel with a pre-compiled version of `healpy` for linux 64bit and MAC OS X platforms, you can install it in Anaconda with:

```
conda config --add channels conda-forge
conda install healpy
```

Source installation with Pip

It is possible to build the latest `healpy` with pip

```
pip install --user healpy
```

If you have installed with `pip`, you can keep your installation up to date by upgrading from time to time:

```
pip install --user --upgrade healpy
```

Installation on Mac OS with MacPorts

If you are using a Mac and have the MacPorts package manager, it's even easer to install Healpy with:

```
sudo port install py27-healpy
```

Binary *apt-get* style packages are also available in the development versions of Debian (sid) and Ubuntu (utopic).

Almost-as-quick installation from official source release

Healpy is also available in the Python Package Index (PyPI). You can download it with:

```
curl -O https://pypi.python.org/packages/source/h/healpy/healpy-1.7.4.tar.gz
```

and build it with:

```
tar -xzf healpy-1.7.4.tar.gz
pushd healpy-1.7.4
python setup.py install --user
popd
```

If everything goes fine, you can test it:

```
python
```

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> import healpy as hp
>>> hp.mollview(np.arange(12))
>>> plt.show()
```

or run the test suite with nose:

```
cd healpy-1.7.4 && python setup.py test
```

Building against external Healpix and cfitsio

Healpy uses pkg-config to detect the presence of the Healpix and cfitsio libraries. pkg-config is available on most systems. If you do not have pkg-config installed, then Healpy will download and use (but not install) a Python clone called pykg-config.

If you want to provide your own external builds of Healpix and cfitsio, then download the following packages:

- pkg-config
- HEALPix autotools-style C++ package
- cfitsio

If you are going to install the packages in a nonstandard location (say, `--prefix=/path/to/local`), then you should set the environment variable `PKG_CONFIG_PATH=/path/to/local/lib/pkgconfig` when building. No other environment variable settings are necessary, and you do not need to set `PKG_CONFIG_PATH` to use Healpy after you have built it.

Then, unpack each of the above packages and build them with the usual `configure; make; make install` recipe.

Development install

Developers building from a snapshot of the github repository need:

- `autoconf` and `libtool` (in Debian or Ubuntu: `sudo apt-get install autoconf automake libtool pkg-config`)
- *cython* > 0.16
- run `git submodule init` and `git submodule update` to get the bundled HEALPix sources

the best way to install healpy if you plan to develop is to build the C++ extensions in place with:

```
python setup.py build_ext --inplace
```

then add the `healpy/healpy` folder to your `PYTHONPATH`.

Clean

When you run "python setup.py", temporary build products are placed in the "build" directory. If you want to clean out and remove the `build` directory, then run:

```
python setup.py clean --all
```

CHAPTER 3

Reference

`pixelfunc` – Pixelisation related functions

conversion from/to sky coordinates

pix2ang(nside, ipix[, nest, lonlat])	pix2ang : nside,ipix,nest=False,lonlat=False -> theta[rad],phi[rad] (default RING)
pix2vec(nside, ipix[, nest])	pix2vec : nside,ipix,nest=False -> x,y,z (default RING)
ang2pix(nside, theta, phi[, nest, lonlat])	ang2pix : nside,theta[rad],phi[rad],nest=False,lonlat=False -> ipix (default:RING)
vec2pix(nside, x, y, z[, nest])	vec2pix : nside,x,y,z,nest=False -> ipix (default:RING)
vec2ang(vectors[, lonlat])	vec2ang: vectors [x, y, z] -> theta[rad], phi[rad]
ang2vec(theta, phi[, lonlat])	ang2vec : convert angles to 3D position vector
get_all_neighbours(nside, theta[, phi, ...])	Return the 8 nearest pixels.
get_interp_weights(nside, theta[, phi, ...])	Return the 4 closest pixels on the two rings above and below the location and corresponding weights.
get_interp_val(m, theta, phi[, nest, lonlat])	Return the bi-linear interpolation value of a map using 4 nearest neighbours.

healpy.pixelfunc.pix2ang

healpy.pixelfunc.**pix2ang**(*nside*, *ipix*, *nest=False*, *lonlat=False*)

pix2ang : nside,ipix,nest=False,lonlat=False -> theta[rad],phi[rad] (default RING)

Parameters **nside** : int or array-like

The healpix nside parameter, must be a power of 2, less than 2**30

ipix : int or array-like

Pixel indices

nest : bool, optional

if True, assume NESTED pixel ordering, otherwise, RING pixel ordering

lonlat : bool, optional

If True, return angles will be longitude and latitude in degree, otherwise, angles will be longitude and co-latitude in radians (default)

Returns **theta, phi** : float, scalar or array-like

The angular coordinates corresponding to ipix. Scalar if all input are scalar, array otherwise. Usual numpy broadcasting rules apply.

See also:

ang2pix, *vec2pix*, *pix2vec*

Examples

```
>>> import healpy as hp
>>> hp.pix2ang(16, 1440)
(1.5291175943723188, 0.0)
```

```
>>> hp.pix2ang(16, [1440,  427, 1520,    0, 3068])
(array([ 1.52911759,  0.78550497,  1.57079633,  0.05103658,  3.09055608]),␣
↪array([ 0.        ,  0.78539816,  1.61988371,  0.78539816,  0.78539816]))
```

```
>>> hp.pix2ang([1, 2, 4, 8], 11)
(array([ 2.30052398,  0.84106867,  0.41113786,  0.2044802 ]), array([ 5.49778714,␣
↪ 5.89048623,  5.89048623,  5.89048623]))
```

```
>>> hp.pix2ang([1, 2, 4, 8], 11, lonlat=True)
(array([ 315. ,  337.5,  337.5,  337.5]), array([-41.8103149 ,  41.8103149 ,  66.
↪44353569,  78.28414761]))
```

healpy.pixelfunc.pix2vec

healpy.pixelfunc.**pix2vec**(*nside*, *ipix*, *nest=False*)

pix2vec : nside,ipix,nest=False -> x,y,z (default RING)

Parameters **nside** : int, scalar or array-like

The healpix nside parameter, must be a power of 2, less than 2**30

ipix : int, scalar or array-like

Healpix pixel number

nest : bool, optional

if True, assume NESTED pixel ordering, otherwise, RING pixel ordering

Returns **x, y, z** : floats, scalar or array-like

The coordinates of vector corresponding to input pixels. Scalar if all input are scalar, array otherwise. Usual numpy broadcasting rules apply.

See also:

ang2pix, *pix2ang*, *vec2pix*

Examples

```
>>> import healpy as hp
>>> hp.pix2vec(16, 1504)
(0.99879545620517241, 0.049067674327418015, 0.0)
```

```
>>> hp.pix2vec(16, [1440,  427])
(array([ 0.99913157,  0.5000534 ]), array([ 0.        ,  0.5000534]), array([ 0.
↪04166667,  0.70703125]))
```

```
>>> hp.pix2vec([1, 2], 11)
(array([ 0.52704628,  0.68861915]), array([-0.52704628, -0.28523539]), array([-0.
↪66666667,  0.66666667]))
```

healpy.pixelfunc.ang2pix

healpy.pixelfunc.**ang2pix**(*nside*, *theta*, *phi*, *nest=False*, *lonlat=False*)

ang2pix : nside,theta[rad],phi[rad],nest=False,lonlat=False -> ipix (default:RING)

Parameters **nside** : int, scalar or array-like

The healpix nside parameter, must be a power of 2, less than 2**30

theta, phi : float, scalars or array-like

Angular coordinates of a point on the sphere

nest : bool, optional

if True, assume NESTED pixel ordering, otherwise, RING pixel ordering

lonlat : bool

If True, input angles are assumed to be longitude and latitude in degree, otherwise, they are co-latitude and longitude in radians.

Returns **pix** : int or array of int

The healpix pixel numbers. Scalar if all input are scalar, array otherwise. Usual numpy broadcasting rules apply.

See also:

pix2ang, *pix2vec*, *vec2pix*

Examples

```
>>> import healpy as hp
>>> hp.ang2pix(16, np.pi/2, 0)
1440
```

```
>>> hp.ang2pix(16, [np.pi/2, np.pi/4, np.pi/2, 0, np.pi], [0., np.pi/4, np.pi/2,␣
↪0, 0])
array([1440,  427, 1520,    0, 3068])
```

```
>>> hp.ang2pix(16, np.pi/2, [0, np.pi/2])
array([1440, 1520])
```

```
>>> hp.ang2pix([1, 2, 4, 8, 16], np.pi/2, 0)
array([   4,   12,   72,  336, 1440])
```

```
>>> hp.ang2pix([1, 2, 4, 8, 16], 0, 0, lonlat=True)
array([   4,   12,   72,  336, 1440])
```

healpy.pixelfunc.vec2pix

healpy.pixelfunc.**vec2pix**(*nside*, *x*, *y*, *z*, *nest=False*)

vec2pix : nside,x,y,z,nest=False -> ipix (default:RING)

Parameters **nside** : int or array-like

The healpix nside parameter, must be a power of 2, less than 2**30

x,y,z : floats or array-like

vector coordinates defining point on the sphere

nest : bool, optional

if True, assume NESTED pixel ordering, otherwise, RING pixel ordering

Returns **ipix** : int, scalar or array-like

The healpix pixel number corresponding to input vector. Scalar if all input are scalar, array otherwise. Usual numpy broadcasting rules apply.

See also:

ang2pix, *pix2ang*, *pix2vec*

Examples

```
>>> import healpy as hp
>>> hp.vec2pix(16, 1, 0, 0)
1504
```

```
>>> hp.vec2pix(16, [1, 0], [0, 1], [0, 0])
array([1504, 1520])
```

```
>>> hp.vec2pix([1, 2, 4, 8], 1, 0, 0)
array([  4,  20,  88, 368])
```

healpy.pixelfunc.vec2ang

healpy.pixelfunc.**vec2ang**(*vectors*, *lonlat=False*)

vec2ang: vectors [x, y, z] -> theta[rad], phi[rad]

Parameters **vectors** : float, array-like

the vector(s) to convert, shape is (3,) or (N, 3)

lonlat : bool, optional

If True, return angles will be longitude and latitude in degree, otherwise, angles will be longitude and co-latitude in radians (default)

Returns **theta, phi** : float, tuple of two arrays

the colatitude and longitude in radians

See also:

ang2vec, `rotator.vec2dir`, `rotator.dir2vec`

healpy.pixelfunc.ang2vec

`healpy.pixelfunc.`**`ang2vec`**(*theta*, *phi*, *lonlat=False*)

ang2vec : convert angles to 3D position vector

Parameters **theta** : float, scalar or arry-like

colatitude in radians measured southward from north pole (in [0,pi]).

phi : float, scalar or array-like

longitude in radians measured eastward (in [0, 2*pi]).

lonlat : bool

If True, input angles are assumed to be longitude and latitude in degree, otherwise, they are co-latitude and longitude in radians.

Returns **vec** : float, array

if theta and phi are vectors, the result is a 2D array with a vector per row otherwise, it is a 1D array of shape (3,)

See also:

vec2ang, `rotator.dir2vec`, `rotator.vec2dir`

healpy.pixelfunc.get_all_neighbours

`healpy.pixelfunc.`**`get_all_neighbours`**(*nside*, *theta*, *phi=None*, *nest=False*, *lonlat=False*)

Return the 8 nearest pixels.

Parameters **nside** : int

the nside to work with

theta, phi : scalar or array-like

if phi is not given or None, theta is interpreted as pixel number, otherwise, theta[rad],phi[rad] are angular coordinates

nest : bool

if `True`, pixel number will be NESTED ordering, otherwise RING ordering.

lonlat : bool

If True, input angles are assumed to be longitude and latitude in degree, otherwise, they are co-latitude and longitude in radians.

Returns **ipix** : int, array

pixel number of the SW, W, NW, N, NE, E, SE and S neighbours, shape is (8,) if input is scalar, otherwise shape is (8, N) if input is of length N. If a neighbor does not exist (it can be the case for W, N, E and S) the corresponding pixel number will be -1.

See also:

get_interp_weights, *get_interp_val*

Examples

```
>>> import healpy as hp
>>> hp.get_all_neighbours(1, 4)
array([11,  7,  3, -1,  0,  5,  8, -1])
```

```
>>> hp.get_all_neighbours(1, np.pi/2, np.pi/2)
array([ 8,  4,  0, -1,  1,  6,  9, -1])
```

```
>>> hp.get_all_neighbours(1, 90, 0, lonlat=True)
array([ 8,  4,  0, -1,  1,  6,  9, -1])
```

healpy.pixelfunc.get_interp_weights

healpy.pixelfunc.**get_interp_weights**(*nside*, *theta*, *phi=None*, *nest=False*, *lonlat=False*)

Return the 4 closest pixels on the two rings above and below the location and corresponding weights. Weights are provided for bilinear interpolation along latitude and longitude

Parameters **nside** : int

the healpix nside

theta, phi : float, scalar or array-like

if phi is not given, theta is interpreted as pixel number, otherwise theta[rad],phi[rad] are angular coordinates

nest : bool

if `True`, NESTED ordering, otherwise RING ordering.

lonlat : bool

If True, input angles are assumed to be longitude and latitude in degree, otherwise, they are co-latitude and longitude in radians.

Returns **res** : tuple of length 2

contains pixel numbers in res[0] and weights in res[1]. Usual numpy broadcasting rules apply.

See also:

get_interp_val, *get_all_neighbours*

Examples

```
>>> import healpy as hp
>>> hp.get_interp_weights(1, 0)
(array([0, 1, 4, 5]), array([ 1.,  0.,  0.,  0.]))
```

```
>>> hp.get_interp_weights(1, 0, 0)
(array([1, 2, 3, 0]), array([ 0.25,  0.25,  0.25,  0.25]))
```

```
>>> hp.get_interp_weights(1, 0, 90, lonlat=True)
(array([1, 2, 3, 0]), array([ 0.25,  0.25,  0.25,  0.25]))
```

```
>>> hp.get_interp_weights(1, [0, np.pi/2], 0)
(array([[ 1,  4],
       [ 2,  5],
       [ 3, 11],
       [ 0,  8]]), array([[ 0.25,  1.  ],
       [ 0.25,  0.  ],
       [ 0.25,  0.  ],
       [ 0.25,  0.  ]]))
```

healpy.pixelfunc.get_interp_val

healpy.pixelfunc.**get_interp_val**(*m*, *theta*, *phi*, *nest=False*, *lonlat=False*)

Return the bi-linear interpolation value of a map using 4 nearest neighbours.

Parameters **m** : array-like

an healpix map, accepts masked arrays

theta, phi : float, scalar or array-like

angular coordinates of point at which to interpolate the map

nest : bool

if True, the is assumed to be in NESTED ordering.

lonlat : bool

If True, input angles are assumed to be longitude and latitude in degree, otherwise, they are co-latitude and longitude in radians.

Returns **val** : float, scalar or arry-like

the interpolated value(s), usual numpy broadcasting rules apply.

See also:

get_interp_weights, *get_all_neighbours*

Examples

```
>>> import healpy as hp
>>> hp.get_interp_val(np.arange(12.), np.pi/2, 0)
4.0
>>> hp.get_interp_val(np.arange(12.), np.pi/2, np.pi/2)
5.0
>>> hp.get_interp_val(np.arange(12.), np.pi/2, np.pi/2 + 2*np.pi)
5.0
>>> hp.get_interp_val(np.arange(12.), np.linspace(0, np.pi, 10), 0)
array([ 1.5       ,  1.5       ,  1.5       ,  2.20618428,  3.40206143,
        5.31546486,  7.94639458,  9.5       ,  9.5       ,  9.5       ])
>>> hp.get_interp_val(np.arange(12.), 0, np.linspace(90, -90, 10), lonlat=True)
```

```
array([ 1.5       ,  1.5       ,  1.5       ,  2.20618428,  3.40206143,
        5.31546486,  7.94639458,  9.5       ,  9.5       ,  9.5       ])
```

conversion between NESTED and RING schemes

nest2ring(nside, ipix)	Convert pixel number from NESTED ordering to RING ordering.
ring2nest(nside, ipix)	Convert pixel number from RING ordering to NESTED ordering.
reorder(map_in, *args, **kwds)	Reorder an healpix map from RING/NESTED ordering to NESTED/RING

healpy.pixelfunc.nest2ring

healpy.pixelfunc.**nest2ring**(*nside*, *ipix*)

Convert pixel number from NESTED ordering to RING ordering.

Parameters **nside** : int, scalar or array-like

the healpix nside parameter

ipix : int, scalar or array-like

the pixel number in NESTED scheme

Returns **ipix** : int, scalar or array-like

the pixel number in RING scheme

See also:

ring2nest, *reorder*

Examples

```
>>> import healpy as hp
>>> hp.nest2ring(16, 1130)
1504
```

```
>>> hp.nest2ring(2, np.arange(10))
array([13,  5,  4,  0, 15,  7,  6,  1, 17,  9])
```

```
>>> hp.nest2ring([1, 2, 4, 8], 11)
array([ 11,   2,  12, 211])
```

healpy.pixelfunc.ring2nest

healpy.pixelfunc.**ring2nest**(*nside*, *ipix*)

Convert pixel number from RING ordering to NESTED ordering.

Parameters **nside** : int, scalar or array-like

the healpix nside parameter

ipix : int, scalar or array-like

the pixel number in RING scheme

Returns **ipix** : int, scalar or array-like

the pixel number in NESTED scheme

See also:

nest2ring, *reorder*

Examples

```
>>> import healpy as hp
>>> hp.ring2nest(16, 1504)
1130
```

```
>>> hp.ring2nest(2, np.arange(10))
array([ 3,  7, 11, 15,  2,  1,  6,  5, 10,  9])
```

```
>>> hp.ring2nest([1, 2, 4, 8], 11)
array([ 11,  13,  61, 253])
```

healpy.pixelfunc.reorder

healpy.pixelfunc.**reorder**(*map_in*, **args*, ***kwds*)

Reorder an healpix map from RING/NESTED ordering to NESTED/RING

Parameters **map_in** : array-like

the input map to reorder, accepts masked arrays

inp, out : `'RING'` or `'NESTED'`

define the input and output ordering

r2n : bool

if True, reorder from RING to NESTED

n2r : bool

if True, reorder from NESTED to RING

Returns **map_out** : array-like

the reordered map, as masked array if the input was a masked array

See also:

nest2ring, *ring2nest*

Notes

if `r2n` or `n2r` is defined, override `inp` and `out`.

Examples

```
>>> import healpy as hp
>>> hp.reorder(np.arange(48), r2n = True)
array([13,  5,  4,  0, 15,  7,  6,  1, 17,  9,  8,  2, 19, 11, 10,  3, 28,
       20, 27, 12, 30, 22, 21, 14, 32, 24, 23, 16, 34, 26, 25, 18, 44, 37,
       36, 29, 45, 39, 38, 31, 46, 41, 40, 33, 47, 43, 42, 35])
>>> hp.reorder(np.arange(12), n2r = True)
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
>>> hp.reorder(hp.ma(np.arange(12.)), n2r = True)
masked_array(data = [  0.   1.   2.   3.   4.   5.   6.   7.   8.   9.  10.  11.],
             mask = False,
       fill_value = -1.6375e+30)

>>> m = [np.arange(12.), np.arange(12.), np.arange(12.)]
>>> m[0][2] = hp.UNSEEN
>>> m[1][2] = hp.UNSEEN
>>> m[2][2] = hp.UNSEEN
>>> m = hp.ma(m)
>>> hp.reorder(m, n2r = True)
masked_array(data =
 [[0.0 1.0 -- 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0]
 [0.0 1.0 -- 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0]
 [0.0 1.0 -- 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0]],
             mask =
 [[False False  True False False False False False False False False False]
 [False False  True False False False False False False False False False]
 [False False  True False False False False False False False False False]],
       fill_value = -1.6375e+30)
```

nside/npix/resolution

nside2npix(nside)	Give the number of pixels for the given nside.
npix2nside(npix)	Give the nside parameter for the given number of pixels.
nside2order(nside)	Give the resolution order for a given nside.
order2nside(order)	Give the nside parameter for the given resolution order.
nside2resol(nside[, arcmin])	Give approximate resolution (pixel size in radian or arcmin) for nside.
nside2pixarea(nside[, degrees])	Give pixel area given nside in square radians or square degrees.
max_pixrad(nside)	Maximum angular distance between any pixel center and its corners
isnsideok(nside[, nest])	Returns `True` if nside is a valid nside parameter, `False` otherwise.
isnpixok(npix)	Return `True` if npix is a valid value for healpix map size, `False` otherwise.
get_map_size(m)	Returns the npix of a given map (implicit or explicit pixelization).
get_min_valid_nside(npix)	Returns the minimum acceptable nside so that npix <= nside2npix(nside).
get_nside(m)	Return the nside of the given map.

Continued on next page

Table 3.3 – continued from previous page

maptype(m)	Describe the type of the map (valid, single, sequence of maps).
ud_grade(map_in, *args, **kwds)	Upgrade or degrade resolution of a map (or list of maps).

healpy.pixelfunc.nside2npix

healpy.pixelfunc.**nside2npix**(*nside*)

Give the number of pixels for the given nside.

Parameters **nside** : int

healpix nside parameter; an exception is raised if nside is not valid (nside must be a power of 2, less than 2**30)

Returns **npix** : int

corresponding number of pixels

Notes

Raise a ValueError exception if nside is not valid.

Examples

```
>>> import healpy as hp
>>> import numpy as np
>>> hp.nside2npix(8)
768
```

```
>>> np.all([hp.nside2npix(nside) == 12 * nside**2 for nside in [2**n for n in␣
↪range(12)]])
True
```

```
>>> hp.nside2npix(7)
588
```

healpy.pixelfunc.npix2nside

healpy.pixelfunc.**npix2nside**(*npix*)

Give the nside parameter for the given number of pixels.

Parameters **npix** : int

the number of pixels

Returns **nside** : int

the nside parameter corresponding to npix

Notes

Raise a ValueError exception if number of pixel does not correspond to the number of pixel of an healpix map.

Examples

```
>>> import healpy as hp
>>> hp.npix2nside(768)
8
```

```
>>> np.all([hp.npix2nside(12 * nside**2) == nside for nside in [2**n for n in␣
↪range(12)]])
True
```

```
>>> hp.npix2nside(1000)
Traceback (most recent call last):
    ...
ValueError: Wrong pixel number (it is not 12*nside**2)
```

healpy.pixelfunc.nside2order

healpy.pixelfunc.**nside2order**(*nside*)

Give the resolution order for a given nside.

Parameters **nside** : int

healpix nside parameter; an exception is raised if nside is not valid (nside must be a power of 2, less than 2**30)

Returns **order** : int

corresponding order where nside = 2**(order)

Notes

Raise a ValueError exception if nside is not valid.

Examples

```
>>> import healpy as hp
>>> import numpy as np
>>> hp.nside2order(128)
7
```

```
>>> np.all([hp.nside2order(2**o) == o for o in range(30)])
True
```

```
>>> hp.nside2order(7)
Traceback (most recent call last):
    ...
ValueError: 7 is not a valid nside parameter (must be a power of 2, less than␣
↪2**30)
```

healpy.pixelfunc.order2nside

healpy.pixelfunc.**order2nside**(*order*)

Give the nside parameter for the given resolution order.

Parameters **order** : int

the resolution order

Returns **nside** : int

the nside parameter corresponding to order

Notes

Raise a ValueError exception if order produces an nside out of range.

Examples

```
>>> import healpy as hp
>>> hp.order2nside(7)
128
```

```
>>> hp.order2nside(np.arange(8))
array([  1,   2,   4,   8,  16,  32,  64, 128])
```

```
>>> hp.order2nside(31)
Traceback (most recent call last):
    ...
ValueError: 2147483648 is not a valid nside parameter (must be a power of 2, less␣
↪than 2**30)
```

healpy.pixelfunc.nside2resol

healpy.pixelfunc.**nside2resol**(*nside*, *arcmin=False*)

Give approximate resolution (pixel size in radian or arcmin) for nside.

Resolution is just the square root of the pixel area, which is a gross approximation given the different pixel shapes

Parameters **nside** : int

healpix nside parameter, must be a power of 2, less than 2**30

arcmin : bool

if True, return resolution in arcmin, otherwise in radian

Returns **resol** : float

approximate pixel size in radians or arcmin

Notes

Raise a ValueError exception if nside is not valid.

Examples

```
>>> import healpy as hp
>>> hp.nside2resol(128, arcmin = True)
27.483891294539248
```

```
>>> hp.nside2resol(256)
0.0039973699529159707
```

```
>>> hp.nside2resol(7)
0.1461895297066412
```

healpy.pixelfunc.nside2pixarea

healpy.pixelfunc.**nside2pixarea**(*nside*, *degrees=False*)

Give pixel area given nside in square radians or square degrees.

Parameters **nside** : int

healpix nside parameter, must be a power of 2, less than 2**30

degrees : bool

if True, returns pixel area in square degrees, in square radians otherwise

Returns **pixarea** : float

pixel area in square radian or square degree

Notes

Raise a ValueError exception if nside is not valid.

Examples

```
>>> import healpy as hp
>>> hp.nside2pixarea(128, degrees = True)
0.2098234113027917
```

```
>>> hp.nside2pixarea(256)
1.5978966540475428e-05
```

```
>>> hp.nside2pixarea(7)
0.021371378595848933
```

healpy.pixelfunc.max_pixrad

healpy.pixelfunc.**max_pixrad**(*nside*)

Maximum angular distance between any pixel center and its corners

Parameters **nside** : int

the nside to work with

Returns rads: double

angular distance (in radians)

Examples

```
>>> '%.15f' % max_pixrad(1)
'0.841068670567930'
>>> '%.15f' % max_pixrad(16)
'0.066014761432513'
```

healpy.pixelfunc.isnsideok

healpy.pixelfunc.**isnsideok**(*nside*, *nest=False*)

Returns `True` if nside is a valid nside parameter, `False` otherwise.

NSIDE needs to be a power of 2 only for nested ordering

Parameters **nside** : int, scalar or array-like

integer value to be tested

Returns **ok** : bool, scalar or array-like

`True` if given value is a valid nside, `False` otherwise.

Examples

```
>>> import healpy as hp
>>> hp.isnsideok(13, nest=True)
False
```

```
>>> hp.isnsideok(13, nest=False)
True
```

```
>>> hp.isnsideok(32)
True
```

```
>>> hp.isnsideok([1, 2, 3, 4, 8, 16], nest=True)
array([ True,  True, False,  True,  True,  True], dtype=bool)
```

healpy.pixelfunc.isnpixok

healpy.pixelfunc.**isnpixok**(*npix*)

Return `True` if npix is a valid value for healpix map size, `False` otherwise.

Parameters **npix** : int, scalar or array-like

integer value to be tested

Returns **ok** : bool, scalar or array-like

`True` if given value is a valid number of pixel, `False` otherwise

Examples

```
>>> import healpy as hp
>>> hp.isnpixok(12)
True
```

```
>>> hp.isnpixok(768)
True
```

```
>>> hp.isnpixok([12, 768, 1002])
array([ True,  True, False], dtype=bool)
```

healpy.pixelfunc.get_map_size

healpy.pixelfunc.**get_map_size**(*m*)

Returns the npix of a given map (implicit or explicit pixelization).

If map is a dict type, assumes explicit pixelization: use nside key if present, or use nside attribute if present, otherwise use the smallest valid npix given the maximum key value. otherwise assumes implicit pixelization and returns len(m).

Parameters **m** : array-like or dict-like

a map with implicit (array-like) or explicit (dict-like) pixellization

Returns **npix** : int

a valid number of pixel

Notes

In implicit pixellization, raise a ValueError exception if the size of the input is not a valid pixel number.

Examples

```
>>> import healpy as hp
 >>> m = {0: 1, 1: 1, 2: 1, 'nside': 1}
 >>> print(hp.get_map_size(m))
 12
```

```
>>> m = {0: 1, 767: 1}
>>> print(hp.get_map_size(m))
768
```

```
>>> print(hp.get_map_size(np.zeros(12 * 8 ** 2)))
768
```

healpy.pixelfunc.get_min_valid_nside

healpy.pixelfunc.**get_min_valid_nside**(*npix*)

Returns the minimum acceptable nside so that npix <= nside2npix(nside).

Parameters **npix** : int

a minimal number of pixel

Returns **nside** : int

a valid healpix nside so that 12 * nside ** 2 >= npix

Examples

```
>>> import healpy as hp
>>> hp.pixelfunc.get_min_valid_nside(355)
8
>>> hp.pixelfunc.get_min_valid_nside(768)
8
```

healpy.pixelfunc.get_nside

healpy.pixelfunc.**get_nside**(*m*)

Return the nside of the given map.

Parameters **m** : sequence

the map to get the nside from.

Returns **nside** : int

the healpix nside parameter of the map (or sequence of maps)

Notes

If the input is a sequence of maps, all of them must have same size. If the input is not a valid map (not a sequence, unvalid number of pixels), a TypeError exception is raised.

healpy.pixelfunc.maptype

healpy.pixelfunc.**maptype**(*m*)

Describe the type of the map (valid, single, sequence of maps). Checks : the number of maps, that all maps have same length and that this length is a valid map size (using *isnpixok()*).

Parameters **m** : sequence

the map to get info from

Returns **info** : int

-1 if the given object is not a valid map, 0 if it is a single map, *info* > 0 if it is a sequence of maps (*info* is then the number of maps)

Examples

```
>>> import healpy as hp
>>> hp.pixelfunc.maptype(np.arange(12))
0
>>> hp.pixelfunc.maptype([np.arange(12), np.arange(12)])
2
```

healpy.pixelfunc.ud_grade

healpy.pixelfunc.**ud_grade**(*map_in*, **args*, ***kwds*)

Upgrade or degrade resolution of a map (or list of maps).

in degrading the resolution, ud_grade sets the value of the superpixel as the mean of the children pixels.

Parameters **map_in** : array-like or sequence of array-like

the input map(s) (if a sequence of maps, all must have same size)

nside_out : int

the desired nside of the output map(s)

pess : bool

if `True`, in degrading, reject pixels which contains a bad sub_pixel. Otherwise, estimate average with good pixels

order_in, order_out : str

pixel ordering of input and output ('RING' or 'NESTED')

power : float

if non-zero, divide the result by (nside_in/nside_out)**power Examples: power=-2 keeps the sum of the map invariant (useful for hitmaps), power=2 divides the mean by another factor of (nside_in/nside_out)**2 (useful for variance maps)

dtype : type

the type of the output map

Returns **map_out** : array-like or sequence of array-like

the upgraded or degraded map(s)

Examples

```
>>> import healpy as hp
>>> hp.ud_grade(np.arange(48.), 1)
array([  5.5 ,   7.25,   9.  ,  10.75,  21.75,  21.75,  23.75,  25.75,
        36.5 ,  38.25,  40.  ,  41.75])
```

Masking pixels

UNSEEN	Special value used for masked pixels
mask_bad(m[, badval, rtol, atol])	Returns a bool array with `True` where m is close to badval.

Continued on next page

Table 3.4 – continued from previous page

mask_good(m[, badval, rtol, atol])	Returns a bool array with `False` where m is close to badval.
ma(m[, badval, rtol, atol, copy])	Return map as a masked array, with `badval` pixels masked.

healpy.pixelfunc.UNSEEN

`healpy.pixelfunc.`**`UNSEEN = -1.6375e+30`**

Special value used for masked pixels

healpy.pixelfunc.mask_bad

`healpy.pixelfunc.`**`mask_bad`**(*m*, *badval=-1.6375e+30*, *rtol=1e-05*, *atol=1e-08*)

Returns a bool array with `True` where m is close to badval.

Parameters **m** : a map (may be a sequence of maps)

badval : float, optional

The value of the pixel considered as bad (*UNSEEN* by default)

rtol : float, optional

The relative tolerance

atol : float, optional

The absolute tolerance

Returns mask

a bool array with the same shape as the input map, `True` where input map is close to badval, and `False` elsewhere.

See also:

mask_good, *ma*

Examples

```
>>> import healpy as hp
>>> import numpy as np
>>> m = np.arange(12.)
>>> m[3] = hp.UNSEEN
>>> hp.mask_bad(m)
array([False, False, False,  True, False, False, False, False, False,
       False, False, False], dtype=bool)
```

healpy.pixelfunc.mask_good

`healpy.pixelfunc.`**`mask_good`**(*m*, *badval=-1.6375e+30*, *rtol=1e-05*, *atol=1e-08*)

Returns a bool array with `False` where m is close to badval.

Parameters **m** : a map (may be a sequence of maps)

badval : float, optional

The value of the pixel considered as bad (*UNSEEN* by default)

rtol : float, optional

The relative tolerance

atol : float, optional

The absolute tolerance

Returns a bool array with the same shape as the input map, `False` where input map is close to badval, and `True` elsewhere.

See also:

mask_bad, *ma*

Examples

```
>>> import healpy as hp
>>> m = np.arange(12.)
>>> m[3] = hp.UNSEEN
>>> hp.mask_good(m)
array([ True,  True,  True, False,  True,  True,  True,  True,  True,
        True,  True,  True], dtype=bool)
```

healpy.pixelfunc.ma

`healpy.pixelfunc.`**`ma`**(*m*, *badval=-1.6375e+30*, *rtol=1e-05*, *atol=1e-08*, *copy=True*)

Return map as a masked array, with `badval` pixels masked.

Parameters **m** : a map (may be a sequence of maps)

badval : float, optional

The value of the pixel considered as bad (*UNSEEN* by default)

rtol : float, optional

The relative tolerance

atol : float, optional

The absolute tolerance

copy : bool, optional

If `True`, a copy of the input map is made.

Returns a masked array with the same shape as the input map, masked where input map is close to badval.

See also:

mask_good, *mask_bad*, `numpy.ma.masked_values`

Examples

```
>>> import healpy as hp
>>> m = np.arange(12.)
>>> m[3] = hp.UNSEEN
>>> hp.ma(m)
masked_array(data = [0.0 1.0 2.0 -- 4.0 5.0 6.0 7.0 8.0 9.0 10.0 11.0],
             mask = [False False False  True False False False False False False␣
↪False False],
       fill_value = -1.6375e+30)
```

Map data manipulation

fit_dipole(m[, nest, bad, gal_cut])	Fit a dipole and a monopole to the map, excluding bad pixels.
fit_monopole(m[, nest, bad, gal_cut])	Fit a monopole to the map, excluding unseen pixels.
remove_dipole(m[, nest, bad, gal_cut, ...])	Fit and subtract the dipole and the monopole from the given map m.
remove_monopole(m[, nest, bad, gal_cut, ...])	Fit and subtract the monopole from the given map m.
get_interp_val(m, theta, phi[, nest, lonlat])	Return the bi-linear interpolation value of a map using 4 nearest neighbours.

healpy.pixelfunc.fit_dipole

healpy.pixelfunc.**fit_dipole**(*m*, *nest=False*, *bad=-1.6375e+30*, *gal_cut=0*)

Fit a dipole and a monopole to the map, excluding bad pixels.

Parameters **m** : float, array-like

the map to which a dipole is fitted and subtracted, accepts masked maps

nest : bool

if `False` m is assumed in RING scheme, otherwise map is NESTED

bad : float

bad values of pixel, default to *UNSEEN*.

gal_cut : float

pixels at latitude in [-gal_cut;+gal_cut] degrees are not taken into account

Returns **res** : tuple of length 2

the monopole value in res[0] and the dipole vector (as array) in res[1]

See also:

remove_dipole, *fit_monopole*, *remove_monopole*

healpy.pixelfunc.fit_monopole

healpy.pixelfunc.**fit_monopole**(*m*, *nest=False*, *bad=-1.6375e+30*, *gal_cut=0*)

Fit a monopole to the map, excluding unseen pixels.

Parameters **m** : float, array-like

the map to which a dipole is fitted and subtracted, accepts masked arrays

nest : bool

if `False` m is assumed in RING scheme, otherwise map is NESTED

bad : float

bad values of pixel, default to *UNSEEN*.

gal_cut : float

pixels at latitude in [-gal_cut;+gal_cut] degrees are not taken into account

Returns res: float

fitted monopole value

See also:

fit_dipole, *remove_monopole*, *remove_monopole*

healpy.pixelfunc.remove_dipole

healpy.pixelfunc.**remove_dipole**(*m*, *nest=False*, *bad=-1.6375e+30*, *gal_cut=0*, *fitval=False*, *copy=True*, *verbose=True*)

Fit and subtract the dipole and the monopole from the given map m.

Parameters **m** : float, array-like

the map to which a dipole is fitted and subtracted, accepts masked arrays

nest : bool

if `False` m is assumed in RING scheme, otherwise map is NESTED

bad : float

bad values of pixel, default to *UNSEEN*.

gal_cut : float

pixels at latitude in [-gal_cut;+gal_cut] are not taken into account

fitval : bool

whether to return or not the fitted values of monopole and dipole

copy : bool

whether to modify input map or not (by default, make a copy)

verbose : bool

print values of monopole and dipole

Returns **res** : array or tuple of length 3

if fitval is False, returns map with monopole and dipole subtracted, otherwise, returns map (array, in res[0]), monopole (float, in res[1]), dipole_vector (array, in res[2])

See also:

fit_dipole, *fit_monopole*, *remove_monopole*

healpy.pixelfunc.remove_monopole

healpy.pixelfunc.**remove_monopole**(*m*, *nest=False*, *bad=-1.6375e+30*, *gal_cut=0*, *fitval=False*, *copy=True*, *verbose=True*)

Fit and subtract the monopole from the given map m.

Parameters **m** : float, array-like

the map to which a monopole is fitted and subtracted

nest : bool

if `False` m is assumed in RING scheme, otherwise map is NESTED

bad : float

bad values of pixel, default to *UNSEEN*.

gal_cut : float

pixels at latitude in [-gal_cut;+gal_cut] are not taken into account

fitval : bool

whether to return or not the fitted value of monopole

copy : bool

whether to modify input map or not (by default, make a copy)

verbose: bool

whether to print values of monopole

Returns **res** : array or tuple of length 3

if fitval is False, returns map with monopole subtracted, otherwise, returns map (array, in res[0]) and monopole (float, in res[1])

See also:

fit_dipole, *fit_monopole*, *remove_dipole*

sphtfunc – Spherical harmonic transforms

From map to spherical harmonics

anafast(map1[, map2, nspec, lmax, mmax, ...])	Computes the power spectrum of an Healpix map, or the cross-spectrum between two maps if *map2* is given.
map2alm(maps[, lmax, mmax, iter, pol, ...])	Computes the alm of an Healpix map.

healpy.sphtfunc.anafast

healpy.sphtfunc.**anafast**(*map1*, *map2=None*, *nspec=None*, *lmax=None*, *mmax=None*, *iter=3*, *alm=False*, *pol=True*, *use_weights=False*, *datapath=None*)

Computes the power spectrum of an Healpix map, or the cross-spectrum between two maps if *map2* is given. No removal of monopole or dipole is performed.

Parameters **map1** : float, array-like shape (Npix,) or (3, Npix)

Either an array representing a map, or a sequence of 3 arrays representing I, Q, U maps

map2 : float, array-like shape (Npix,) or (3, Npix)

Either an array representing a map, or a sequence of 3 arrays representing I, Q, U maps

nspec : None or int, optional

The number of spectra to return. If None, returns all, otherwise returns cls[:nspec]

lmax : int, scalar, optional

Maximum l of the power spectrum (default: 3*nside-1)

mmax : int, scalar, optional

Maximum m of the alm (default: lmax)

iter : int, scalar, optional

Number of iteration (default: 3)

alm : bool, scalar, optional

If True, returns both cl and alm, otherwise only cl is returned

pol : bool, optional

If True, assumes input maps are TQU. Output will be TEB cl's and correlations (input must be 1 or 3 maps). If False, maps are assumed to be described by spin 0 spherical harmonics. (input can be any number of maps) If there is only one input map, it has no effect. Default: True.

datapath : None or str, optional

If given, the directory where to find the weights data.

Returns **res** : array or sequence of arrays

If *alm* is False, returns cl or a list of cl's (TT, EE, BB, TE, EB, TB for polarized input map) Otherwise, returns a tuple (cl, alm), where cl is as above and alm is the spherical harmonic transform or a list of almT, almE, almB for polarized input

healpy.sphtfunc.map2alm

`healpy.sphtfunc.`**`map2alm`**(*maps*, *lmax=None*, *mmax=None*, *iter=3*, *pol=True*, *use_weights=False*, *datapath=None*)

Computes the alm of an Healpix map.

Parameters **maps** : array-like, shape (Npix,) or (n, Npix)

The input map or a list of n input maps.

lmax : int, scalar, optional

Maximum l of the power spectrum. Default: 3*nside-1

mmax : int, scalar, optional

Maximum m of the alm. Default: lmax

iter : int, scalar, optional

Number of iteration (default: 3)

pol : bool, optional

If True, assumes input maps are TQU. Output will be TEB alm's. (input must be 1 or 3 maps) If False, apply spin 0 harmonic transform to each map. (input can be any number of maps) If there is only one input map, it has no effect. Default: True.

use_weights: bool, scalar, optional

If True, use the ring weighting. Default: False.

datapath : None or str, optional

If given, the directory where to find the weights data.

Returns **alms** : array or tuple of array

alm or a tuple of 3 alm (almT, almE, almB) if polarized input.

Notes

The pixels which have the special *UNSEEN* value are replaced by zeros before spherical harmonic transform. They are converted back to *UNSEEN* value, so that the input maps are not modified. Each map have its own, independent mask.

From spherical harmonics to map

synfast(cls, nside[, lmax, mmax, alm, pol, ...])	Create a map(s) from cl(s).
alm2map(alms, nside[, lmax, mmax, pixwin, ...])	Computes an Healpix map given the alm.
alm2map_der1(alm, nside[, lmax, mmax])	Computes an Healpix map and its first derivatives given the alm.

healpy.sphtfunc.synfast

healpy.sphtfunc.**synfast**(*cls*, *nside*, *lmax=None*, *mmax=None*, *alm=False*, *pol=True*, *pixwin=False*, *fwhm=0.0*, *sigma=None*, *new=False*, *verbose=True*)

Create a map(s) from cl(s).

Parameters **cls** : array or tuple of array

A cl or a list of cl (either 4 or 6, see *synalm()*)

nside : int, scalar

The nside of the output map(s)

lmax : int, scalar, optional

Maximum l for alm. Default: min of 3*nside-1 or length of the cls - 1

mmax : int, scalar, optional

Maximum m for alm.

alm : bool, scalar, optional

If True, return also alm(s). Default: False.

pol : bool, optional

If True, assumes input cls are TEB and correlation. Output will be TQU maps. (input must be 1, 4 or 6 cl's) If False, fields are assumed to be described by spin 0 spherical

harmonics. (input can be any number of cl's) If there is only one input cl, it has no effect. Default: True.

pixwin : bool, scalar, optional

If True, convolve the alm by the pixel window function. Default: False.

fwhm : float, scalar, optional

The fwhm of the Gaussian used to smooth the map (applied on alm) [in radians]

sigma : float, scalar, optional

The sigma of the Gaussian used to smooth the map (applied on alm) [in radians]

Returns **maps** : array or tuple of arrays

The output map (possibly list of maps if polarized input). or, if alm is True, a tuple of (map,alm) (alm possibly a list of alm if polarized input)

Notes

The order of the spectra will change in a future release. The new= parameter help to make the transition smoother. You can start using the new order by setting new=True. In the next version of healpy, the default will be new=True. This change is done for consistency between the different tools (alm2cl, synfast, anafast). In the new order, the spectra are ordered by diagonal of the correlation matrix. Eg, if fields are T, E, B, the spectra are TT, EE, BB, TE, EB, TB with new=True, and TT, TE, TB, EE, EB, BB if new=False.

healpy.sphtfunc.alm2map

`healpy.sphtfunc.`**`alm2map`**(*alms*, *nside*, *lmax=None*, *mmax=None*, *pixwin=False*, *fwhm=0.0*, *sigma=None*, *pol=True*, *inplace=False*, *verbose=True*)

Computes an Healpix map given the alm.

The alm are given as a complex array. You can specify lmax and mmax, or they will be computed from array size (assuming lmax==mmax).

Parameters **alms** : complex, array or sequence of arrays

A complex array or a sequence of complex arrays. Each array must have a size of the form: mmax * (2 * lmax + 1 - mmax) / 2 + lmax + 1

nside : int, scalar

The nside of the output map.

lmax : None or int, scalar, optional

Explicitly define lmax (needed if mmax!=lmax)

mmax : None or int, scalar, optional

Explicitly define mmax (needed if mmax!=lmax)

pixwin : bool, optional

Smooth the alm using the pixel window functions. Default: False.

fwhm : float, scalar, optional

The fwhm of the Gaussian used to smooth the map (applied on alm) [in radians]

sigma : float, scalar, optional

The sigma of the Gaussian used to smooth the map (applied on alm) [in radians]

pol : bool, optional

If True, assumes input alms are TEB. Output will be TQU maps. (input must be 1 or 3 alms) If False, apply spin 0 harmonic transform to each alm. (input can be any number of alms) If there is only one input alm, it has no effect. Default: True.

inplace : bool, optional

If True, input alms may be modified by pixel window function and beam smoothing (if alm(s) are complex128 contiguous arrays). Otherwise, input alms are not modified. A copy is made if needed to apply beam smoothing or pixel window.

Returns **maps** : array or list of arrays

An Healpix map in RING scheme at nside or a list of T,Q,U maps (if polarized input)

healpy.sphtfunc.alm2map_der1

healpy.sphtfunc.**alm2map_der1**(*alm*, *nside*, *lmax=None*, *mmax=None*)

Computes an Healpix map and its first derivatives given the alm.

The alm are given as a complex array. You can specify lmax and mmax, or they will be computed from array size (assuming lmax==mmax).

Parameters **alm** : array, complex

A complex array of alm. Size must be of the form mmax(lmax-mmax+1)/2+lmax

nside : int

The nside of the output map.

lmax : None or int, optional

Explicitly define lmax (needed if mmax!=lmax)

mmax : None or int, optional

Explicitly define mmax (needed if mmax!=lmax)

Returns **m, d_theta, d_phi** : tuple of arrays

The maps correponding to alm, and its derivatives with respect to theta and phi. d_phi is already divided by sin(theta)

Spherical harmonic transform tools

smoothing(map_in, *args, **kwds)	Smooth a map with a Gaussian symmetric beam.
smoothalm(alms[, fwhm, sigma, pol, mmax, ...])	Smooth alm with a Gaussian symmetric beam function.
alm2cl(alms1[, alms2, lmax, mmax, lmax_out, ...])	Computes (cross-)spectra from alm(s).
synalm(cls[, lmax, mmax, new, verbose])	Generate a set of alm given cl.
almxfl(alm, fl[, mmax, inplace])	Multiply alm by a function of l.
pixwin(nside[, pol])	Return the pixel window function for the given nside.
Alm()	This class provides some static methods for alm index computation.

healpy.sphtfunc.smoothing

healpy.sphtfunc.**smoothing**(*map_in*, **args*, ***kwds*)

Smooth a map with a Gaussian symmetric beam.

No removal of monopole or dipole is performed.

Parameters **map_in** : array or sequence of 3 arrays

Either an array representing one map, or a sequence of 3 arrays representing 3 maps, accepts masked arrays

fwhm : float, optional

The full width half max parameter of the Gaussian [in radians]. Default:0.0

sigma : float, optional

The sigma of the Gaussian [in radians]. Override fwhm.

pol : bool, optional

If True, assumes input maps are TQU. Output will be TQU maps. (input must be 1 or 3 alms) If False, each map is assumed to be a spin 0 map and is treated independently (input can be any number of alms). If there is only one input map, it has no effect. Default: True.

iter : int, scalar, optional

Number of iteration (default: 3)

lmax : int, scalar, optional

Maximum l of the power spectrum. Default: 3*nside-1

mmax : int, scalar, optional

Maximum m of the alm. Default: lmax

use_weights: bool, scalar, optional

If True, use the ring weighting. Default: False.

datapath : None or str, optional

If given, the directory where to find the weights data.

verbose : bool, optional

If True prints diagnostic information. Default: True

Returns **maps** : array or list of 3 arrays

The smoothed map(s)

healpy.sphtfunc.smoothalm

healpy.sphtfunc.**smoothalm**(*alms*, *fwhm=0.0*, *sigma=None*, *pol=True*, *mmax=None*, *verbose=True*, *inplace=True*)

Smooth alm with a Gaussian symmetric beam function.

Parameters **alms** : array or sequence of 3 arrays

Either an array representing one alm, or a sequence of arrays. See *pol* parameter.

fwhm : float, optional

The full width half max parameter of the Gaussian. Default:0.0 [in radians]

sigma : float, optional

The sigma of the Gaussian. Override fwhm. [in radians]

pol : bool, optional

If True, assumes input alms are TEB. Output will be TQU maps. (input must be 1 or 3 alms) If False, apply spin 0 harmonic transform to each alm. (input can be any number of alms) If there is only one input alm, it has no effect. Default: True.

mmax : None or int, optional

The maximum m for alm. Default: mmax=lmax

inplace : bool, optional

If True, the alm's are modified inplace if they are contiguous arrays of type complex128. Otherwise, a copy of alm is made. Default: True.

verbose : bool, optional

If True prints diagnostic information. Default: True

Returns **alms** : array or sequence of 3 arrays

The smoothed alm. If alm[i] is a contiguous array of type complex128, and *inplace* is True the smoothing is applied inplace. Otherwise, a copy is made.

healpy.sphtfunc.alm2cl

healpy.sphtfunc.**alm2cl**(*alms1*, *alms2=None*, *lmax=None*, *mmax=None*, *lmax_out=None*, *nspec=None*)

Computes (cross-)spectra from alm(s). If alm2 is given, cross-spectra between alm and alm2 are computed. If alm (and alm2 if provided) contains n alm, then n(n+1)/2 auto and cross-spectra are returned.

Parameters **alm** : complex, array or sequence of arrays

The alm from which to compute the power spectrum. If n>=2 arrays are given, computes both auto- and cross-spectra.

alms2 : complex, array or sequence of 3 arrays, optional

If provided, computes cross-spectra between alm and alm2. Default: alm2=alm, so auto-spectra are computed.

lmax : None or int, optional

The maximum l of the input alm. Default: computed from size of alm and mmax_in

mmax : None or int, optional

The maximum m of the input alm. Default: assume mmax_in = lmax_in

lmax_out : None or int, optional

The maximum l of the returned spectra. By default: the lmax of the given alm(s).

nspec : None or int, optional

The number of spectra to return. None means all, otherwise returns cl[:nspec]

Returns **cl** : array or tuple of n(n+1)/2 arrays

the spectrum <*alm* x *alm2*> if *alm* (and *alm2*) is one alm, or the auto- and cross-spectra <*alm*[i] x *alm2*[j]> *if alm (and alm2) contains more than one spectra. If more than one spectrum is returned, they are ordered by diagonal. For example, if *alm* is almT, almE, almB, then the returned spectra are: TT, EE, BB, TE, EB, TB.

healpy.sphtfunc.synalm

healpy.sphtfunc.**synalm**(*cls*, *lmax=None*, *mmax=None*, *new=False*, *verbose=True*)

Generate a set of alm given cl. The cl are given as a float array. Corresponding alm are generated. If lmax is None, it is assumed lmax=cl.size-1 If mmax is None, it is assumed mmax=lmax.

Parameters **cls** : float, array or tuple of arrays

Either one cl (1D array) or a tuple of either 4 cl or of n*(n+1)/2 cl. Some of the cl may be None, implying no cross-correlation. See *new* parameter.

lmax : int, scalar, optional

The lmax (if None or <0, the largest size-1 of cls)

mmax : int, scalar, optional

The mmax (if None or <0, =lmax)

new : bool, optional

If True, use the new ordering of cl's, ie by diagonal (e.g. TT, EE, BB, TE, EB, TB or TT, EE, BB, TE if 4 cl as input). If False, use the old ordering, ie by row (e.g. TT, TE, TB, EE, EB, BB or TT, TE, EE, BB if 4 cl as input).

Returns **alms** : array or list of arrays

the generated alm if one spectrum is given, or a list of n alms (with n(n+1)/2 the number of input cl, or n=3 if there are 4 input cl).

Notes

The order of the spectra will change in a future release. The new= parameter help to make the transition smoother. You can start using the new order by setting new=True. In the next version of healpy, the default will be new=True. This change is done for consistency between the different tools (alm2cl, synfast, anafast). In the new order, the spectra are ordered by diagonal of the correlation matrix. Eg, if fields are T, E, B, the spectra are TT, EE, BB, TE, EB, TB with new=True, and TT, TE, TB, EE, EB, BB if new=False.

healpy.sphtfunc.almxfl

healpy.sphtfunc.**almxfl**(*alm*, *fl*, *mmax=None*, *inplace=False*)

Multiply alm by a function of l. The function is assumed to be zero where not defined.

Parameters **alm** : array

The alm to multiply

fl : array

The function (at l=0..fl.size-1) by which alm must be multiplied.

mmax : None or int, optional

The maximum m defining the alm layout. Default: lmax.

inplace : bool, optional

If True, modify the given alm, otherwise make a copy before multiplying.

Returns **alm** : array

The modified alm, either a new array or a reference to input alm, if inplace is True.

healpy.sphtfunc.pixwin

healpy.sphtfunc.**pixwin**(*nside*, *pol=False*)

Return the pixel window function for the given nside.

Parameters **nside** : int

The nside for which to return the pixel window function

pol : bool, optional

If True, return also the polar pixel window. Default: False

Returns **pw or pwT,pwP** : array or tuple of 2 arrays

The temperature pixel window function, or a tuple with both temperature and polarisation pixel window functions.

healpy.sphtfunc.Alm

class healpy.sphtfunc.**Alm**

This class provides some static methods for alm index computation.

Methods

getlm(lmax[, i])	Get the l and m from index and lmax.
getidx(lmax, l, m)	Returns index corresponding to (l,m) in an array describing alm up to lmax.
getsize(lmax[, mmax])	Returns the size of the array needed to store alm up to *lmax* and *mmax*
getlmax(s[, mmax])	Returns the lmax corresponding to a given array size.

healpy.sphtfunc.Alm.getlm

static Alm.**getlm**(*lmax*, *i=None*)

Get the l and m from index and lmax.

Parameters **lmax** : int

The maximum l defining the alm layout

i : int or None

The index for which to compute the l and m. If None, the function return l and m for i=0..Alm.getsize(lmax)

healpy.sphtfunc.Alm.getidx

static Alm.**getidx**(*lmax*, *l*, *m*)
Returns index corresponding to (l,m) in an array describing alm up to lmax.

Parameters **lmax** : int

The maximum l, defines the alm layout

l : int

The l for which to get the index

m : int

The m for which to get the index

Returns **idx** : int

The index corresponding to (l,m)

healpy.sphtfunc.Alm.getsize

static Alm.**getsize**(*lmax*, *mmax=None*)
Returns the size of the array needed to store alm up to *lmax* and *mmax*

Parameters **lmax** : int

The maximum l, defines the alm layout

mmax : int, optional

The maximum m, defines the alm layout. Default: lmax.

Returns **size** : int

The size of the array needed to store alm up to lmax, mmax.

healpy.sphtfunc.Alm.getlmax

static Alm.**getlmax**(*s*, *mmax=None*)
Returns the lmax corresponding to a given array size.

Parameters **s** : int

Size of the array

mmax : None or int, optional

The maximum m, defines the alm layout. Default: lmax.

Returns **lmax** : int

The maximum l of the array, or -1 if it is not a valid size.

Other tools

gauss_beam(fwhm[, lmax, pol])	Gaussian beam window function

healpy.sphtfunc.gauss_beam

healpy.sphtfunc.**gauss_beam**(*fwhm*, *lmax=512*, *pol=False*)

Gaussian beam window function

Computes the spherical transform of an axisimmetric gaussian beam

For a sky of underlying power spectrum C(l) observed with beam of given FWHM, the measured power spectrum will be C(l)_meas = C(l) B(l)^2 where B(l) is given by gaussbeam(Fwhm,Lmax). The polarization beam is also provided (when pol = True) assuming a perfectly co-polarized beam (e.g., Challinor et al 2000, astro-ph/0008228)

Parameters **fwhm** : float

full width half max in radians

lmax : integer

ell max

pol : bool

if False, output has size (lmax+1) and is temperature beam if True output has size (lmax+1, 4) with components: * temperature beam * grad/electric polarization beam * curl/magnetic polarization beam * temperature * grad beam

Returns **beam** : array

beam window function [0, lmax] if dim not specified otherwise (lmax+1, 4) contains polarized beam

visufunc – Visualisation

Map projections

mollview([map, fig, rot, coord, unit, ...])	Plot an healpix map (given as an array) in Mollweide projection.
gnomview([map, fig, rot, coord, unit, ...])	Plot an healpix map (given as an array) in Gnomonic projection.
cartview([map, fig, rot, zat, coord, unit, ...])	Plot an healpix map (given as an array) in Cartesian projection.
orthview([map, fig, rot, coord, unit, ...])	Plot an healpix map (given as an array) in Orthographic projection.

healpy.visufunc.mollview

healpy.visufunc.**mollview**(*map=None*, *fig=None*, *rot=None*, *coord=None*, *unit=''*, *xsize=800*, *title='Mollweide view'*, *nest=False*, *min=None*, *max=None*, *flip='astro'*, *remove_dip=False*, *remove_mono=False*, *gal_cut=0*, *format='%g'*, *format2='%g'*, *cbar=True*, *cmap=None*, *notext=False*, *norm=None*, *hold=False*, *margins=None*, *sub=None*, *return_projected_map=False*)

Plot an healpix map (given as an array) in Mollweide projection.

Parameters **map** : float, array-like or None

An array containing the map, supports masked maps, see the *ma* function. If None, will display a blank map, useful for overplotting.

fig : int or None, optional

The figure number to use. Default: create a new figure

rot : scalar or sequence, optional

Describe the rotation to apply. In the form (lon, lat, psi) (unit: degrees) : the point at longitude *lon* and latitude *lat* will be at the center. An additional rotation of angle *psi* around this direction is applied.

coord : sequence of character, optional

Either one of 'G', 'E' or 'C' to describe the coordinate system of the map, or a sequence of 2 of these to rotate the map from the first to the second coordinate system.

unit : str, optional

A text describing the unit of the data. Default: ''

xsize : int, optional

The size of the image. Default: 800

title : str, optional

The title of the plot. Default: 'Mollweide view'

nest : bool, optional

If True, ordering scheme is NESTED. Default: False (RING)

min : float, optional

The minimum range value

max : float, optional

The maximum range value

flip : {'astro', 'geo'}, optional

Defines the convention of projection : 'astro' (default, east towards left, west towards right) or 'geo' (east towards right, west towards left)

remove_dip : bool, optional

If `True`, remove the dipole+monopole

remove_mono : bool, optional

If `True`, remove the monopole

gal_cut : float, scalar, optional

Symmetric galactic cut for the dipole/monopole fit. Removes points in latitude range [-gal_cut, +gal_cut]

format : str, optional

The format of the scale label. Default: '%g'

format2 : str, optional

Format of the pixel value under mouse. Default: '%g'

cbar : bool, optional

Display the colorbar. Default: True

notext : bool, optional

If True, no text is printed around the map

norm : {'hist', 'log', None}

Color normalization, hist= histogram equalized color mapping, log= logarithmic color mapping, default: None (linear color mapping)

hold : bool, optional

If True, replace the current Axes by a MollweideAxes. use this if you want to have multiple maps on the same figure. Default: False

sub : int, scalar or sequence, optional

Use only a zone of the current figure (same syntax as subplot). Default: None

margins : None or sequence, optional

Either None, or a sequence (left,bottom,right,top) giving the margins on left,bottom,right and top of the axes. Values are relative to figure (0-1). Default: None

return_projected_map : bool

if True returns the projected map in a 2d numpy array

See also:

gnomview, *cartview*, *orthview*, azeqview

healpy.visufunc.gnomview

healpy.visufunc.**gnomview**(*map=None*, *fig=None*, *rot=None*, *coord=None*, *unit=''*, *xsize=200*, *ysize=None*, *reso=1.5*, *title='Gnomonic view'*, *nest=False*, *remove_dip=False*, *remove_mono=False*, *gal_cut=0*, *min=None*, *max=None*, *flip='astro'*, *format='%.3g'*, *cbar=True*, *cmap=None*, *norm=None*, *hold=False*, *sub=None*, *margins=None*, *notext=False*, *return_projected_map=False*)

Plot an healpix map (given as an array) in Gnomonic projection.

Parameters **map** : array-like

The map to project, supports masked maps, see the *ma* function. If None, use a blank map, useful for overplotting.

fig : None or int, optional

A figure number. Default: None= create a new figure

rot : scalar or sequence, optional

Describe the rotation to apply. In the form (lon, lat, psi) (unit: degrees) : the point at longitude *lon* and latitude *lat* will be at the center. An additional rotation of angle *psi* around this direction is applied.

coord : sequence of character, optional

Either one of 'G', 'E' or 'C' to describe the coordinate system of the map, or a sequence of 2 of these to rotate the map from the first to the second coordinate system.

unit : str, optional

A text describing the unit of the data. Default: ''

xsize : int, optional

The size of the image. Default: 200

ysize : None or int, optional

The size of the image. Default: None= xsize

reso : float, optional

Resolution (in arcmin). Default: 1.5 arcmin

title : str, optional

The title of the plot. Default: 'Gnomonic view'

nest : bool, optional

If True, ordering scheme is NESTED. Default: False (RING)

min : float, scalar, optional

The minimum range value

max : float, scalar, optional

The maximum range value

flip : {'astro', 'geo'}, optional

Defines the convention of projection : 'astro' (default, east towards left, west towards right) or 'geo' (east towards roght, west towards left)

remove_dip : bool, optional

If `True`, remove the dipole+monopole

remove_mono : bool, optional

If `True`, remove the monopole

gal_cut : float, scalar, optional

Symmetric galactic cut for the dipole/monopole fit. Removes points in latitude range [-gal_cut, +gal_cut]

format : str, optional

The format of the scale label. Default: '%g'

hold : bool, optional

If True, replace the current Axes by a MollweideAxes. use this if you want to have multiple maps on the same figure. Default: False

sub : int or sequence, optional

Use only a zone of the current figure (same syntax as subplot). Default: None

margins : None or sequence, optional

Either None, or a sequence (left,bottom,right,top) giving the margins on left,bottom,right and top of the axes. Values are relative to figure (0-1). Default: None

notext: bool, optional

If True: do not add resolution info text. Default=False

return_projected_map : bool

if True returns the projected map in a 2d numpy array

See also:

mollview, *cartview*, *orthview*, azeqview

healpy.visufunc.cartview

healpy.visufunc.**cartview**(*map=None*, *fig=None*, *rot=None*, *zat=None*, *coord=None*, *unit=''*, *xsize=800*, *ysize=None*, *lonra=None*, *latra=None*, *title='Cartesian view'*, *nest=False*, *remove_dip=False*, *remove_mono=False*, *gal_cut=0*, *min=None*, *max=None*, *flip='astro'*, *format='%.3g'*, *cbar=True*, *cmap=None*, *norm=None*, *aspect=None*, *hold=False*, *sub=None*, *margins=None*, *notext=False*, *return_projected_map=False*)

Plot an healpix map (given as an array) in Cartesian projection.

Parameters **map** : float, array-like or None

An array containing the map, supports masked maps, see the *ma* function. If None, will display a blank map, useful for overplotting.

fig : int or None, optional

The figure number to use. Default: create a new figure

rot : scalar or sequence, optional

Describe the rotation to apply. In the form (lon, lat, psi) (unit: degrees) : the point at longitude *lon* and latitude *lat* will be at the center. An additional rotation of angle *psi* around this direction is applied.

coord : sequence of character, optional

Either one of 'G', 'E' or 'C' to describe the coordinate system of the map, or a sequence of 2 of these to rotate the map from the first to the second coordinate system.

unit : str, optional

A text describing the unit of the data. Default: ''

xsize : int, optional

The size of the image. Default: 800

lonra : sequence, optional

Range in longitude. Default: [-180,180]

latra : sequence, optional

Range in latitude. Default: [-90,90]

title : str, optional

The title of the plot. Default: 'Mollweide view'

nest : bool, optional

If True, ordering scheme is NESTED. Default: False (RING)

min : float, optional

The minimum range value

max : float, optional

The maximum range value

flip : {'astro', 'geo'}, optional

Defines the convention of projection : 'astro' (default, east towards left, west towards right) or 'geo' (east towards roght, west towards left)

remove_dip : bool, optional

If `True`, remove the dipole+monopole

remove_mono : bool, optional

If `True`, remove the monopole

gal_cut : float, scalar, optional

Symmetric galactic cut for the dipole/monopole fit. Removes points in latitude range [-gal_cut, +gal_cut]

format : str, optional

The format of the scale label. Default: '%g'

cbar : bool, optional

Display the colorbar. Default: True

notext : bool, optional

If True, no text is printed around the map

norm : {'hist', 'log', None}, optional

Color normalization, hist= histogram equalized color mapping, log= logarithmic color mapping, default: None (linear color mapping)

hold : bool, optional

If True, replace the current Axes by a CartesianAxes. use this if you want to have multiple maps on the same figure. Default: False

sub : int, scalar or sequence, optional

Use only a zone of the current figure (same syntax as subplot). Default: None

margins : None or sequence, optional

Either None, or a sequence (left,bottom,right,top) giving the margins on left,bottom,right and top of the axes. Values are relative to figure (0-1). Default: None

return_projected_map : bool

if True returns the projected map in a 2d numpy array

See also:

mollview, *gnomview*, *orthview*, `azeqview`

healpy.visufunc.orthview

healpy.visufunc.**orthview**(*map=None*, *fig=None*, *rot=None*, *coord=None*, *unit=''*, *xsize=800*, *half_sky=False*, *title='Orthographic view'*, *nest=False*, *min=None*, *max=None*, *flip='astro'*, *remove_dip=False*, *remove_mono=False*, *gal_cut=0*, *format='%g'*, *format2='%g'*, *cbar=True*, *cmap=None*, *notext=False*, *norm=None*, *hold=False*, *margins=None*, *sub=None*, *return_projected_map=False*)

Plot an healpix map (given as an array) in Orthographic projection.

Parameters **map** : float, array-like or None

An array containing the map. If None, will display a blank map, useful for overplotting.

fig : int or None, optional

The figure number to use. Default: create a new figure

rot : scalar or sequence, optional

Describe the rotation to apply. In the form (lon, lat, psi) (unit: degrees) : the point at longitude *lon* and latitude *lat* will be at the center. An additional rotation of angle *psi* around this direction is applied.

coord : sequence of character, optional

Either one of 'G', 'E' or 'C' to describe the coordinate system of the map, or a sequence of 2 of these to rotate the map from the first to the second coordinate system.

half_sky : bool, optional

Plot only one side of the sphere. Default: False

unit : str, optional

A text describing the unit of the data. Default: ''

xsize : int, optional

The size of the image. Default: 800

title : str, optional

The title of the plot. Default: 'Orthographic view'

nest : bool, optional

If True, ordering scheme is NESTED. Default: False (RING)

min : float, optional

The minimum range value

max : float, optional

The maximum range value

flip : {'astro', 'geo'}, optional

Defines the convention of projection : 'astro' (default, east towards left, west towards right) or 'geo' (east towards roght, west towards left)

remove_dip : bool, optional

If `True`, remove the dipole+monopole

remove_mono : bool, optional

If `True`, remove the monopole

gal_cut : float, scalar, optional

Symmetric galactic cut for the dipole/monopole fit. Removes points in latitude range [-gal_cut, +gal_cut]

format : str, optional

The format of the scale label. Default: '%g'

format2 : str, optional

Format of the pixel value under mouse. Default: '%g'

cbar : bool, optional

Display the colorbar. Default: True

notext : bool, optional

If True, no text is printed around the map

norm : {'hist', 'log', None}

Color normalization, hist= histogram equalized color mapping, log= logarithmic color mapping, default: None (linear color mapping)

hold : bool, optional

If True, replace the current Axes by an OrthographicAxes. use this if you want to have multiple maps on the same figure. Default: False

sub : int, scalar or sequence, optional

Use only a zone of the current figure (same syntax as subplot). Default: None

margins : None or sequence, optional

Either None, or a sequence (left,bottom,right,top) giving the margins on left,bottom,right and top of the axes. Values are relative to figure (0-1). Default: None

return_projected_map : bool

if True returns the projected map in a 2d numpy array

See also:

mollview, *gnomview*, *cartview*, azeqview

Graticules

graticule([dpar, dmer, coord, local])	Draw a graticule on the current Axes.
delgraticules()	Delete all graticules previously created on the Axes.

healpy.visufunc.graticule

healpy.visufunc.**graticule**(*dpar=None*, *dmer=None*, *coord=None*, *local=None*, ***kwds*)

Draw a graticule on the current Axes.

Parameters dpar, dmer : float, scalars

Interval in degrees between meridians and between parallels

coord : {'E', 'G', 'C'}

The coordinate system of the graticule (make rotation if needed, using coordinate system of the map if it is defined).

local : bool

If True, draw a local graticule (no rotation is performed, useful for a gnomonic view, for example)

See also:

delgraticules

Notes

Other keyword parameters will be transmitted to the projplot function.

healpy.visufunc.delgraticules

`healpy.visufunc.`**`delgraticules`**`()`

Delete all graticules previously created on the Axes.

See also:

graticule

Tracing lines or points

projplot(*args, **kwds)	projplot is a wrapper around `matplotlib.Axes.plot()` to take into account the
projscatter(*args, **kwds)	Projscatter is a wrapper around `matplotlib.Axes.scatter()` to take into account the spherical projection.
projtext(*args, **kwds)	Projtext is a wrapper around `matplotlib.Axes.text()` to take into account the spherical projection.

healpy.visufunc.projplot

`healpy.visufunc.`**`projplot`**`(`*args*, ***kwds*`)`

projplot is a wrapper around `matplotlib.Axes.plot()` to take into account the spherical projection.

You can call this function as:

```
projplot(theta, phi)        # plot a line going through points at coord (theta,␣
↪phi)
projplot(theta, phi, 'bo')  # plot 'o' in blue at coord (theta, phi)
projplot(thetaphi)          # plot a line going through points at coord␣
↪(thetaphi[0], thetaphi[1])
projplot(thetaphi, 'bx')    # idem but with blue 'x'
```

Parameters **theta, phi** : float, array-like

Coordinates of point to plot. Can be put into one 2-d array, first line is then *theta* and second line is *phi*. See *lonlat* parameter for unit.

fmt : str

A format string (see `matplotlib.Axes.plot()` for details)

lonlat : bool, optional

If True, theta and phi are interpreted as longitude and latitude in degree, otherwise, as colatitude and longitude in radian

coord : {'E', 'G', 'C', None}

The coordinate system of the points, only used if the coordinate coordinate system of the Axes has been defined and in this case, a rotation is performed

rot : None or sequence

rotation to be applied =(lon, lat, psi) : lon, lat will be position of the new Z axis, and psi is rotation around this axis, all in degree. if None, no rotation is performed

direct : bool

if True, the rotation to center the projection is not taken into account

See also:

projscatter, *projtext*

Notes

Other keywords are passed to `matplotlib.Axes.plot()`.

healpy.visufunc.projscatter

`healpy.visufunc.`**`projscatter`**(**args*, ***kwds*)

Projscatter is a wrapper around `matplotlib.Axes.scatter()` to take into account the spherical projection.

You can call this function as:

```
projscatter(theta, phi)     # plot points at coord (theta, phi)
projplot(thetaphi)          # plot points at coord (thetaphi[0], thetaphi[1])
```

Parameters **theta, phi** : float, array-like

Coordinates of point to plot. Can be put into one 2-d array, first line is then *theta* and second line is *phi*. See *lonlat* parameter for unit.

lonlat : bool, optional

If True, theta and phi are interpreted as longitude and latitude in degree, otherwise, as colatitude and longitude in radian

coord : {'E', 'G', 'C', None}, optional

The coordinate system of the points, only used if the coordinate coordinate system of the axes has been defined and in this case, a rotation is performed

rot : None or sequence, optional

rotation to be applied =(lon, lat, psi) : lon, lat will be position of the new Z axis, and psi is rotation around this axis, all in degree. if None, no rotation is performed

direct : bool, optional

if True, the rotation to center the projection is not taken into account

See also:

projplot, *projtext*

Notes

Other keywords are passed to `matplotlib.Axes.plot()`.

healpy.visufunc.projtext

`healpy.visufunc.`**`projtext`**`(`*args*, ***kwds*`)`

Projtext is a wrapper around `matplotlib.Axes.text()` to take into account the spherical projection.

Parameters **theta, phi** : float, array-like

Coordinates of point to plot. Can be put into one 2-d array, first line is then *theta* and second line is *phi*. See *lonlat* parameter for unit.

text : str

The text to be displayed.

lonlat : bool, optional

If True, theta and phi are interpreted as longitude and latitude in degree, otherwise, as colatitude and longitude in radian

coord : {'E', 'G', 'C', None}, optional

The coordinate system of the points, only used if the coordinate coordinate system of the axes has been defined and in this case, a rotation is performed

rot : None or sequence, optional

rotation to be applied =(lon, lat, psi) : lon, lat will be position of the new Z axis, and psi is rotation around this axis, all in degree. if None, no rotation is performed

direct : bool, optional

if True, the rotation to center the projection is not taken into account

See also:

projplot, *projscatter*

Notes

Other keywords are passed to `matplotlib.Axes.text()`.

`fitsfunc` – FITS file related functions

Reading/writing maps

read_map(filename[, field, dtype, nest, ...])	Read an healpix map from a fits file.
write_map(filename, m[, nest, dtype, ...])	Writes an healpix map into an healpix file.

healpy.fitsfunc.read_map

healpy.fitsfunc.**read_map**(*filename*, *field=0*, *dtype=<type 'numpy.float64'>*, *nest=False*, *partial=False*, *hdu=1*, *h=False*, *verbose=True*, *memmap=False*)

Read an healpix map from a fits file. Partial-sky files, if properly identified, are expanded to full size and filled with UNSEEN.

Parameters **filename** : str or HDU or HDUList

the fits file name

field : int or tuple of int, or None, optional

The column to read. Default: 0. By convention 0 is temperature, 1 is Q, 2 is U. Field can be a tuple to read multiple columns (0,1,2) If the fits file is a partial-sky file, field=0 corresponds to the first column after the pixel index column. If None, all columns are read in.

dtype : data type or list of data types, optional

Force the conversion to some type. Passing a list allows different types for each field. In that case, the length of the list must correspond to the length of the field parameter. Default: np.float64

nest : bool, optional

If True return the map in NEST ordering, otherwise in RING ordering; use fits keyword ORDERING to decide whether conversion is needed or not If None, no conversion is performed.

partial : bool, optional

If True, fits file is assumed to be a partial-sky file with explicit indexing, if the indexing scheme cannot be determined from the header. If False, implicit indexing is assumed. Default: False. A partial sky file is one in which OBJECT=PARTIAL and INDXSCHM=EXPLICIT, and the first column is then assumed to contain pixel indices. A full sky file is one in which OBJECT=FULLSKY and INDXSCHM=IMPLICIT. At least one of these keywords must be set for the indexing scheme to be properly identified.

hdu : int, optional

the header number to look at (start at 0)

h : bool, optional

If True, return also the header. Default: False.

verbose : bool, optional

If True, print a number of diagnostic messages

memmap : bool, optional

Argument passed to astropy.io.fits.open, if True, the map is not read into memory, but only the required pixels are read when needed. Default: False.

Returns **m | (m0, m1, ...) [, header]** : array or a tuple of arrays, optionally with header appended

The map(s) read from the file, and the header if *h* is True.

healpy.fitsfunc.write_map

healpy.fitsfunc.**write_map**(*filename*, *m*, *nest=False*, *dtype=<type 'numpy.float32'>*, *fits_IDL=True*, *coord=None*, *partial=False*, *column_names=None*, *column_units=None*, *extra_header=()*, *overwrite=False*)

Writes an healpix map into an healpix file.

Parameters **filename** : str

the fits file name

m : array or sequence of 3 arrays

the map to write. Possibly a sequence of 3 maps of same size. They will be considered as I, Q, U maps. Supports masked maps, see the *ma* function.

nest : bool, optional

If True, ordering scheme is assumed to be NESTED, otherwise, RING. Default: RING. The map ordering is not modified by this function, the input map array should already be in the desired ordering (run *ud_grade* beforehand).

fits_IDL : bool, optional

If True, reshapes columns in rows of 1024, otherwise all the data will go in one column. Default: True

coord : str

The coordinate system, typically 'E' for Ecliptic, 'G' for Galactic or 'C' for Celestial (equatorial)

partial : bool, optional

If True, fits file is written as a partial-sky file with explicit indexing. Otherwise, implicit indexing is used. Default: False.

column_names : str or list

Column name or list of column names, if None we use: I_STOKES for 1 component, I/Q/U_STOKES for 3 components, II, IQ, IU, QQ, QU, UU for 6 components, COLUMN_0, COLUMN_1... otherwise

column_units : str or list

Units for each column, or same units for all columns.

extra_header : list

Extra records to add to FITS header.

dtype: np.dtype or list of np.dtypes, optional

The datatype in which the columns will be stored. Will be converted internally from the numpy datatype to the fits convention. If a list, the length must correspond to the number of map arrays. Default: np.float32.

overwrite : bool, optional

If True, existing file is silently overwritten. Otherwise trying to write an existing file raises an OSError (IOError for Python 2).

Reading/writing alm

read_alm(filename[, hdu, return_mmax])	Read alm from a fits file.
write_alm(filename, alms[, out_dtype, lmax, ...])	Write alms to a fits file.

healpy.fitsfunc.read_alm

healpy.fitsfunc.**read_alm**(*filename*, *hdu=1*, *return_mmax=False*)

Read alm from a fits file.

In the fits file, the alm are written with explicit index scheme, index = l**2+l+m+1, while healpix cxx uses index = m*(2*lmax+1-m)/2+l. The conversion is done in this function.

Parameters **filename** : str or HDUList or HDU

The name of the fits file to read

hdu : int, or tuple of int, optional

The header to read. Start at 0. Default: hdu=1

return_mmax : bool, optional

If true, both the alms and mmax is returned in a tuple. Default: return_mmax=False

Returns **alms[, mmax]** : complex array or tuple of a complex array and an int

The alms read from the file and optionally mmax read from the file

healpy.fitsfunc.write_alm

healpy.fitsfunc.**write_alm**(*filename*, *alms*, *out_dtype=None*, *lmax=-1*, *mmax=-1*, *mmax_in=-1*, *overwrite=False*)

Write alms to a fits file.

In the fits file the alms are written with explicit index scheme, index = l*l + l + m +1, possibly out of order. By default write_alm makes a table with the same precision as the alms. If specified, the lmax and mmax parameters truncate the input data to include only alms for which l <= lmax and m <= mmax.

Parameters **filename** : str

The filename of the output fits file

alms : array, complex or list of arrays

A complex ndarray holding the alms, index = m*(2*lmax+1-m)/2+l, see Alm.getidx

lmax : int, optional

The maximum l in the output file

mmax : int, optional

The maximum m in the output file

out_dtype : data type, optional

data type in the output file (must be a numpy dtype). Default: *alms*.real.dtype

mmax_in : int, optional

maximum m in the input array

Reading/writing cl

read_cl(filename[, dtype, h])	Reads Cl from an healpix file, as IDL fits2cl.
write_cl(filename, cl[, dtype, overwrite])	Writes Cl into an healpix file, as IDL cl2fits.

healpy.fitsfunc.read_cl

healpy.fitsfunc.**read_cl**(*filename*, *dtype=<type 'numpy.float64'>*, *h=False*)

Reads Cl from an healpix file, as IDL fits2cl.

Parameters **filename** : str or HDUList or HDU

the fits file name

dtype : data type, optional

the data type of the returned array

Returns **cl** : array

the cl array

healpy.fitsfunc.write_cl

healpy.fitsfunc.**write_cl**(*filename*, *cl*, *dtype=<type 'numpy.float64'>*, *overwrite=False*)

Writes Cl into an healpix file, as IDL cl2fits.

Parameters **filename** : str

the fits file name

cl : array

the cl array to write to file

overwrite : bool, optional

If True, existing file is silently overwritten. Otherwise trying to write an existing file raises an OSError (IOError for Python 2).

Reading/writing column in fits file

mrdfits(filename[, hdu])	Read a table in a fits file.
mwrfits(filename, data[, hdu, colnames, keys])	Write columns to a fits file in a table extension.

healpy.fitsfunc.mrdfits

healpy.fitsfunc.**mrdfits**(*filename*, *hdu=1*)

Read a table in a fits file.

Parameters **filename** : str or HDUList or HDU

The name of the fits file to read, or an HDUList or HDU instance.

hdu : int, optional

The header to read. Start at 0. Default: hdu=1

Returns **cols** : a list of arrays

A list of column data in the given header

healpy.fitsfunc.mwrfits

healpy.fitsfunc.**mwrfits**(*filename*, *data*, *hdu=1*, *colnames=None*, *keys=None*)

Write columns to a fits file in a table extension.

Parameters **filename** : str

The fits file name

data : list of 1D arrays

A list of 1D arrays to write in the table

hdu : int, optional

The header where to write the data. Default: 1

colnames : list of str

The column names

keys : dict-like

A dictionary with keywords to write in the header

Helper

getformat(t)	Get the FITS convention format string of data type t.

healpy.fitsfunc.getformat

healpy.fitsfunc.**getformat**(*t*)

Get the FITS convention format string of data type t.

Parameters **t** : data type

The data type for which the FITS type is requested

Returns **fits_type** : str or None

The FITS string code describing the data type, or None if unknown type.

Pixel querying routines

query_disc(nside, vec, radius[, inclusive, ...])	Returns pixels whose centers lie within the disk defined by *vec* and *radius* (in radians) (if *inclusive* is False), or which overlap with this disk (if *inclusive* is True).
query_polygon(nside, vertices[, inclusive, ...])	Returns the pixels whose centers lie within the convex polygon defined by the *vertices* array (if *inclusive* is False), or which overlap with this polygon (if *inclusive* is True).
query_strip(nside, theta1, theta2[, ...])	Returns pixels whose centers lie within the colatitude range defined by *theta1* and *theta2* (if inclusive is False), or which overlap with this region (if *inclusive* is True).

Continued on next page

Table 3.19 – continued from previous page

boundaries(nside, pix[, step, nest])	Returns an array containing vectors to the boundary of the nominated pixel.

healpy.query_disc

healpy.**query_disc**(*nside*, *vec*, *radius*, *inclusive=False*, *fact=4*, *nest=False*, *ndarray buff=None*)

Returns pixels whose centers lie within the disk defined by *vec* and *radius* (in radians) (if *inclusive* is False), or which overlap with this disk (if *inclusive* is True).

Parameters **nside** : int

The nside of the Healpix map.

vec : float, sequence of 3 elements

The coordinates of unit vector defining the disk center.

radius : float

The radius (in radians) of the disk

inclusive : bool, optional

If False, return the exact set of pixels whose pixel centers lie within the disk; if True, return all pixels that overlap with the disk, and maybe a few more. Default: False

fact : int, optional

Only used when inclusive=True. The overlapping test will be done at the resolution fact*nside. For NESTED ordering, fact must be a power of 2, less than 2**30, else it can be any positive integer. Default: 4.

nest: bool, optional

if True, assume NESTED pixel ordering, otherwise, RING pixel ordering

buff: int array, optional

if provided, this numpy array is used to contain the return values and must be at least long enough to do so

Returns **ipix** : int, array

The pixels which lie within the given disk.

healpy.query_polygon

healpy.**query_polygon**(*nside*, *vertices*, *inclusive=False*, *fact=4*, *nest=False*, *ndarray buff=None*)

Returns the pixels whose centers lie within the convex polygon defined by the *vertices* array (if *inclusive* is False), or which overlap with this polygon (if *inclusive* is True).

Parameters **nside** : int

The nside of the Healpix map.

vertices : float, array-like

Vertex array containing the vertices of the polygon, shape (N, 3).

inclusive : bool, optional

If False, return the exact set of pixels whose pixel centers lie within the polygon; if True, return all pixels that overlap with the polygon, and maybe a few more. Default: False.

fact : int, optional

Only used when inclusive=True. The overlapping test will be done at the resolution fact*nside. For NESTED ordering, fact must be a power of 2, less than 2**30, else it can be any positive integer. Default: 4.

nest: bool, optional

if True, assume NESTED pixel ordering, otherwise, RING pixel ordering

buff: int array, optional

if provided, this numpy array is used to contain the return values and must be at least long enough to do so

Returns **ipix** : int, array

The pixels which lie within the given polygon.

healpy.query_strip

healpy.**query_strip**(*nside*, *theta1*, *theta2*, *inclusive=False*, *nest=False*, *ndarray buff=None*)

Returns pixels whose centers lie within the colatitude range defined by *theta1* and *theta2* (if inclusive is False), or which overlap with this region (if *inclusive* is True). If theta1<theta2, the region between both angles is considered, otherwise the regions 0<theta<theta2 and theta1<theta<pi.

Parameters **nside** : int

The nside of the Healpix map.

theta1 : float

First colatitude (radians)

theta2 : float

Second colatitude (radians)

inclusive ; bool

If False, return the exact set of pixels whose pixels centers lie within the region; if True, return all pixels that overlap with the region.

nest: bool, optional

if True, assume NESTED pixel ordering, otherwise, RING pixel ordering

buff: int array, optional

if provided, this numpy array is used to contain the return values and must be at least long enough to do so

Returns **ipix** : int, array

The pixels which lie within the given strip.

healpy.boundaries

healpy.**boundaries**(*nside*, *pix*, *step=1*, *nest=False*)

Returns an array containing vectors to the boundary of the nominated pixel.

The returned array has shape (3, 4*step), the elements of which are the x,y,z positions on the unit sphere of the pixel boundary. In order to get vector positions for just the corners, specify step=1.

Parameters **nside** : int

The nside of the Healpix map.

pix : int

Pixel identifier

step : int, optional

Number of elements for each side of the pixel.

nest : bool, optional

if True, assume NESTED pixel ordering, otherwise, RING pixel ordering

Returns **boundary** : float, array

x,y,z for positions on the boundary of the pixel

rotator – Rotation and geometry functions

Rotation

Rotator([rot, coord, inv, deg, eulertype])	Rotation operator, including astronomical coordinate systems.
rotateVector(rotmat, vec[, vy, vz, do_rot])	Rotate a vector (or a list of vectors) using the rotation matrix given as first argument.
rotateDirection(rotmat, theta[, phi, ...])	Rotate the vector described by angles theta,phi using the rotation matrix given as first argument.

healpy.rotator.Rotator

class healpy.rotator.**Rotator**(*rot=None*, *coord=None*, *inv=None*, *deg=True*, *eulertype='ZYX'*)

Rotation operator, including astronomical coordinate systems.

This class provides tools for spherical rotations. It is meant to be used in the healpy library for plotting, and for this reason reflects the convention used in the Healpix IDL library.

Parameters **rot** : None or sequence

Describe the rotation by its euler angle. See `euler_matrix_new()`.

coord : None or sequence of str

Describe the coordinate system transform. If *rot* is also given, the coordinate transform is applied first, and then the rotation.

inv : bool

If True, the inverse rotation is defined. (Default: False)

deg : bool

If True, angles are assumed to be in degree. (Default: True)

eulertype : str

The Euler angle convention used. See `euler_matrix_new()`.

Examples

```
>>> r = Rotator(coord=['G','E'])  # Transforms galactic to ecliptic coordinates
>>> theta_gal, phi_gal = np.pi/2., 0.
>>> theta_ecl, phi_ecl = r(theta_gal, phi_gal)  # Apply the conversion
>>> print(theta_ecl)
1.66742286715
>>> print(phi_ecl)
-1.62596400306
>>> theta_ecl, phi_ecl = Rotator(coord='ge')(theta_gal, phi_gal) # In one line
>>> print(theta_ecl)
1.66742286715
>>> print(phi_ecl)
-1.62596400306
>>> vec_gal = np.array([1, 0, 0]) #Using vectors
>>> vec_ecl = r(vec_gal)
>>> print(vec_ecl)
[-0.05488249 -0.99382103 -0.09647625]
```

Attributes

mat	The matrix representing the rotation.
coordin	The input coordinate system.
coordout	The output coordinate system.
coordinstr	The input coordinate system in str.
coordoutstr	The output coordinate system in str.
rots	The sequence of rots defining the rotation.
coords	The sequence of coords defining the rotation.

healpy.rotator.Rotator.mat

Rotator.**mat**
The matrix representing the rotation.

healpy.rotator.Rotator.coordin

Rotator.**coordin**
The input coordinate system.

healpy.rotator.Rotator.coordout

Rotator.**coordout**
The output coordinate system.

healpy.rotator.Rotator.coordinstr

Rotator.**coordinstr**
The input coordinate system in str.

healpy.rotator.Rotator.coordoutstr

Rotator.**coordoutstr**
The output coordinate system in str.

healpy.rotator.Rotator.rots

Rotator.**rots**
The sequence of rots defining the rotation.

healpy.rotator.Rotator.coords

Rotator.**coords**
The sequence of coords defining the rotation.

Methods

I(*args, **kwds)	Rotate the given vector or direction using the inverse matrix.
__call__(*args, **kwds)	Use the rotator to rotate either spherical coordinates (theta, phi) or a vector (x,y,z).
angle_ref(*args, **kwds)	Compute the angle between transverse reference direction of initial and final frames
do_rot(i)	Returns True if rotation is not (close to) identity.
get_inverse()	

healpy.rotator.Rotator.I

Rotator.**I**(**args*, ***kwds*)
Rotate the given vector or direction using the inverse matrix. rot.I(vec) <==> rot(vec,inv=True)

healpy.rotator.Rotator.__call__

Rotator.**__call__**(**args*, ***kwds*)
Use the rotator to rotate either spherical coordinates (theta, phi) or a vector (x,y,z). You can use lonla key-

word to use longitude, latitude (in degree) instead of theta, phi (in radian). In this case, returns longitude, latitude in degree.

Accepted forms:

r(x,y,z) # x,y,z either scalars or arrays r(theta,phi) # theta, phi scalars or arrays r(lon,lat,lonlat=True) # lon, lat scalars or arrays r(vec) # vec 1-D array with 3 elements, or 2-D array 3xN r(direction) # direction 1-D array with 2 elements, or 2xN array

Parameters **vec_or_dir** : array or multiple arrays

The direction to rotate. See above for accepted formats.

lonlat : bool, optional

If True, assumes the input direction is longitude/latitude in degrees. Otherwise, assumes co-latitude/longitude in radians. Default: False

inv : bool, optional

If True, applies the inverse rotation. Default: False.

healpy.rotator.Rotator.angle_ref

Rotator.**angle_ref**(**args*, ***kwds*)

Compute the angle between transverse reference direction of initial and final frames

For example, if angle of polarisation is psi in initial frame, it will be psi+angle_ref in final frame.

Parameters **dir_or_vec** : array

Direction or vector (see Rotator.__call__)

lonlat: bool, optional

If True, assume input is longitude,latitude in degrees. Otherwise, theta,phi in radian. Default: False

inv : bool, optional

If True, use the inverse transforms. Default: False

Returns **angle** : float, scalar or array

Angle in radian (a scalar or an array if input is a sequence of direction/vector)

healpy.rotator.Rotator.do_rot

Rotator.**do_rot**(*i*)

Returns True if rotation is not (close to) identity.

healpy.rotator.Rotator.get_inverse

Rotator.**get_inverse**()

healpy.rotator.rotateVector

healpy.rotator.**rotateVector**(*rotmat*, *vec*, *vy=None*, *vz=None*, *do_rot=True*)

Rotate a vector (or a list of vectors) using the rotation matrix given as first argument.

Parameters **rotmat** : float, array-like shape (3,3)

The rotation matrix

vec : float, scalar or array-like

The vector to transform (shape (3,) or (3,N)), or x component (scalar or shape (N,)) if vy and vz are given

vy : float, scalar or array-like, optional

The y component of the vector (scalar or shape (N,))

vz : float, scalar or array-like, optional

The z component of the vector (scalar or shape (N,))

do_rot : bool, optional

if True, really perform the operation, if False do nothing.

Returns **vec** : float, array

The component of the rotated vector(s).

See also:

Rotator

healpy.rotator.rotateDirection

healpy.rotator.**rotateDirection**(*rotmat*, *theta*, *phi=None*, *do_rot=True*, *lonlat=False*)

Rotate the vector described by angles theta,phi using the rotation matrix given as first argument.

Parameters **rotmat** : float, array-like shape (3,3)

The rotation matrix

theta : float, scalar or array-like

The angle theta (scalar or shape (N,)) or both angles (scalar or shape (2, N)) if phi is not given.

phi : float, scalar or array-like, optionnal

The angle phi (scalar or shape (N,)).

do_rot : bool, optional

if True, really perform the operation, if False do nothing.

lonlat : bool

If True, input angles are assumed to be longitude and latitude in degree, otherwise, they are co-latitude and longitude in radians.

Returns **angles** : float, array

The angles of describing the rotated vector(s).

See also:

Rotator

Geometrical helpers

vec2dir(vec[, vy, vz, lonlat])	Transform a vector to angle given by theta,phi.
dir2vec(theta[, phi, lonlat])	Transform a direction theta,phi to a unit vector.
angdist(dir1, dir2[, lonlat])	Returns the angular distance between dir1 and dir2.

healpy.rotator.vec2dir

healpy.rotator.**vec2dir**(*vec*, *vy=None*, *vz=None*, *lonlat=False*)

Transform a vector to angle given by theta,phi.

Parameters **vec** : float, scalar or array-like

The vector to transform (shape (3,) or (3,N)), or x component (scalar or shape (N,)) if vy and vz are given

vy : float, scalar or array-like, optional

The y component of the vector (scalar or shape (N,))

vz : float, scalar or array-like, optional

The z component of the vector (scalar or shape (N,))

lonlat : bool, optional

If True, return angles will be longitude and latitude in degree, otherwise, angles will be longitude and co-latitude in radians (default)

Returns **angles** : float, array

The angles (unit depending on *lonlat*) in an array of shape (2,) (if scalar input) or (2, N)

See also:

dir2vec(), pixelfunc.ang2vec(), pixelfunc.vec2ang()

healpy.rotator.dir2vec

healpy.rotator.**dir2vec**(*theta*, *phi=None*, *lonlat=False*)

Transform a direction theta,phi to a unit vector.

Parameters **theta** : float, scalar or array-like

The angle theta (scalar or shape (N,)) or both angles (scalar or shape (2, N)) if phi is not given.

phi : float, scalar or array-like, optionnal

The angle phi (scalar or shape (N,)).

lonlat : bool

If True, input angles are assumed to be longitude and latitude in degree, otherwise, they are co-latitude and longitude in radians.

Returns **vec** : array

The vector(s) corresponding to given angles, shape is (3,) or (3, N).

See also:

vec2dir(), pixelfunc.ang2vec(), pixelfunc.vec2ang()

healpy.rotator.angdist

healpy.rotator.**angdist**(*dir1*, *dir2*, *lonlat=False*)

Returns the angular distance between dir1 and dir2.

Parameters **dir1, dir2** : float, array-like

The directions between which computing the angular distance. Angular if len(dir) == 2 or vector if len(dir) == 3. See *lonlat* for unit

lonlat : bool, scalar or sequence

If True, angles are assumed to be longitude and latitude in degree, otherwise they are interpreted as colatitude and longitude in radian. If a sequence, lonlat[0] applies to dir1 and lonlat[1] applies to dir2.

Returns **angles** : float, scalar or array-like

The angle(s) between dir1 and dir2 in radian.

Examples

```
>>> import healpy as hp
>>> hp.rotator.angdist([.2,0], [.2, 1e-6])
array([  1.98669331e-07])
```

projector – Spherical projections

Basic classes

SphericalProj([rot, coord, flipconv])	This class defines functions for spherical projection.
GnomonicProj([rot, coord, xsize, ysize, reso])	This class provides class methods for Gnomonic projection.
MollweideProj([rot, coord, xsize])	This class provides class methods for Mollweide projection.
CartesianProj([rot, coord, xsize, ysize, ...])	This class provides class methods for Cartesian projection.

healpy.projector.SphericalProj

class healpy.projector.**SphericalProj**(*rot=None*, *coord=None*, *flipconv=None*, ***kwds*)

This class defines functions for spherical projection.

This class contains class method for spherical projection computation. It should not be instantiated. It should be inherited from and methods should be overloaded for desired projection.

Attributes

arrayinfo	Dictionary with information on the projection array

healpy.projector.SphericalProj.arrayinfo

SphericalProj.**arrayinfo**
Dictionary with information on the projection array

Methods

ang2xy(theta[, phi, lonlat, direct])	From angular direction to position in the projection plane (%s).
get_center([lonlat])	Get the center of the projection.
get_extent()	Get the extension of the projection plane.
get_fov()	Get the field of view in degree of the plane of projection
get_proj_plane_info()	
ij2xy([i, j])	From image array indices to position in projection plane (%s).
mkcoord(coord)	
projmap(map, vec2pix_func[, rot, coord])	Create an array containing the projection of the map.
set_flip(flipconv)	flipconv is either 'astro' or 'geo'. None will be default.
set_proj_plane_info(**kwds)	
vec2xy(vx[, vy, vz, direct])	From unit vector direction to position in the projection plane (%s).
xy2ang(x[, y, lonlat, direct])	From position in the projection plane to angular direction (%s).
xy2ij(x[, y])	From position in the projection plane to image array index (%s).
xy2vec(x[, y, direct])	From position in the projection plane to unit vector direction (%s).

healpy.projector.SphericalProj.ang2xy

SphericalProj.**ang2xy**(*theta*, *phi=None*, *lonlat=False*, *direct=False*)
From angular direction to position in the projection plane (%s).

Input:

- theta: if phi is None, theta[0] contains theta, theta[1] contains phi
- phi : if phi is not None, theta,phi are direction
- lonlat: if True, angle are assumed in degree, and longitude, latitude
- flipconv is either 'astro' or 'geo'. None will be default.

Return:

- x, y: position in %s plane.

healpy.projector.SphericalProj.get_center

SphericalProj.**get_center**(*lonlat=False*)
Get the center of the projection.

Input:

- **lonlat** [if True, will return longitude and latitude in degree,] otherwise, theta and phi in radian

Return:

- theta,phi or lonlat depending on lonlat keyword

healpy.projector.SphericalProj.get_extent

SphericalProj.**get_extent**()
Get the extension of the projection plane.

Return: extent = (left,right,bottom,top)

healpy.projector.SphericalProj.get_fov

SphericalProj.**get_fov**()
Get the field of view in degree of the plane of projection

Return: fov: the diameter in radian of the field of view

healpy.projector.SphericalProj.get_proj_plane_info

SphericalProj.**get_proj_plane_info**()

healpy.projector.SphericalProj.ij2xy

SphericalProj.**ij2xy**(*i=None*, *j=None*)
From image array indices to position in projection plane (%s).

Input:

- if i and j are None, generate arrays of i and j as input
- i : if j is None, i[0], j[1] define array indices in %s image.
- j : if defined, i,j define array indices in image.
- projinfo : additional projection information.

Return:

- x,y : position in projection plane.

healpy.projector.SphericalProj.mkcoord

SphericalProj.**mkcoord**(*coord*)

healpy.projector.SphericalProj.projmap

SphericalProj.**projmap**(*map*, *vec2pix_func*, *rot=None*, *coord=None*)

Create an array containing the projection of the map.

Input:

- vec2pix_func: a function taking theta,phi and returning pixel number
- **map: an array containing the spherical map to project,** the pixelisation is described by vec2pix_func

Return:

- a 2D array with the projection of the map.

Note: the Projector must contain information on the array.

healpy.projector.SphericalProj.set_flip

SphericalProj.**set_flip**(*flipconv*)

flipconv is either 'astro' or 'geo'. None will be default.

With 'astro', east is toward left and west toward right. It is the opposite for 'geo'

healpy.projector.SphericalProj.set_proj_plane_info

SphericalProj.**set_proj_plane_info**(***kwds*)

healpy.projector.SphericalProj.vec2xy

SphericalProj.**vec2xy**(*vx*, *vy=None*, *vz=None*, *direct=False*)

From unit vector direction to position in the projection plane (%s).

Input:

- vx: if vy and vz are None, vx[0],vx[1],vx[2] defines the unit vector.
- vy,vz: if defined, vx,vy,vz define the unit vector
- lonlat: if True, angle are assumed in degree, and longitude, latitude
- flipconv is either 'astro' or 'geo'. None will be default.

Return:

- x, y: position in %s plane.

healpy.projector.SphericalProj.xy2ang

SphericalProj.**xy2ang**(*x*, *y=None*, *lonlat=False*, *direct=False*)

From position in the projection plane to angular direction (%s).

Input:

- x : if y is None, x[0], x[1] define the position in %s plane.
- y : if defined, x,y define the position in projection plane.

- lonlat: if True, angle are assumed in degree, and longitude, latitude
- flipconv is either 'astro' or 'geo'. None will be default.

Return:

- theta, phi : angular direction.

healpy.projector.SphericalProj.xy2ij

SphericalProj.**xy2ij**(*x*, *y=None*)

From position in the projection plane to image array index (%s).

Input:

- x : if y is None, x[0], x[1] define the position in %s plane.
- y : if defined, x,y define the position in projection plane.
- projinfo : additional projection information.

Return:

- i,j : image array indices.

healpy.projector.SphericalProj.xy2vec

SphericalProj.**xy2vec**(*x*, *y=None*, *direct=False*)

From position in the projection plane to unit vector direction (%s).

Input:

- x : if y is None, x[0], x[1] define the position in %s plane.
- y : if defined, x,y define the position in projection plane.
- lonlat: if True, angle are assumed in degree, and longitude, latitude
- flipconv is either 'astro' or 'geo'. None will be default.

Return:

- theta, phi : angular direction.

healpy.projector.GnomonicProj

class healpy.projector.**GnomonicProj**(*rot=None*, *coord=None*, *xsize=None*, *ysize=None*, *reso=None*, ***kwds*)

This class provides class methods for Gnomonic projection.

Attributes

arrayinfo	Dictionary with information on the projection array

healpy.projector.GnomonicProj.arrayinfo

GnomonicProj.**arrayinfo**
Dictionary with information on the projection array

Methods

ang2xy(theta[, phi, lonlat, direct])	From angular direction to position in the projection plane (Gnomonic).
get_center([lonlat])	Get the center of the projection.
get_extent()	
get_fov()	
get_proj_plane_info()	
ij2xy([i, j])	From image array indices to position in projection plane (Gnomonic).
mkcoord(coord)	
projmap(map, vec2pix_func[, rot, coord])	Create an array containing the projection of the map.
set_flip(flipconv)	flipconv is either 'astro' or 'geo'. None will be default.
set_proj_plane_info([xsize, ysize, reso])	
vec2xy(vx[, vy, vz, direct])	From angular direction to position in the projection plane (Gnomonic).
xy2ang(x[, y, lonlat, direct])	From position in the projection plane to angular direction (Gnomonic).
xy2ij(x[, y])	From position in the projection plane to image array index (Gnomonic).
xy2vec(x[, y, direct])	From position in the projection plane to unit vector direction (Gnomonic).

healpy.projector.GnomonicProj.ang2xy

GnomonicProj.**ang2xy**(*theta*, *phi=None*, *lonlat=False*, *direct=False*)
From angular direction to position in the projection plane (Gnomonic).

Input:

- theta: if phi is None, theta[0] contains theta, theta[1] contains phi
- phi : if phi is not None, theta,phi are direction
- lonlat: if True, angle are assumed in degree, and longitude, latitude
- flipconv is either 'astro' or 'geo'. None will be default.

Return:

- x, y: position in Gnomonic plane.

healpy.projector.GnomonicProj.get_center

GnomonicProj.**get_center**(*lonlat=False*)
Get the center of the projection.

Input:

- **lonlat** [if True, will return longitude and latitude in degree,] otherwise, theta and phi in radian

Return:

- theta,phi or lonlat depending on lonlat keyword

healpy.projector.GnomonicProj.get_extent

GnomonicProj.**get_extent**()

healpy.projector.GnomonicProj.get_fov

GnomonicProj.**get_fov**()

healpy.projector.GnomonicProj.get_proj_plane_info

GnomonicProj.**get_proj_plane_info**()

healpy.projector.GnomonicProj.ij2xy

GnomonicProj.**ij2xy**(*i=None*, *j=None*)

From image array indices to position in projection plane (Gnomonic).

Input:

- if i and j are None, generate arrays of i and j as input
- i : if j is None, i[0], j[1] define array indices in Gnomonic image.
- j : if defined, i,j define array indices in image.
- projinfo : additional projection information.

Return:

- x,y : position in projection plane.

healpy.projector.GnomonicProj.mkcoord

GnomonicProj.**mkcoord**(*coord*)

healpy.projector.GnomonicProj.projmap

GnomonicProj.**projmap**(*map*, *vec2pix_func*, *rot=None*, *coord=None*)

Create an array containing the projection of the map.

Input:

- vec2pix_func: a function taking theta,phi and returning pixel number
- **map: an array containing the spherical map to project,** the pixelisation is described by vec2pix_func

Return:

- a 2D array with the projection of the map.

Note: the Projector must contain information on the array.

healpy.projector.GnomonicProj.set_flip

GnomonicProj.**set_flip**(*flipconv*)
flipconv is either 'astro' or 'geo'. None will be default.

With 'astro', east is toward left and west toward right. It is the opposite for 'geo'

healpy.projector.GnomonicProj.set_proj_plane_info

GnomonicProj.**set_proj_plane_info**(*xsize=200*, *ysize=None*, *reso=1.5*)

healpy.projector.GnomonicProj.vec2xy

GnomonicProj.**vec2xy**(*vx*, *vy=None*, *vz=None*, *direct=False*)
From angular direction to position in the projection plane (Gnomonic).

Input:

- theta: if phi is None, theta[0] contains theta, theta[1] contains phi
- phi : if phi is not None, theta,phi are direction
- lonlat: if True, angle are assumed in degree, and longitude, latitude
- flipconv is either 'astro' or 'geo'. None will be default.

Return:

- x, y: position in Gnomonic plane.

healpy.projector.GnomonicProj.xy2ang

GnomonicProj.**xy2ang**(*x*, *y=None*, *lonlat=False*, *direct=False*)
From position in the projection plane to angular direction (Gnomonic).

Input:

- x : if y is None, x[0], x[1] define the position in Gnomonic plane.
- y : if defined, x,y define the position in projection plane.
- lonlat: if True, angle are assumed in degree, and longitude, latitude
- flipconv is either 'astro' or 'geo'. None will be default.

Return:

- theta, phi : angular direction.

healpy.projector.GnomonicProj.xy2ij

GnomonicProj.**xy2ij**(*x*, *y=None*)
From position in the projection plane to image array index (Gnomonic).

Input:

- x : if y is None, x[0], x[1] define the position in Gnomonic plane.
- y : if defined, x,y define the position in projection plane.
- projinfo : additional projection information.

Return:

- i,j : image array indices.

healpy.projector.GnomonicProj.xy2vec

GnomonicProj.**xy2vec**(*x*, *y=None*, *direct=False*)
From position in the projection plane to unit vector direction (Gnomonic).

Input:

- x : if y is None, x[0], x[1] define the position in Gnomonic plane.
- y : if defined, x,y define the position in projection plane.
- lonlat: if True, angle are assumed in degree, and longitude, latitude
- flipconv is either 'astro' or 'geo'. None will be default.

Return:

- theta, phi : angular direction.

healpy.projector.MollweideProj

class healpy.projector.**MollweideProj**(*rot=None*, *coord=None*, *xsize=800*, ***kwds*)
This class provides class methods for Mollweide projection.

Attributes

arrayinfo	Dictionary with information on the projection array

healpy.projector.MollweideProj.arrayinfo

MollweideProj.**arrayinfo**
Dictionary with information on the projection array

Methods

ang2xy(theta[, phi, lonlat, direct])	From angular direction to position in the projection plane (Mollweide).
get_center([lonlat])	Get the center of the projection.
get_extent()	
get_fov()	Get the field of view in degree of the plane of projection
get_proj_plane_info()	
ij2xy([i, j])	From image array indices to position in projection plane (Mollweide).
mkcoord(coord)	
projmap(map, vec2pix_func[, rot, coord])	Create an array containing the projection of the map.
set_flip(flipconv)	flipconv is either 'astro' or 'geo'. None will be default.
set_proj_plane_info(xsize)	
vec2xy(vx[, vy, vz, direct])	From unit vector direction to position in the projection plane (Mollweide).
xy2ang(x[, y, lonlat, direct])	From position in the projection plane to angular direction (Mollweide).
xy2ij(x[, y])	From position in the projection plane to image array index (Mollweide).
xy2vec(x[, y, direct])	From position in the projection plane to unit vector direction (Mollweide).

healpy.projector.MollweideProj.ang2xy

MollweideProj.**ang2xy**(*theta*, *phi=None*, *lonlat=False*, *direct=False*)

From angular direction to position in the projection plane (Mollweide).

Input:

- theta: if phi is None, theta[0] contains theta, theta[1] contains phi
- phi : if phi is not None, theta,phi are direction
- lonlat: if True, angle are assumed in degree, and longitude, latitude
- flipconv is either 'astro' or 'geo'. None will be default.

Return:

- x, y: position in Mollweide plane.

healpy.projector.MollweideProj.get_center

MollweideProj.**get_center**(*lonlat=False*)

Get the center of the projection.

Input:

- **lonlat** [if True, will return longitude and latitude in degree,] otherwise, theta and phi in radian

Return:

- theta,phi or lonlat depending on lonlat keyword

healpy.projector.MollweideProj.get_extent

MollweideProj.**get_extent**()

healpy.projector.MollweideProj.get_fov

MollweideProj.**get_fov**()
Get the field of view in degree of the plane of projection

Return: fov: the diameter in radian of the field of view

healpy.projector.MollweideProj.get_proj_plane_info

MollweideProj.**get_proj_plane_info**()

healpy.projector.MollweideProj.ij2xy

MollweideProj.**ij2xy**(*i=None*, *j=None*)
From image array indices to position in projection plane (Mollweide).

Input:

- if i and j are None, generate arrays of i and j as input
- i : if j is None, i[0], j[1] define array indices in Mollweide image.
- j : if defined, i,j define array indices in image.
- projinfo : additional projection information.

Return:

- x,y : position in projection plane.

healpy.projector.MollweideProj.mkcoord

MollweideProj.**mkcoord**(*coord*)

healpy.projector.MollweideProj.projmap

MollweideProj.**projmap**(*map*, *vec2pix_func*, *rot=None*, *coord=None*)
Create an array containing the projection of the map.

Input:

- vec2pix_func: a function taking theta,phi and returning pixel number
- **map: an array containing the spherical map to project,** the pixelisation is described by vec2pix_func

Return:

- a 2D array with the projection of the map.

Note: the Projector must contain information on the array.

healpy.projector.MollweideProj.set_flip

MollweideProj.**set_flip**(*flipconv*)
flipconv is either 'astro' or 'geo'. None will be default.

With 'astro', east is toward left and west toward right. It is the opposite for 'geo'

healpy.projector.MollweideProj.set_proj_plane_info

MollweideProj.**set_proj_plane_info**(*xsize*)

healpy.projector.MollweideProj.vec2xy

MollweideProj.**vec2xy**(*vx*, *vy=None*, *vz=None*, *direct=False*)
From unit vector direction to position in the projection plane (Mollweide).

Input:

- vx: if vy and vz are None, vx[0],vx[1],vx[2] defines the unit vector.
- vy,vz: if defined, vx,vy,vz define the unit vector
- lonlat: if True, angle are assumed in degree, and longitude, latitude
- flipconv is either 'astro' or 'geo'. None will be default.

Return:

- x, y: position in Mollweide plane.

healpy.projector.MollweideProj.xy2ang

MollweideProj.**xy2ang**(*x*, *y=None*, *lonlat=False*, *direct=False*)
From position in the projection plane to angular direction (Mollweide).

Input:

- x : if y is None, x[0], x[1] define the position in Mollweide plane.
- y : if defined, x,y define the position in projection plane.
- lonlat: if True, angle are assumed in degree, and longitude, latitude
- flipconv is either 'astro' or 'geo'. None will be default.

Return:

- theta, phi : angular direction.

healpy.projector.MollweideProj.xy2ij

MollweideProj.**xy2ij**(*x*, *y=None*)
From position in the projection plane to image array index (Mollweide).

Input:

- x : if y is None, x[0], x[1] define the position in Mollweide plane.
- y : if defined, x,y define the position in projection plane.

- projinfo : additional projection information.

Return:

- i,j : image array indices.

healpy.projector.MollweideProj.xy2vec

MollweideProj.**xy2vec**(*x*, *y=None*, *direct=False*)

From position in the projection plane to unit vector direction (Mollweide).

Input:

- x : if y is None, x[0], x[1] define the position in Mollweide plane.
- y : if defined, x,y define the position in projection plane.
- lonlat: if True, angle are assumed in degree, and longitude, latitude
- flipconv is either 'astro' or 'geo'. None will be default.

Return:

- theta, phi : angular direction.

healpy.projector.CartesianProj

class healpy.projector.**CartesianProj**(*rot=None*, *coord=None*, *xsize=800*, *ysize=None*, *lonra=None*, *latra=None*, ***kwds*)

This class provides class methods for Cartesian projection.

Attributes

arrayinfo	Dictionary with information on the projection array

healpy.projector.CartesianProj.arrayinfo

CartesianProj.**arrayinfo**

Dictionary with information on the projection array

Methods

ang2xy(theta[, phi, lonlat, direct])	From angular direction to position in the projection plane (Cartesian).
get_center([lonlat])	Get the center of the projection.
get_extent()	Get the extension of the projection plane.
get_fov()	
get_proj_plane_info()	
ij2xy([i, j])	From image array indices to position in projection plane (Cartesian).
mkcoord(coord)	

Continued on next page

Table 3.32 – continued from previous page

projmap(map, vec2pix_func[, rot, coord])	Create an array containing the projection of the map.
set_flip(flipconv)	flipconv is either 'astro' or 'geo'. None will be default.
set_proj_plane_info(xsize, ysize, lonra, latra)	
vec2xy(vx[, vy, vz, direct])	From unit vector direction to position in the projection plane (Cartesian).
xy2ang(x[, y, lonlat, direct])	From position in the projection plane to angular direction (Cartesian).
xy2ij(x[, y])	From position in the projection plane to image array index (Cartesian).
xy2vec(x[, y, direct])	From position in the projection plane to unit vector direction (Cartesian).

healpy.projector.CartesianProj.ang2xy

CartesianProj.**ang2xy**(*theta*, *phi=None*, *lonlat=False*, *direct=False*)

From angular direction to position in the projection plane (Cartesian).

Input:

- theta: if phi is None, theta[0] contains theta, theta[1] contains phi
- phi : if phi is not None, theta,phi are direction
- lonlat: if True, angle are assumed in degree, and longitude, latitude
- flipconv is either 'astro' or 'geo'. None will be default.

Return:

- x, y: position in Cartesian plane.

healpy.projector.CartesianProj.get_center

CartesianProj.**get_center**(*lonlat=False*)

Get the center of the projection.

Input:

- **lonlat** [if True, will return longitude and latitude in degree,] otherwise, theta and phi in radian

Return:

- theta,phi or lonlat depending on lonlat keyword

healpy.projector.CartesianProj.get_extent

CartesianProj.**get_extent**()

Get the extension of the projection plane.

Return: extent = (left,right,bottom,top)

healpy.projector.CartesianProj.get_fov

CartesianProj.**get_fov**()

healpy.projector.CartesianProj.get_proj_plane_info

CartesianProj.**get_proj_plane_info**()

healpy.projector.CartesianProj.ij2xy

CartesianProj.**ij2xy**(*i=None*, *j=None*)

From image array indices to position in projection plane (Cartesian).

Input:

- if i and j are None, generate arrays of i and j as input
- i : if j is None, i[0], j[1] define array indices in Cartesian image.
- j : if defined, i,j define array indices in image.
- projinfo : additional projection information.

Return:

- x,y : position in projection plane.

healpy.projector.CartesianProj.mkcoord

CartesianProj.**mkcoord**(*coord*)

healpy.projector.CartesianProj.projmap

CartesianProj.**projmap**(*map*, *vec2pix_func*, *rot=None*, *coord=None*)

Create an array containing the projection of the map.

Input:

- vec2pix_func: a function taking theta,phi and returning pixel number
- **map: an array containing the spherical map to project,** the pixelisation is described by vec2pix_func

Return:

- a 2D array with the projection of the map.

Note: the Projector must contain information on the array.

healpy.projector.CartesianProj.set_flip

CartesianProj.**set_flip**(*flipconv*)

flipconv is either 'astro' or 'geo'. None will be default.

With 'astro', east is toward left and west toward right. It is the opposite for 'geo'

healpy.projector.CartesianProj.set_proj_plane_info

CartesianProj.**set_proj_plane_info**(*xsize*, *ysize*, *lonra*, *latra*)

healpy.projector.CartesianProj.vec2xy

CartesianProj.**vec2xy**(*vx*, *vy=None*, *vz=None*, *direct=False*)

From unit vector direction to position in the projection plane (Cartesian).

Input:

- vx: if vy and vz are None, vx[0],vx[1],vx[2] defines the unit vector.
- vy,vz: if defined, vx,vy,vz define the unit vector
- lonlat: if True, angle are assumed in degree, and longitude, latitude
- flipconv is either 'astro' or 'geo'. None will be default.

Return:

- x, y: position in Cartesian plane.

healpy.projector.CartesianProj.xy2ang

CartesianProj.**xy2ang**(*x*, *y=None*, *lonlat=False*, *direct=False*)

From position in the projection plane to angular direction (Cartesian).

Input:

- x : if y is None, x[0], x[1] define the position in Cartesian plane.
- y : if defined, x,y define the position in projection plane.
- lonlat: if True, angle are assumed in degree, and longitude, latitude
- flipconv is either 'astro' or 'geo'. None will be default.

Return:

- theta, phi : angular direction.

healpy.projector.CartesianProj.xy2ij

CartesianProj.**xy2ij**(*x*, *y=None*)

From position in the projection plane to image array index (Cartesian).

Input:

- x : if y is None, x[0], x[1] define the position in Cartesian plane.
- y : if defined, x,y define the position in projection plane.
- projinfo : additional projection information.

Return:

- i,j : image array indices.

healpy.projector.CartesianProj.xy2vec

CartesianProj.**xy2vec**(*x*, *y=None*, *direct=False*)

From position in the projection plane to unit vector direction (Cartesian).

Input:

- x : if y is None, x[0], x[1] define the position in Cartesian plane.
- y : if defined, x,y define the position in projection plane.
- lonlat: if True, angle are assumed in degree, and longitude, latitude
- flipconv is either 'astro' or 'geo'. None will be default.

Return:

- theta, phi : angular direction.

`zoomtool` – Interactive visualisation

Interactive map visualization

mollzoom([map, fig, rot, coord, unit, ...])	Interactive mollweide plot with zoomed gnomview.

healpy.zoomtool.mollzoom

healpy.zoomtool.**mollzoom**(*map=None*, *fig=None*, *rot=None*, *coord=None*, *unit=''*, *xsize=800*, *title='Mollweide view'*, *nest=False*, *min=None*, *max=None*, *flip='astro'*, *remove_dip=False*, *remove_mono=False*, *gal_cut=0*, *format='%g'*, *cmap=None*, *norm=None*, *hold=False*, *margins=None*, *sub=None*)

Interactive mollweide plot with zoomed gnomview.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

License

Licenses

Healpy License

Healpy is licensed under the GNU General Public License.

GNU GENERAL PUBLIC LICENSE Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The licenses for most software are designed to take away your

freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not

price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid

anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether

gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and

(2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain

that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software

patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and

modification follow.

GNU GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains

a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's

source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion

of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it,

under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program

except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received

copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not

signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the

Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent

infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in

certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions

of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free

programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY

FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING

WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest

possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest

to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the program's name and a brief idea of what it does.> Copyright (C) <year> <name of author>
>
> This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

> Gnomovision version 69, Copyright (C) year name of author Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

> Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (which makes passes at compilers) written by James Hacker.
>
> <signature of Ty Coon>, 1 April 1989 Ty Coon, President of Vice

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Index

Symbols

A

B

C

D

F

G

X