pixell Documentation

Release 0.23.8+6.gcc0a46f.dirty

Simons Observatory Collaboration Analysis Library Task Force

Apr 11, 2024

CONTENTS:

CHAPTER

ONE

PIXELL

`pixell` is a library for loading, manipulating and analyzing maps stored in rectangular pixelization. It is mainly targeted for use with maps of the sky (e.g. CMB intensity and polarization maps, stacks of 21 cm intensity maps, binned galaxy positions or shear) in cylindrical projection, but its core functionality is more general. It extends numpy's `ndarray` to an `ndmap` class that associates a World Coordinate System (WCS) with a numpy array. It includes tools for Fourier transforms (through numpy or pyfft) and spherical harmonic transforms (through ducc0) of such maps and tools for visualization (through the Python Image Library).

- Free software: BSD license
- Documentation: https://pixell.readthedocs.io.
- Tutorials

1.1 Dependencies

- Python>=3.7
- gcc/gfortran or Intel compilers (clang might not work out of the box), if compiling from source
- ducc0, healpy, Cython, astropy, numpy, scipy, matplotlib, pyyaml, h5py, Pillow (Python Image Library)

On MacOS, and other systems with non-traditional environments, you should specify the following standard environment variables:

- `CC`: C compiler (example: `gcc`)
- `CXX`: C++ compiler (example: `g++`)
- `FC`: Fortran compiler (example: `gfortran`)

We recommend using `gcc` installed from Homebrew to access these compilers on MacOS, and you should make sure to point e.g. `$CC` to the full path of your gcc installation, as the `gcc` name usually points to the Apple `clang` install by default.

1.2 Runtime threading behaviour

Certain parts of `pixell` are parallelized using OpenMP, with the underlying `ducc0` library using pthreads. By default, these libraries use the number of cores on your system to determine the number of threads to use. If you wish to override this behaviour, you can use two environment variables:

- `OMP_NUM_THREADS` will set both the number of `pixell` threads and `ducc0` threads.
- `DUCC0_NUM_THREADS` will set the number of threads for the `ducc0` library to use, overwriting `OMP_NUM_THREADS` if both are set. `pixell` behaviour is not affected.

If you are using a modern chip (e.g. Apple M series chips, Intel 12th Gen or newer) that have both efficiency and performance cores, you may wish to set `OMP_NUM_THREADS` to the number of performance cores in your system. This will ensure that the efficiency cores are not used for the parallelized parts of `pixell` and `ducc0`.

1.3 Installing

Make sure your `pip` tool is up-to-date. To install `pixell`, run:

```
$ pip install pixell --user
$ test-pixell
```

This will install a pre-compiled binary suitable for your system (only Linux and Mac OS X with Python>=3.7 are supported). Note that you need `~/.local/bin` to be in your `PATH` for the latter `test-pixell` to work.

If you require more control over your installation, e.g. using Intel compilers, please see the section below on compiling from source. The `test-pixell` command will run a suite of unit tests.

1.3.1 Compiling from source (advanced / development workflow)

For compilation instructions specific to NERSC/cori, see NERSC.

For all other, below are general instructions.

First, download the source distribution or `git clone` this repository. You can work from `master` or checkout one of the released version tags (see the Releases section on Github). Then change into the cloned/source directory.

1.3.2 Run `setup.py`

If not using Intel compilers (see below), build the package using

```
$ python setup.py build_ext -i
```

You may now test the installation:

```
$ py.test pixell/tests/
```

If the tests pass, you can install the package (optionally with `-e` if you would like to edit the files after installation)

```
$ python setup.py install --user
```

1.3.3 Intel compilers

Intel compilers require you to modify the build step above as follows

```
$ python setup.py build_ext -i --fcompiler=intelem --compiler=intelem
```

On some systems, further specification might be required (make sure to get a fresh copy of the repository before trying out a new install method), e.g.:

```
$ LDSHARED="icc -shared" LD=icc LINKCC=icc CC=icc python setup.py build_ext -i --
↪fcompiler=intelem --compiler=intelem
```

1.4 Contributions

If you have write access to this repository, please:

1. create a new branch
2. push your changes to that branch
3. merge or rebase to get in sync with master
4. submit a pull request on github

If you do not have write access, create a fork of this repository and proceed as described above. For more details, see Contributing.

CHAPTER

TWO

1 USAGE

2.1 1.1 The ndmap object

The pixell library supports manipulation of sky maps that are represented as 2-dimensional grids of rectangular pixels. The supported projection and pixelization schemes are a subset of the schemes supported by FITS conventions. In addition, we provide support for a `plain' coordinate system, corresponding to a Cartesian plane with identically shaped pixels (useful for true flat-sky calculations).

In pixell, a map is encapsulated in an ndmap, which combines two objects: a numpy array (of at least two dimensions) whose two trailing dimensions correspond to two coordinate axes of the map, and a wcs object that specifies the World Coordinate System. The wcs component is an instance of Astropy's astropy.wcs.wcs.WCS class. The combination of the wcs and the shape of the numpy array completely specifies the footprint of a map of the sky, and is called the geometry. This library helps with manipulation of ndmap objects in ways that are aware of and preserve the validity of the wcs information.

2.1.1 1.1.1 ndmap as an extension of numpy.ndarray

The ndmap class extends the numpy.ndarray class, and thus has all of the usual attributes (.shape, .dtype, etc.) of an ndarray. It is likely that an ndmap object can be used in any functions that usually operate on an ndarray; this includes the usual numpy array arithmetic, slicing, broadcasting, etc.

```
>>> from pixell import enmap
>>> #... code that resulted in an ndmap called imap
>>> print(imap.shape, imap.wcs)
(100, 100) :{cdelt:[1,1],crval:[0,0],crpix:[0,0]}
>>> imap_extract = imap[:50,:50]   # A view of one corner of the map.
>>> imap_extract *= 1e6            # Re-calibrate. (Also affects imap!)
```

An ndmap must have at least two dimensions. The two right-most axes represent celestial coordinates (typically Declination and Right Ascension). Maps can have arbitrary number of leading dimensions, but many of the pixell CMB-related tools interpret 3D arrays with shape (ncomp,Ny,Nx) as representing Ny x Nx maps of intensity, polarization Q and U Stokes parameters, in that order.

Note that wcs information is correctly adjusted when the array is sliced; for example the object returned by imap[:50, :50] is a view into the imap data attached to a new wcs object that correctly describes the footprint of the extracted pixels.

Apart from all the numpy functionality, ndmap comes with a host of additional attributes and functions that utilize the WCS information.

2.1.2 1.1.2 `ndmap.wcs`

The `wcs` information describes the correspondence between celestial coordinates (typically the Right Ascension and Declination in the Equatorial system) and the pixel indices in the two right-most axes. In some projections, such as CEA or CAR, rows (and columns) of the pixel grid will often follow lines of constant Declination (and Right Ascension). In other projections, this will not be the case.

The WCS system is very flexible in how celestial coordinates may be associated with the pixel array. By observing certain conventions, we can make life easier for users of our maps. We recommend the following:

- The first pixel, index [0,0], should be the one that you would normally display (on a monitor or printed figure) in the lower left-hand corner of the image. The pixel indexed by [0,1] should appear to the right of [0,0], and pixel [1,0] should be above pixel [0,0]. (This recommendation originates in FITS standards documentation.)
- When working with large maps that are not near the celestial poles, Right Ascension should be roughly horizontal and Declination should be roughly vertical. (It should go without saying that you should also present information "as it would appear on the sky", i.e. with Right Ascension increasing to the left!)

The examples in the rest of this document are designed to respect these two conventions.

TODO: I've listed below common operations that would be useful to demonstrate here. Finish this! (See *2 Reference* for a dump of all member functions)

2.2 1.2 Creating an `ndmap`

To create an empty `ndmap`, call the `enmap.zeros` or `enmap.empty` functions and specify the map shape as well as the pixelization information (the WCS). Here is a basic example:

```
>>> from pixell import enmap, utils
>>> box = np.array([[-5,10],[5,-10]]) * utils.degree
>>> shape,wcs = enmap.geometry(pos=box,res=0.5 * utils.arcmin,proj='car')
>>> imap = enmap.zeros((3,) + shape, wcs=wcs)
```

In this example we are requesting a pixelization that spans from -5 to +5 in declination, and +10 to -10 in Right Ascension. Note that we need to specify the Right Ascension coordinates in decreasing order, or the map, when we display it with pixel [0,0] in the lower left-hand corner, will not have the usual astronomical orientation.

For more information on designing the geometry, see *1.10 Building a map geometry*.

2.3 1.3 Passing maps through functions that act on `numpy` arrays

You can also perform arithmetic with and use functions that act on numpy arrays. In most situations, functions that usually act on numpy arrays will return an `ndmap` when an `ndmap` is passed to it in lieu of a numpy array. In those situations where the WCS information is removed, one can always add it back like this:

```
>>> from pixell import enmap
>>> #... code that resulted in an ndmap called imap
>>> print(imap.shape, imap.wcs)
(100, 100) :{cdelt:[1,1],crval:[0,0],crpix:[0,0]}
>>> omap = some_function(imap)
>>> print(omap.wcs)
Traceback (most recent call last):
AttributeError: 'numpy.ndarray' object has no attribute 'wcs'
```

(continues on next page)

(continued from previous page)

```
>>> # Uh oh, the WCS information was removed by some_function
>>> omap = enmap.ndmap(omap, wcs) # restore the wcs
>>> omap = enmap.samewcs(omap, imap) # another way to restore the wcs
>>> # This does the same thing, but force-copies the data array.
>>> omap = enmap.enmap(omap, wcs)
```

Note that `ndmap` and `samewcs` will not copy the underlying data array if they don't have to; the returned object will reference the same memory used by the input array (as though you had done numpy.asarray). In contrast, `enmap.enmap` will always create a copy of the input data.

2.4 1.4 Reading maps from disk

An entire map in `FITS` or `HDF` format can be loaded using `read_map`, which is found in the module `pixell.enmap`. The `enmap` module contains the majority of map manipulation functions.

```
>>> from pixell import enmap
>>> imap = enmap.read_map("map_on_disk.fits")
```

Alternatively, one can select a rectangular region specified through its bounds using the `box` argument,

```
>>> import numpy as np
>>> from pixell import utils
>>> dec_min = -5 ; ra_min = -5 ; dec_max = 5 ; ra_max = 5
>>> # All coordinates in pixell are specified in radians
>>> box = np.array([[dec_min,ra_min],[dec_max,ra_max]]) * utils.degree
>>> imap = enmap.read_map("map_on_disk.fits",box=box)
```

Note the convention used to define coordinate boxes in pixell. To learn how to use a pixel coordinate box or a numpy slice, please read the docstring for `read_map`.

2.5 1.5 Inspecting a map

An `ndmap` has all the attributes of a `ndarray` numpy array. In particular, you can inspect its shape.

```
>>> print(imap.shape)
(3,500,1000)
```

Here, `imap` consists of three maps each with 500 pixels along the Y axis and 1000 pixels along the X axis. One can also inspect the WCS of the map,

```
>>> print(imap.wcs)
car:{cdelt:[0.03333,0.03333],crval:[0,0],crpix:[500.5,250.5]}
```

Above, we learn that the map is represented in the `CAR` projection system and what the WCS attributes are.

2.6 1.6 Selecting regions of the sky

If you know the pixel coordinates of the sub-region you would like to select, the cleanest thing to do is to slice it like a numpy array.

```
>>> imap = enmap.zeros((1000,1000))
>>> print(imap.shape)
(1000,1000)
>>> omap = imap[100:200,50:80]
>>> print(omap.shape)
(100, 30)
```

However, if you only know the physical coordinate bounding box in radians, you can use the `submap` function.

```
>>> box = np.array([[dec_min,ra_min],[dec_max,ra_max]]) # in radians
>>> omap = imap.submap(box)
>>> omap = enmap.submap(imap,box) # an alternative way
```

2.7 1.7 Relating pixels to the sky

The geometry specified through `shape` and `wcs` contains all the information to get properties of the map related to the sky. `pixell` always specifies the Y coordinate first. So a sky position is often in the form `(dec,ra)` where `dec` could be the declination and `ra` could be the right ascension in radians in the equatorial coordinate system.

The pixel corresponding to ra=180,dec=20 can be obtained like

```
>>> dec = 20 ; ra = 180
>>> coords = np.deg2rad(np.array((dec,ra)))
>>> ypix,xpix = enmap.sky2pix(shape,wcs,coords)
```

Note that you don't need to pass each dec,ra separately. You can pass a large number of coordinates for a vectorized conversion. In this case *coords* should have the shape (2,Ncoords), where Ncoords is the number of coordinates you want to convert, with the first row containing declination and the second row containing right ascension. Also, the returned pixel coordinates are in general fractional.

Similarly, pixel coordinates can be converted to sky coordinates

```
>>> ypix = 100 ; xpix = 300
>>> pixes = np.array((ypix,xpix))
>>> dec,ra = enmap.pix2sky(shape,wcs,pixes)
```

with similar considerations as above for passing a large number of coordinates.

Using the `enmap.posmap` function, you can get a map of shape (2,Ny,Nx) containing the coordinate positions in radians of each pixel of the map.

```
>>> posmap = imap.posmap()
>>> dec = posmap[0] # declination in radians
>>> ra = posmap[1] # right ascension in radians
```

Using the `enmap.pixmap` function, you can get a map of shape (2,Ny,Nx) containing the integer pixel coordinates of each pixel of the map.

```
>>> pixmap = imap.pixmap()
>>> pixy = posmap[0]
>>> pixx = posmap[1]
```

Using the `enmap.modrmap` function, you can get a map of shape (Ny,Nx) containing the physical coordinate distance of each pixel from a given reference point specified in radians. If the reference point is unspecified, the distance of each pixel from the center of the map is returned.

```
>>> modrmap = imap.modrmap() # 2D map of distances from center
```

2.8 1.8 Fourier operations

Maps can be 2D Fourier-transformed for manipulation in Fourier space. The 2DFT of the (real) map is generally a complex `ndmap` with the same shape as the original map (unless a real transform function is used). To facilitate 2DFTs, there are functions that do the Fourier transforms themselves, and functions that provide metadata associated with such transforms.

Since an *ndmap* contains information about the physical extent of the map and the physical width of the pixels, the discrete frequencies corresponding to its numpy array need to be converted to physical wavenumbers of the map.

This is done by the `laxes` function, which returns the wavenumbers along the Y and X directions. The `lmap` function returns a map of all the `(ly,lx)` wavenumbers in each pixel of the Fourier-space map. The `modlmap` function returns the "modulus of lmap", i.e. a map of the distances of each Fourier-pixel from `(ly=0,lx=0)`.

You can perform a fast Fourier transform of an (…,Ny,Nx) dimensional *ndmap* to return an (…,Ny,Nx) dimensional complex map using `enmap.fft` and `enmap.ifft` (inverse FFT).

2.9 1.9 Filtering maps in Fourier space

A filter can be applied to a map in three steps:

1. prepare a Fourier space filter `kfilter`
2. Fourier transform the map `imap` to `kmap`
3. multiply the filter and k-map
4. inverse Fourier transform the result

2.10 1.10 Building a map geometry

You can create a geometry if you know what its bounding box and pixel size are:

```
>>> from pixell import enmap, utils
>>> box = np.array([[-5,10],[5,-10]]) * utils.degree
>>> shape,wcs = enmap.geometry(pos=box,res=0.5 * utils.arcmin,proj='car')
```

This creates a CAR geometry centered on RA=0d,DEC=0d with a width of 20 degrees, a height of 10 degrees, and a pixel size of 0.5 arcminutes.

You can create a full-sky geometry by just specifying the resolution:

```
>>> from pixell import enmap, utils
>>> shape,wcs = enmap.fullsky_geometry(res=0.5 * utils.arcmin,proj='car')
```

This creates a CAR geometry with pixel size of 0.5 arcminutes that wraps around the whole sky.

You can create a geometry that wraps around the full sky but does not extend everywhere in declination:

```
>>> shape,wcs = enmap.band_geometry(dec_cut=20*utils.degree, res=0.5 * utils.arcmin,proj=
↪'car')
```

This creates a CAR geometry with pixel size of 0.5 arcminutes that wraps around the whole sky but is limited to DEC=-20d to 20d. The following creates the same except with a declination extent from -60d to 30d.

```
>>> shape,wcs = enmap.band_geometry(dec_cut=np.array([-60,30])*utils.degree, res=0.5 *␣
↪utils.arcmin,proj='car')
```

2.11 1.11 Resampling maps

2.12 1.12 Masking and windowing

2.13 1.13 Flat-sky diagnostic power spectra

2.14 1.14 Curved-sky operations

The resulting spherical harmonic *alm* coefficients of an SHT are stored in the same convention as with `HEALPIX`, so one can use `healpy.almxfl` to apply an isotropic filter to an SHT.

2.15 1.15 Reprojecting maps

2.16 1.16 Simulating maps

CHAPTER

THREE

2 REFERENCE

See *1 Usage* for how to use these functions for common map manipulation tasks.

3.1 2.1 enmap - General map manipulation

class pixell.enmap.**ndmap**(*arr*, *wcs*)

Implements (stacks of) flat, rectangular, 2-dimensional maps as a dense numpy array with a fits WCS. The axes have the reverse ordering as in the fits file, and hence the WCS object. This class is usually constructed by using one of the functions following it, much like numpy arrays. We assume that the WCS only has two axes with unit degrees. The ndmap itself uses radians for everything.

copy(*order='C'*)

Return a copy of the array.

Parameters

order (*{'C', 'F', 'A', 'K'}, optional*) – Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `numpy.copy()` are very similar but have different default values for their order= arguments, and this function always passes sub-classes through.)

See also:

`numpy.copy`
Similar function with different default behavior

`numpy.copyto`

Notes

This function is the preferred method for creating an array copy. The function `numpy.copy()` is similar, but it defaults to using order 'K', and will not pass sub-classes through by default.

Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

sky2pix(*coords*, *safe=True*, *corner=False*)

pix2sky(*pix*, *safe=True*, *corner=False*)

l2pix(*ls*)

pix2l(*pix*)

contains(*pos*, *unit='coord'*)

corners(*npoint=10*, *corner=True*)

box(*npoint=10*, *corner=True*)

pixbox_of(*oshape*, *owcs*)

posmap(*safe=True*, *corner=False*, *separable='auto'*, *dtype=<class 'numpy.float64'>*)

posaxes(*safe=True*, *corner=False*, *dtype=<class 'numpy.float64'>*)

pixmap()

laxes(*oversample=1*, *method='auto'*)

lmap(*oversample=1*)

lform(*shift=True*)

modlmap(*oversample=1*, *min=0*)

modrmap(*ref='center'*, *safe=True*, *corner=False*)

lbin(*bsize=None*, *brel=1.0*, *return_nhit=False*, *return_bins=False*)

rbin(*center=[0, 0]*, *bsize=None*, *brel=1.0*, *return_nhit=False*, *return_bins=False*)

area()

`pixsize`()

`pixshape`(*signed=False*)

`pixsizemap`(*separable='auto'*, *broadcastable=False*)

`pixshapemap`(*separable='auto'*, *signed=False*)

`lpixsize`(*signed=False*, *method='auto'*)

`lpixshape`(*signed=False*, *method='auto'*)

`extent`(*method='auto'*, *signed=False*)

`property preflat`

Returns a view of the map with the non-pixel dimensions flattened.

`property npix`

`property geometry`

`resample`(*oshape*, *off=(0, 0)*, *method='fft'*, *mode='wrap'*, *corner=False*, *order=3*)

`project`(*shape*, *wcs*, *order=3*, *mode='constant'*, *cval=0*, *prefilter=True*, *mask_nan=False*, *safe=True*)

`extract`(*shape*, *wcs*, *omap=None*, *wrap='auto'*, *op=<function ndmap.<lambda>>*, *cval=0*, *iwcs=None*, *reverse=False*)

`extract_pixbox`(*pixbox*, *omap=None*, *wrap='auto'*, *op=<function ndmap.<lambda>>*, *cval=0*, *iwcs=None*, *reverse=False*)

`insert`(*imap*, *wrap='auto'*, *op=<function ndmap.<lambda>>*, *cval=0*, *iwcs=None*)

`insert_at`(*pix*, *imap*, *wrap='auto'*, *op=<function ndmap.<lambda>>*, *cval=0*, *iwcs=None*)

`at`(*pos*, *order=3*, *mode='constant'*, *cval=0.0*, *unit='coord'*, *prefilter=True*, *mask_nan=False*, *safe=True*)

`argmax`(*axis=None*, *out=None*)

Return indices of the maximum values along the given axis.

Refer to *numpy.argmax* for full documentation.

See also:

`numpy.argmax`
equivalent function

`autocrop`(*method='plain'*, *value='auto'*, *margin=0*, *factors=None*, *return_info=False*)

`apod`(*width*, *profile='cos'*, *fill='zero'*)

`stamps`(*pos*, *shape*, *aslist=False*)

`distance_from`(*points*, *omap=None*, *odomains=None*, *domains=False*, *method='cellgrid'*, *rmax=None*, *step=1024*)

`distance_transform`(*omap=None*, *rmax=None*, *method='cellgrid'*)

`labeled_distance_transform`(*omap=None*, *odomains=None*, *rmax=None*, *method='cellgrid'*)

property plain

padslice(*box*, *default=nan*)

center()

downgrade(*factor*, *op=<function mean>*, *ref=None*, *off=None*)

upgrade(*factor*, *off=None*, *oshape=None*, *inclusive=False*)

fillbad(*val=0*, *inplace=False*)

to_healpix(*nside=0*, *order=3*, *omap=None*, *chunk=100000*, *destroy_input=False*)

to_flipper(*omap=None*, *unpack=True*)

submap(*box*, *mode=None*, *wrap='auto'*)

Extract the part of the map inside the given coordinate box box : array_like

The [[fromy,fromx],[toy,tox]] coordinate box to select. The resulting map will have bottom-left and top-right corners as close as possible to this, but will differ slightly due to the finite pixel size.

mode
[str]

How to handle partially selected pixels:
"round": round bounds using standard rules "floor": both upper and lower bounds will be rounded down "ceil": both upper and lower bounds will be rounded up "inclusive": lower bounds are rounded down, and upper bounds up "exclusive": lower bounds are rounded up, and upper bounds down

subinds(*box*, *mode=None*, *cap=True*)

write(*fname*, *fmt=None*)

pixell.enmap.**submap**(*map*, *box*, *mode=None*, *wrap='auto'*, *iwcs=None*)

Extract the part of the map inside the given coordinate box box : array_like

The [[fromy,fromx],[toy,tox]] coordinate box to select. The resulting map will have corners as close as possible to this, but will differ slightly due to the finite pixel size.

mode
[str]

How to handle partially selected pixels:
"round": round bounds using standard rules "floor": both upper and lower bounds will be rounded down "ceil": both upper and lower bounds will be rounded up "inclusive": lower bounds are rounded down, and upper bounds up "exclusive": lower bounds are rounded up, and upper bounds down

The iwcs argument allows the wcs to be overriden. This is usually not necessary.

pixell.enmap.**subinds**(*shape*, *wcs*, *box*, *mode=None*, *cap=True*, *noflip=False*)

Helper function for submap. Translates the coordinate box provided into a pixel units.

When box is translated into pixels, the result will in general have fractional pixels, which need to be rounded before we can do any slicing. To get as robust results as possible, we want

1. two boxes that touch should results in iboxses that also touch. This means that upper and lower bounds must be handled consistently. inclusive and exclusive modes break this, and should be used with caution.

2. tiny floating point errors should not usually be able to cause the ibox to change. Most boxes will have some simple fraction of a whole degree, and most have pixels with centers at a simple fraction of a whole degree. Hence, it is likely that box edges will fall almost exactly on an integer pixel value. floor and ceil will then move us around by a whole pixel based on tiny numerical jitter around this value. Hence these should be used with caution.

These concerns leave us with mode = "round" as the only generally safe alternative, which is why it's default.

`pixell.enmap.`**`slice_geometry`**`(`*shape*, *wcs*, *sel*, *nowrap=False*`)`

Slice a geometry specified by shape and wcs according to the slice sel. Returns a tuple of the output shape and the correponding wcs.

`pixell.enmap.`**`scale_geometry`**`(`*shape*, *wcs*, *scale*`)`

Scale the geometry so that the number of pixels is scaled by the factor *scale*.

`pixell.enmap.`**`get_unit`**`(`*wcs*`)`

`pixell.enmap.`**`npix`**`(`*shape*`)`

`class` `pixell.enmap.`**`Geometry`**`(`*shape*, *wcs=None*`)`

`property npix`

`property nopre`

`with_pre``(`*pre*`)`

`submap``(`*box=None*, *pixbox=None*, *mode=None*, *wrap='auto'*, *noflip=False*`)`

`scale``(`*scale*`)`

`downgrade``(`*factor*, *op=<function mean>*`)`

`copy``()`

`sky2pix``(`*coords*, *safe=True*, *corner=False*`)`

`pix2sky``(`*pix*, *safe=True*, *corner=False*`)`

`l2pix``(`*ls*`)`

`pix2l``(`*pix*`)`

`pixell.enmap.`**`corners`**`(`*shape*, *wcs*, *npoint=10*, *corner=True*`)`

Return the coordinates of the bottom left and top right corners of the geometry given by shape, wcs.

If corner==True it is similar to enmap.pix2sky([[-0.5,shape[-2]-0.5],[-0.5,shape[-1]-0.5]]). That is, it return sthe coordinate of the bottom left corner of the bottom left pixel and the top right corner of the top right pixel. If corner==False, then it instead returns the corresponding pixel centers.

It differs from the simple pix2sky calls above by handling 2*pi wrapping ambiguities differently. enmap.corners ensures that the coordinates returned are on the same side of the wrapping cut so that the coordinates of the two corners can be compared without worrying about wrapping. It does this by evaluating a set of intermediate points between the corners and counting and undoing any sudden jumps in coordinates it finds. This is controlled by the npoint option. The default of 10 should be more than enough.

Returns [{bottom left,top right},{dec,ra}] in radians (or equivalent for other coordinate systems). e.g. an array of the form [[dec_min, ra_min], [dec_max, ra_max]].

`pixell.enmap.`**`box`**(*shape*, *wcs*, *npoint=10*, *corner=True*)

Alias for corners.

`pixell.enmap.`**`enmap`**(*arr*, *wcs=None*, *dtype=None*, *copy=True*)

Construct an ndmap from data.

Parameters

- **arr** (*array_like*) – The data to initialize the map with. Must be at least two-dimensional.
- **wcs** (*WCS object*) –
- **dtype** (*data-type, optional*) – The data type of the map. Default: Same as arr.
- **copy** (*boolean*) – If true, arr is copied. Otherwise, a referance is kept.

`pixell.enmap.`**`empty`**(*shape*, *wcs=None*, *dtype=None*)

Return an enmap with entries uninitialized (like numpy.empty).

`pixell.enmap.`**`zeros`**(*shape*, *wcs=None*, *dtype=None*)

Return an enmap with entries initialized to zero (like numpy.zeros).

`pixell.enmap.`**`ones`**(*shape*, *wcs=None*, *dtype=None*)

Return an enmap with entries initialized to one (like numpy.ones).

`pixell.enmap.`**`full`**(*shape*, *wcs*, *val*, *dtype=None*)

Return an enmap with entries initialized to val (like numpy.full).

`pixell.enmap.`**`posmap`**(*shape*, *wcs*, *safe=True*, *corner=False*, *separable='auto'*, *dtype=<class 'numpy.float64'>*, *bsize=1000000.0*, *bcheck=False*)

Return an enmap where each entry is the coordinate of that entry, such that posmap(shape,wcs)[{0,1},j,k] is the {y,x}-coordinate of pixel (j,k) in the map. Results are returned in radians, and if safe is true (default), then sharp coordinate edges will be avoided. separable controls whether a fast calculation that assumes that ra is only a function of x and dec is only a function of y is used. The default is "auto", which determines this based on the wcs, but True or False can also be passed to control this manually.

For even greater speed, and to save memory, consider using posaxes directly for cases where you know that the wcs will be separable. For separable cases, separable=True is typically 15-20x faster than separable=False, while posaxes is 1000x faster.

`pixell.enmap.`**`posmap_old`**(*shape*, *wcs*, *safe=True*, *corner=False*)

`pixell.enmap.`**`posaxes`**(*shape*, *wcs*, *safe=True*, *corner=False*, *dtype=<class 'numpy.float64'>*, *bcheck=False*)

`pixell.enmap.`**`pixmap`**(*shape*, *wcs=None*)

Return an enmap where each entry is the pixel coordinate of that entry.

`pixell.enmap.`**`pix2sky`**(*shape*, *wcs*, *pix*, *safe=True*, *corner=False*, *bcheck=False*)

Given an array of pixel coordinates [{y,x},...], return sky coordinates in the same ordering.

`pixell.enmap.`**`sky2pix`**(*shape*, *wcs*, *coords*, *safe=True*, *corner=False*, *bcheck=False*)

Given an array of coordinates [{dec,ra},...], return pixel coordinates with the same ordering. The corner argument specifies whether pixel coordinates start at pixel corners or pixel centers. This represents a shift of half a pixel. If corner is False, then the integer pixel closest to a position is round(sky2pix(...)). Otherwise, it is floor(sky2pix(...)).

`pixell.enmap.`**`pix2l`**(*shape*, *wcs*, *pix*)

Given an array of fourier-pixel coordinates [{y,x},...], returns the 2d fourier coordinates [{ly,lx},...].

`pixell.enmap.l2pix`(*shape*, *wcs*, *ls*)

Given an array of fourier-pixel coordinates [{y,x},…], returns the 2d fourier coordinates [{ly,lx},…].

`pixell.enmap.skybox2pixbox`(*shape*, *wcs*, *skybox*, *npoint=10*, *corner=False*, *include_direction=False*)

Given a coordinate box [{from,to},{dec,ra}], compute a corresponding pixel box [{from,to},{y,x}]. We avoid wrapping issues by evaluating a number of subpoints.

`pixell.enmap.pixbox2skybox`(*shape*, *wcs*, *pixbox*)

`pixell.enmap.contains`(*shape*, *wcs*, *pos*, *unit='coord'*)

For the points with coordinates pos[{dec,ra},…] return whether each is inside the geometry given by shape, wcs

`pixell.enmap.project`(*map*, *shape*, *wcs*, *order=3*, *mode='constant'*, *cval=0.0*, *force=False*, *prefilter=True*, *mask_nan=False*, *safe=True*, *bsize=1000*)

Project the map into a new map given by the specified shape and wcs, interpolating as necessary. Handles nan regions in the map by masking them before interpolating. This uses local interpolation, and will lose information when downgrading compared to averaging down.

`pixell.enmap.pixbox_of`(*iwcs*, *oshape*, *owcs*)

Obtain the pixbox which when extracted from a map with WCS=iwcs returns a map that has geometry oshape,owcs.

`pixell.enmap.extract`(*map*, *shape*, *wcs*, *omap=None*, *wrap='auto'*, *op=<function <lambda>>*, *cval=0*, *iwcs=None*, *reverse=False*)

Like project, but only works for pixel-compatible wcs. Much faster because it simply copies over pixels.

Can be used in co-adding by specifying an output map and a combining operation. The deafult operation overwrites the output. Use np.ndarray.__iadd__ to get a copy-less += operation. Not that areas outside are not assumed to be zero if an omap is specified - instead those areas will simply not be operated on.

The optional iwcs argument is there to support input maps that are numpy-like but can't be made into actual enmaps. The main example of this is a fits hdu object, which can be sliced like an array to avoid reading more into memory than necessary.

`pixell.enmap.extract_pixbox`(*map*, *pixbox*, *omap=None*, *wrap='auto'*, *op=<function <lambda>>*, *cval=0*, *iwcs=None*, *reverse=False*)

This function extracts a rectangular area from an enmap based on the given pixbox[{from,to,[stride]},{y,x}]. The difference between this function and plain slicing of the enmap is that this one supports wrapping around the sky. This is necessary to make things like fast thumbnail or tile extraction at the edge of a (horizontally) fullsky map work.

`pixell.enmap.insert`(*omap*, *imap*, *wrap='auto'*, *op=<function <lambda>>*, *cval=0*, *iwcs=None*)

Insert imap into omap based on their world coordinate systems, which must be compatible. Essentially the reverse of extract.

`pixell.enmap.insert_at`(*omap*, *pix*, *imap*, *wrap='auto'*, *op=<function <lambda>>*, *cval=0*, *iwcs=None*)

Insert imap into omap at the position given by pix. If pix is [y,x], then [0:ny,0:nx] in imap will be copied into [y:y+ny,x:x+nx] in omap. If pix is [{from,to,[stride]},{y,x}], then this specifies the omap pixbox into which to copy imap. Wrapping is handled the same way as in extract.

`pixell.enmap.overlap`(*shape*, *wcs*, *shape2_or_pixbox*, *wcs2=None*, *wrap='auto'*)

Compute the overlap between the given geometry (shape, wcs) and another *compatible* geometry. This can be either another shape, wcs pair or a pixbox[{from,to},{y,x}]. Returns the geometry of the overlapping region.

`pixell.enmap.neighborhood_pixboxes`(*shape*, *wcs*, *poss*, *r*)

Given a set of positions poss[npos,{dec,ra}] in radians and a distance r in radians, return pixboxes[npos][{from,to},{y,x}] corresponding to the regions within a distance of r from each entry in poss.

`pixell.enmap.`**`at`**(*map*, *pos*, *order=3*, *mode='constant'*, *cval=0.0*, *unit='coord'*, *prefilter=True*, *mask_nan=False*, *safe=True*)

`pixell.enmap.`**`argmax`**(*map*, *unit='coord'*)

Return the coordinates of the maximum value in the specified map. If map has multiple components, the maximum value for each is returned separately, with the last axis being the position. If unit is "pix", the position will be given in pixels. Otherwise it will be in physical coordinates.

`pixell.enmap.`**`argmin`**(*map*, *unit='coord'*)

Return the coordinates of the minimum value in the specified map. See argmax for details.

`pixell.enmap.`**`rand_map`**(*shape*, *wcs*, *cov*, *scalar=False*, *seed=None*, *pixel_units=False*, *iau=False*, *spin=[0, 2]*)

Generate a standard flat-sky pixel-space CMB map in TQU convention based on the provided power spectrum. If cov.ndim is 4, 2D power is assumed else 1D power is assumed. If pixel_units is True, the 2D power spectra is assumed to be in pixel units, not in steradians.

`pixell.enmap.`**`rand_gauss`**(*shape*, *wcs*, *dtype=None*)

Generate a map with random gaussian noise in pixel space.

`pixell.enmap.`**`rand_gauss_harm`**(*shape*, *wcs*)

Mostly equivalent to np.fft.fft2(np.random.standard_normal(shape)), but avoids the fft by generating the numbers directly in frequency domain. Does not enforce the symmetry requried for a real map. If box is passed, the result will be an enmap.

`pixell.enmap.`**`rand_gauss_iso_harm`**(*shape*, *wcs*, *cov*, *pixel_units=False*)

Generates a random map with component covariance cov in harmonic space, where cov is a (comp,comp,l) array or a (comp,comp,Ny,Nx) array. Despite the name, the map doesn't need to be isotropic since 2D power spectra are allowed.

If cov.ndim is 4, cov is assumed to be an array of 2D power spectra. else cov is assumed to be an array of 1D power spectra. If pixel_units is True, the 2D power spectra is assumed to be in pixel units, not in steradians.

`pixell.enmap.`**`massage_spectrum`**(*cov*, *shape*)

given a spectrum cov[nl] or cov[n,n,nl] and a shape (stokes,ny,nx) or (ny,nx), return a new ocov that has a shape compatible with shape, padded with zeros if necessary. If shape is scalar (ny,nx), then ocov will be scalar (nl). If shape is (stokes,ny,nx), then ocov will be (stokes,stokes,nl).

`pixell.enmap.`**`extent`**(*shape*, *wcs*, *nsub=None*, *signed=False*, *method='auto'*)

Returns the area of a patch with the given shape and wcs, in steradians.

`pixell.enmap.`**`extent_intermediate`**(*shape*, *wcs*, *signed=False*)

Estimate the flat-sky extent of the map as the WCS intermediate coordinate extent. This is very simple, but is only appropriate for very flat coordinate systems

`pixell.enmap.`**`extent_subgrid`**(*shape*, *wcs*, *nsub=None*, *safe=True*, *signed=False*)

Returns an estimate of the "physical" extent of the patch given by shape and wcs as [height,width] in radians. That is, if the patch were on a sphere with radius 1 m, then this function returns approximately how many meters tall and wide the patch is. These are defined such that their product equals the physical area of the patch. Obs: Has trouble with areas near poles.

`pixell.enmap.`**`extent_cyl`**(*shape*, *wcs*, *signed=False*)

Extent specialized for a cylindrical projection. Vertical: ny*cdelt[1] Horizontal: Each row is nx*cdelt[0]*cos(dec), but we want a single representative number, which will be some kind of average, and we're free to choose which. We choose the one that makes the product equal the true area. Area = nx*ny*cdelt[0]*cdelt[1]*mean(cos(dec)) = vertical*(nx*cdelt[0]*mean(cos)), so horizontal = nx*cdelt[0]*mean(cos) `pixell.enmap.`**`area`**`(`*shape*, *wcs*, *nsamp=1000*, *method='auto'*`)`

Returns the area of a patch with the given shape and wcs, in steradians.

`pixell.enmap.`**`area_intermediate`**`(`*shape*, *wcs*`)`

Get the area of a completely flat sky

`pixell.enmap.`**`area_cyl`**`(`*shape*, *wcs*`)`

Get the area of a cylindrical projection. Fast and exact.

`pixell.enmap.`**`area_contour`**`(`*shape*, *wcs*, *nsamp=1000*`)`

Get the area of the map by doing a contour integral (1-sin(dec)) d(RA) over the closed path (dec(t), ra(t)) that bounds the valid region of the map, so it only works for projections where we can figure out this boundary. Using only d(RA) in the integral corresponds to doing a top-hat integral instead of something trapezoidal, but this method is fast enough that we can afford many points to compensate. The present implementation works for cases where the valid region of the map runs through the centers of the pixels on each edge or through the outer edge of those pixels (this detail can be different for each edge). The former case is needed in the full-sky cylindrical projections that have pixels centered exactly on the poles.

`pixell.enmap.`**`pixsize`**`(`*shape*, *wcs*`)`

Returns the average pixel area, in steradians.

`pixell.enmap.`**`pixshape`**`(`*shape*, *wcs*, *signed=False*`)`

Returns the average pixel height and width, in radians.

`pixell.enmap.`**`pixshapemap`**`(`*shape*, *wcs*, *bsize=1000*, *separable='auto'*, *signed=False*, *bcheck=False*`)`

Returns the physical width and heigh of each pixel in the map in radians. Heavy for big maps. Much faster approaches are possible for known pixelizations.

`pixell.enmap.`**`pixshapes_cyl`**`(`*shape*, *wcs*, *signed=False*, *bcheck=False*`)`

Returns the physical width and height of pixels for each row of a cylindrical map with the given shape, wcs, in radians, as an array [{height,width},ny]. All pixels in a row have the same shape in a cylindrical projection.

`pixell.enmap.`**`pixsizemap`**`(`*shape*, *wcs*, *separable='auto'*, *broadcastable=False*, *bsize=1000*, *bcheck=False*`)`

Returns the physical area of each pixel in the map in steradians.

If separable is True, then the map will be assumed to be in a cylindircal projection, where the pixel size is only a function of declination. This makes the calculation dramatically faster, and the resulting array will also use much less memory due to numpy striding tricks. The default, separable=auto", determines whether to use this shortcut based on the properties of the wcs.

Normally the function returns a ndmap of shape [ny,nx]. If broadcastable is True, then it is allowed to return a smaller array that would still broadcast correctly to the right shape. This can be useful to save work if one is going to be doing additional manipulation of the pixel size before using it. For a cylindrical map, the result would have shape [ny,1] if broadcastable is True.

BUG: This function assumes parallelogram-shaped pixels. This breaks for non-cylindrical projections!

`pixell.enmap.`**`pixsizemap_contour`**`(`*shape*, *wcs*, *bsize=1000*, *bcheck=False*`)`

`pixell.enmap.`**`pixshapebounds`**`(`*shape*, *wcs*, *separable='auto'*`)`

Return the minimum and maximum pixel height and width for the given geometry, in the form [{min,max},{y,x}]. Fast for separable geometries like cylindrical ones, which it will try to recognize, but this can be forced by setting separable to True (or disabled with False). Heavy in the general case.

`pixell.enmap.`**`lmap`**`(`*shape*, *wcs*, *oversample=1*, *method='auto'*`)`

Return a map of all the wavenumbers in the fourier transform of a map with the given shape and wcs.

`pixell.enmap.`**`modlmap`**(*shape*, *wcs*, *oversample=1*, *method='auto'*, *min=0*)

Return a map of all the abs wavenumbers in the fourier transform of a map with the given shape and wcs.

`pixell.enmap.`**`center`**(*shape*, *wcs*)

`pixell.enmap.`**`modrmap`**(*shape*, *wcs*, *ref='center'*, *safe=True*, *corner=False*)

Return an enmap where each entry is the distance from center of that entry. Results are returned in radians, and if safe is true (default), then sharp coordinate edges will be avoided.

`pixell.enmap.`**`laxes`**(*shape*, *wcs*, *oversample=1*, *method='auto'*)

`pixell.enmap.`**`lrmap`**(*shape*, *wcs*, *oversample=1*)

Return a map of all the wavenumbers in the fourier transform of a map with the given shape and wcs.

`pixell.enmap.`**`lpixsize`**(*shape*, *wcs*, *signed=False*, *method='auto'*)

`pixell.enmap.`**`lpixshape`**(*shape*, *wcs*, *signed=False*, *method='auto'*)

`pixell.enmap.`**`fft`**(*emap*, *omap=None*, *nthread=0*, *normalize=True*, *adjoint_ifft=False*, *dct=False*)

Performs the 2d FFT of the enmap pixels, returning a complex enmap. If normalize is “phy”, “phys” or “physical”, then an additional normalization is applied such that the binned square of the fourier transform can be directly compared to theory (apart from mask corrections) , i.e., pixel area factors are corrected for.

`pixell.enmap.`**`ifft`**(*emap*, *omap=None*, *nthread=0*, *normalize=True*, *adjoint_fft=False*, *dct=False*)

Performs the 2d iFFT of the complex enmap given, and returns a pixel-space enmap.

`pixell.enmap.`**`dct`**(*emap*, *omap=None*, *nthread=0*, *normalize=True*)

`pixell.enmap.`**`idct`**(*emap*, *omap=None*, *nthread=0*, *normalize=True*)

`pixell.enmap.`**`fft_adjoint`**(*emap*, *omap=None*, *nthread=0*, *normalize=True*)

`pixell.enmap.`**`ifft_adjoint`**(*emap*, *omap=None*, *nthread=0*, *normalize=True*)

`pixell.enmap.`**`idct_adjoint`**(*emap*, *omap=None*, *nthread=0*, *normalize=True*)

`pixell.enmap.`**`dct_adjoint`**(*emap*, *omap=None*, *nthread=0*, *normalize=True*)

`pixell.enmap.`**`map2harm`**(*emap*, *nthread=0*, *normalize=True*, *iau=False*, *spin=[0, 2]*, *adjoint_harm2map=False*)

Performs the 2d FFT of the enmap pixels, returning a complex enmap. If normalize starts with “phy” (for physical), then an additional normalization is applied such that the binned square of the fourier transform can be directly compared to theory (apart from mask corrections) , i.e., pixel area factors are corrected for.

`pixell.enmap.`**`harm2map`**(*emap*, *nthread=0*, *normalize=True*, *iau=False*, *spin=[0, 2]*, *keep_imag=False*, *adjoint_map2harm=False*)

`pixell.enmap.`**`map2harm_adjoint`**(*emap*, *nthread=0*, *normalize=True*, *iau=False*, *spin=[0, 2]*, *keep_imag=False*)

`pixell.enmap.`**`harm2map_adjoint`**(*emap*, *nthread=0*, *normalize=True*, *iau=False*, *spin=[0, 2]*)

`pixell.enmap.`**`queb_rotmat`**(*lmap*, *inverse=False*, *iau=False*, *spin=2*)

`pixell.enmap.`**`rotate_pol`**(*emap*, *angle*, *comps=[-2, -1]*, *spin=2*, *axis=-3*)

Rotate the polarization of the given enmap “emap” by angle (in radians) along the given components (the last two by default) of the given axis (the 3rd-last axis by default). In standard enmaps the 3rd-last axis is holds the Stokes components of the map in the order T, Q, U. The spin argument controls the spin, and defaults to spin-2. This function is flexible enough to work with non-enmaps too.

`pixell.enmap.`**`map_mul`**`(`*mat*, *vec*`)`

Elementwise matrix multiplication mat*vec. Result will have the same shape as vec. Multiplication happens along the last non-pixel indices.

`pixell.enmap.`**`smooth_gauss`**`(`*emap*, *sigma*`)`

Smooth the map given as the first argument with a gaussian beam with the given standard deviation sigma in radians. If sigma is negative, then the complement of the smoothed map will be returned instead (so it will be a highpass filter).

`pixell.enmap.`**`inpaint`**`(`*map*, *mask*, *method='nearest'*`)`

Inpaint regions in emap where mask==True based on the nearest unmasked pixels. Uses scipy.interpolate.griddata internally. See its documentation for the meaning of method. Note that if method is not "nearest", then areas where the mask touches the edge will be filled with NaN instead of sensible values.

The purpose of this function is mainly to allow inapinting bad values with a continuous signal with the right order of magnitude, for example to allow fourier operations of masked data with large values near the edge of the mask (e.g. a galactic mask). Its goal is not to inpaint with something realistic-looking. For that heavier methods are needed.

FIXME: This function is slow and not very good. Fix or remove.

`pixell.enmap.`**`calc_window`**`(`*shape*, *order=0*, *scale=1*`)`

Compute fourier-space pixel window function. Since the window function is separable, it is returned as an x and y part, such that window = wy[:,None]*wx[None,:]. By default the pixel window for interpolation order 0 mapmaking (nearest neighbor) is returned. Pass 1 for bilinear mapmaking's pixel window. The scale argument can be used to calculate the pixel window at non-native resolutions. For example, with scale=2 you will get the pixwin for a map with twice the resolution

`pixell.enmap.`**`apply_window`**`(`*emap*, *pow=1.0*, *order=0*, *scale=1*, *nofft=False*`)`

Apply the pixel window function to the specified power to the map, returning a modified copy. Use pow=-1 to unapply the pixel window. By default the pixel window for interpolation order 0 mapmaking (nearest neighbor) is applied. Pass 1 for bilinear mapmaking's pixel window.

`pixell.enmap.`**`unapply_window`**`(`*emap*, *pow=1.0*, *order=0*, *scale=1*, *nofft=False*`)`

The inverse of apply_window. Equivalent to just flipping the sign of the pow argument.

`pixell.enmap.`**`samewcs`**`(`*arr*, **args*`)`

Returns arr with the same wcs information as the first enmap among args. If no matches are found, arr is returned as is. Will reference, rather than copy, the underlying array data whenever possible.

`pixell.enmap.`**`geometry`**`(`*pos*, *res=None*, *shape=None*, *proj='car'*, *deg=False*, *pre=()*, *force=False*, *ref=None*, ***kwargs*`)`

Consruct a shape,wcs pair suitable for initializing enmaps. pos can be either a {dec,ra} center position or a [{from,to},{dec,ra}] array giving the bottom-left and top-right corners of the desired geometry. At least one of res or shape must be specified. If res is specified, it must either be a number, in which the same resolution is used in each direction, or {dec,ra}. If shape is specified, it must be at least [2]. All angles are given in radians.

The projection type is chosen with the proj argument. The default is "car", corresponding to the equirectangular plate carree projection. Other valid projections are "cea", "zea", "gnom", etc. See wcsutils for details.

By default the geometry is tweaked so that a standard position, typically ra=0,dec=0, would be at an integer logical pixel position (even if that position is outside the physical map). This makes it easier to generate maps that are compatible up to an integer pixel offset, as well as maps that are compatible with the predefined spherical harmonics transform ring weights. The cost of this tweaking is that the resulting corners can differ by a fraction of a pixel from the one requested. To force the geometry to exactly match the corners provided you can pass force=True. It is also possible to manually choose the reference point via the ref argument, which must be a dec,ra coordinate pair (in radians).

`pixell.enmap.`**`fullsky_geometry`**`(`*res=None*, *shape=None*, *dims=()*, *proj='car'*, *variant='CC'*`)`

Build an enmap covering the full sky, with the outermost pixel centers at the poles and wrap-around points. Only the car projection is supported for now, but the variants CC and fejer1 can be selected using the variant keyword. This currently defaults to CC, but will likely change to fejer1 in the future.

`pixell.enmap.`**`band_geometry`**`(`*dec_cut*, *res=None*, *shape=None*, *dims=()*, *proj='car'*, *variant='CC'*`)`

Return a geometry corresponding to a sky that had a full-sky geometry but to which a declination cut was applied. If dec_cut is a single number, the declination range will be (-dec_cut,dec_cut) radians, and if specified with two components, it is interpreted as (dec_cut_min,dec_cut_max). The remaining arguments are the same as fullsky_geometry and pertain to the geometry before cropping to the cut-sky.

`pixell.enmap.`**`thumbnail_geometry`**`(`*r=None*, *res=None*, *shape=None*, *dims=()*, *proj='tan'*`)`

Build a geometry in the given projection centered on (0,0), which will be exactly at a pixel center.

> r: The radius from the center to the edges of the patch, in radians. res: The resolution of the patch, in radians. shape: The target shape of the patch. Will be forced to odd numbers if necessary.

Any two out of these three arguments must be specified. The most common usage will probably be to specify r and res, e.g.

> shape, wcs = enmap.thumbnail_geometry(r=1*utils.degree, res=0.5*utils.arcmin)

The purpose of this function is to provide a geometry appropriate for object stacking, etc. Ideally enmap.geometry would do this, but this specialized function makes it easier to ensure that the center of the coordinate system will be at excactly the pixel index (y,x) = shape//2+1, which was a commonly requested feature (even though which pixel is at the center shouldn't really matter as long as one takes into account the actual coordinates of each pixel).

`pixell.enmap.`**`union_geometry`**`(`*geometries*`)`

Given a list of compatible geometries, return a new geometry that's the union if the inputs, in the sense that it contains all the pixels that the individual ones contain

`pixell.enmap.`**`create_wcs`**`(`*shape*, *box=None*, *proj='cea'*`)`

`pixell.enmap.`**`spec2flat`**`(`*shape*, *wcs*, *cov*, *exp=1.0*, *mode='constant'*, *oversample=1*, *smooth='auto'*`)`

Given a (ncomp,ncomp,l) power spectrum, expand it to harmonic map space, returning (ncomp,ncomp,y,x). This involves a rescaling which converts from power in terms of multipoles, to power in terms of 2d frequency. The optional exp argument controls the exponent of the rescaling factor. To use this with the inverse power spectrum, pass exp=-1, for example. If apply_exp is True, the power spectrum will be taken to the exp'th power. Otherwise, it is assumed that this has already been done, and the exp argument only controls the normalization of the result.

It is irritating that this function needs to know what kind of matrix it is expanding, but I can't see a way to avoid it. Changing the units of harmonic space is not sufficient, as the following demonstrates:

> m = harm2map(map_mul(spec2flat(s, b, multi_pow(ps, 0.5), 0.5), map2harm(rand_gauss(s,b))))

The map m is independent of the units of harmonic space, and will be wrong unless the spectrum is properly scaled. Since this scaling depends on the shape of the map, this is the appropriate place to do so, ugly as it is.

`pixell.enmap.`**`spec2flat_corr`**`(`*shape*, *wcs*, *cov*, *exp=1.0*, *mode='constant'*`)`

`pixell.enmap.`**`smooth_spectrum`**`(`*ps*, *kernel='gauss'*, *weight='mode'*, *width=1.0*`)`

Smooth the spectrum ps with the given kernel, using the given weighting.

`pixell.enmap.`**`calc_ps2d`**`(`*harm*, *harm2=None*`)`

Compute the 2d power spectrum of the harmonic-space enmap "harm", as output by map2harm. Use map2harm with norm="phys" to get physical units in this spectrum. If harm2 is specified, then the cross-spectrum between harm and harm2 is computed instead.

Some example usage, where the notation a[{x,y,z},n,m] specifies that the array a has shape [3,n,m], and the 3 entries in the first axis should be interpreted as x, y and z respectively.

1. cl[nl] = calc_ps2d(harm[ny,nx]) This just computes the standard power spectrum of the given harm, resulting in a single 2d enmap.
2. cl[nl] = calc_ps2d(harm1[ny,nx], harm2[ny,nx]) This compues the 1d cross-spectrum between the 2d enmaps harm1 and harm2.
3. cl[{T,E,B},{T,E,B},nl] = calc_ps2d(harm[{T,E,B},None,ny,nx], harm[None,{T,E,B},ny,nx]) This computes the 3x3 polarization auto-spectrum for a 3d polarized harmonic enmap.
4. cl[{T,E,B},{T,E,B},nl] = calc_ps2d(harm1[{T,E,B},None,ny,nx], harm2[None,{T,E,B},ny,nx]) As above, but gives the 3x3 polarization cross-spectrum between two 3d harmonic enmaps.

The output is in the shape one would expect from numpy broadcasting. For example, in the last example, the TE power spectrum would be found in cl[0,1], and the ET power spectrum (which is different for the cross-spectrum case) is in cl[1,0].

`pixell.enmap.`**`multi_pow`**(*mat*, *exp*, *axes=[0, 1]*)

Raise each sub-matrix of mat (ncomp,ncomp,…) to the given exponent in eigen-space.

`pixell.enmap.`**`get_downgrade_offset`**(*shape*, *wcs*, *factor*, *ref=None*)

Get the pixel offset required to keep a map downgraded by the given factor aligned with the reference point.

`pixell.enmap.`**`downgrade`**(*emap*, *factor*, *op=<function mean>*, *ref=None*, *off=None*, *inclusive=False*)

Returns enmap "emap" downgraded by the given integer factor (may be a list for each direction, or just a number) by averaging inside pixels.

`pixell.enmap.`**`upgrade`**(*emap*, *factor*, *off=None*, *oshape=None*, *inclusive=False*)

Upgrade emap to a larger size using nearest neighbor interpolation, returning the result. More advanced interpolation can be had using enmap.interpolate.

`pixell.enmap.`**`downgrade_geometry`**(*shape*, *wcs*, *factor*)

Returns the oshape, owcs corresponding to a map with geometry shape, wcs that has been downgraded by the given factor. Similar to scale_geometry, but truncates the same way as downgrade, and only supports integer factors.

`pixell.enmap.`**`upgrade_geometry`**(*shape*, *wcs*, *factor*)

`pixell.enmap.`**`distance_transform`**(*mask*, *omap=None*, *rmax=None*, *method='cellgrid'*)

Given a boolean mask, produce an output map where the value in each pixel is the distance to the closest false pixel in the mask. See distance_from for the meaning of rmax.

`pixell.enmap.`**`labeled_distance_transform`**(*labels*, *omap=None*, *odomains=None*, *rmax=None*, *method='cellgrid'*)

Given a map of labels going from 1 to nlabel, produce an output map where the value in each pixel is the distance to the closest nonzero pixel in the labels, as well as a map of which label each pixel was closest to. See distance_from for the meaning of rmax.

`pixell.enmap.`**`distance_from`**(*shape*, *wcs*, *points*, *omap=None*, *odomains=None*, *domains=False*, *method='cellgrid'*, *rmax=None*, *step=1024*)

Find the distance from each pixel in the geometry (shape, wcs) to the nearest of the points[{dec,ra},npoint], returning a [ny,nx] map of distances. If domains==True, then it will also return a [ny,nx] map of the index of the point that was closest to each pixel. If rmax is specified and the method is "cellgrid" or "bubble", then distances will only be computed up to rmax. Beyond that distance will be set to rmax and domains to -1. This can be used to speed up the calculation when one only cares about nearby areas.

`pixell.enmap.`**`distance_transform_healpix`**(*mask*, *omap=None*, *rmax=None*, *method='heap'*)

Given a boolean healpix mask, produce an output map where the value in each pixel is the distance to the closest false pixel in the mask. See distance_from for the meaning of rmax.

pixell.enmap.**labeled_distance_transform_healpix**(*labels*, *omap=None*, *odomains=None*, *rmax=None*, *method='heap'*)

Given a healpix map of labels going from 1 to nlabel, produce an output map where the value in each pixel is the distance to the closest nonzero pixel in the labels, as well as a map of which label each pixel was closest to. See distance_from for the meaning of rmax.

pixell.enmap.**distance_from_healpix**(*nside*, *points*, *omap=None*, *odomains=None*, *domains=False*, *rmax=None*, *method='bubble'*)

Find the distance from each pixel in healpix map with nside nside to the nearest of the points[{dec,ra},npoint], returning a [ny,nx] map of distances. If domains==True, then it will also return a [ny,nx] map of the index of the point that was closest to each pixel. If rmax is specified, then distances will only be computed up to rmax. Beyond that distance will be set to rmax and domains to -1. This can be used to speed up the calculation when one only cares about nearby areas.

pixell.enmap.**grow_mask**(*mask*, *r*)

Grow the True part of boolean mask "mask" by a distance of r radians

pixell.enmap.**shrink_mask**(*mask*, *r*)

Shrink the True part of boolean mask "mask" by a distance of r radians

pixell.enmap.**pad**(*emap*, *pix*, *return_slice=False*, *wrap=False*)

Pad enmap "emap", creating a larger map with zeros filled in on the sides. How much to pad is controlled via pix, which har format [{from,to},{y,x}], [{y,x}] or just a single number to apply on all sides. E.g. pix=5 would pad by 5 on all sides, and pix=[[1,2],[3,4]] would pad by 1 on the bottom, 2 on the left, 3 on the top and 4 on the right.

pixell.enmap.**find_blank_edges**(*m*, *value='auto'*)

Returns blanks[{front,back},{y,x}], the size of the blank area at the beginning and end of each axis of the map, where the argument "value" determines which value is considered blank. Can be a float value, or the strings "auto" or "none". Auto will choose the value that maximizes the edge area considered blank. None will result in nothing being consideered blank.

pixell.enmap.**autocrop**(*m*, *method='plain'*, *value='auto'*, *margin=0*, *factors=None*, *return_info=False*)

Adjust the size of m to be more fft-friendly. If possible, blank areas at the edge of the map are cropped to bring us to a nice length. If there there aren't enough blank areas, the map is padded instead. If value="none" no values are considered blank, so no cropping will happen. This can be used to autopad for fourier-friendliness.

pixell.enmap.**padcrop**(*m*, *info*)

pixell.enmap.**grad**(*m*)

Returns the gradient of the map m as [2,...].

pixell.enmap.**grad_pix**(*m*)

The gradient of map m expressed in units of pixels. Not the same as the gradient of m with resepect to pixels. Useful for avoiding sky2pix-calls for e.g. lensing, and removes the complication of axes that increase in nonstandard directions.

pixell.enmap.**div**(*m*)

Returns the divergence of the map m[2,...] as [...].

pixell.enmap.**laplace**(*m*)

pixell.enmap.**apod**(*m*, *width*, *profile='cos'*, *fill='zero'*)

Apodize the provided map. Currently only cosine apodization is
implemented.

Parameters

- `imap` – (. . . ,Ny,Nx) or (Ny,Nx) ndarray to be apodized
- `width` – The width in pixels of the apodization on each edge.
- `profile` – The shape of the apodization. Only "cos" is supported.

`pixell.enmap.`**`apod_profile_lin`**`(`*x*`)`

`pixell.enmap.`**`apod_profile_cos`**`(`*x*`)`

`pixell.enmap.`**`apod_mask`**`(`*mask*, *width=0.017453292519943295*, *edge=True*, *profile=<function apod_profile_cos>*`)`

Given an enmap mask that's 0 in bad regions and 1 in good regions, return an apodization map that's still 0 in bad regions, but transitions smoothly to 1 in the good region over the given width in radians. The transition profile is controlled by the profile argument. Regions outside the image are considered to be bad.

`pixell.enmap.`**`lform`**`(`*map*, *shift=True*`)`

Given an enmap, return a new enmap that has been fftshifted (unless shift=False), and which has had the wcs replaced by one describing fourier space. This is mostly useful for plotting or writing 2d power spectra.

It could have been useful more generally, but because all "plain" coordinate systems are assumed to need conversion between degrees and radians, sky2pix etc. get confused when applied to lform-maps.

`pixell.enmap.`**`lwcs`**`(`*shape*, *wcs*`)`

Build world coordinate system for l-space

`pixell.enmap.`**`rbin`**`(`*map*, *center=[0, 0]*, *bsize=None*, *brel=1.0*, *return_nhit=False*, *return_bins=False*`)`

Radially bin map around the given center point ([0,0] by default). If bsize it given it will be the constant bin width. This defaults to the pixel size. brel can be used to scale up the bin size. This is mostly useful when using automatic bsize.

Returns bvals[. . . ,nbin], r[nbin], where bvals is the mean of the map in each radial bin and r is the mid-point of each bin

`pixell.enmap.`**`lbin`**`(`*map*, *bsize=None*, *brel=1.0*, *return_nhit=False*, *return_bins=False*`)`

Like rbin, but for fourier space. Returns b(l),l

`pixell.enmap.`**`radial_average`**`(`*map*, *center=[0, 0]*, *step=1.0*`)`

`pixell.enmap.`**`padslice`**`(`*map*, *box*, *default=nan*`)`

Equivalent to map[. . . ,box[0,0]:box[1,0],box[0,1]:box[1,1]], except that pixels outside the map are treated as actually being present, but filled with the value given by "default". Hence, ther esult will always have size box[1]-box[0].

`pixell.enmap.`**`tile_maps`**`(`*maps*`)`

Given a 2d list of enmaps representing contiguous tiles in the same global pixelization, stack them into a total map and return it. E.g. if maps = [[a,b],[c,d]], then the result would be

c d

map = a b

`pixell.enmap.`**`stamps`**`(`*map*, *pos*, *shape*, *aslist=False*`)`

Given a map, extract a set of identically shaped postage stamps with corners at pos[ntile,2]. The result will be an enmap with shape [ntile,. . . ,ny,nx] and a wcs appropriate for the *first* tile only. If that is not the behavior wanted, you can specify aslist=True, in which case the result will be a list of enmaps, each with the correct wcs.

`pixell.enmap.`**`to_healpix`**`(`*imap*, *omap=None*, *nside=0*, *order=3*, *chunk=100000*, *destroy_input=False*`)`

Project the enmap "imap" onto the healpix pixelization. If omap is given, the output will be written to it. Otherwise, a new healpix map will be constructed. The healpix map must be in RING order. nside controls the resolution of the output map. If 0, nside is chosen such that the output map is higher resolution than the input. This is needed to avoid losing information. To go to a lower-resolution output map, you should first degrade the input map. The chunk argument affects the speed/memory tradeoff of the function. Higher values use more memory, and might (and might not) give higher speed. If destroy_input is True, then the input map will be prefiltered in-place, which saves memory but modifies its values.

`pixell.enmap.`**`to_flipper`**`(`*imap*, *omap=None*, *unpack=True*`)`

Convert the enmap "imap" into a flipper map with the same geometry. If omap is given, the output will be written to it. Otherwise, a an array of flipper maps will be constructed. If the input map has dimensions [a,b,c,ny,nx], then the output will be an [a,b,c] array with elements that are flipper maps with dimension [ny,nx]. The exception is for a 2d enmap, which is returned as a plain flipper map, not a 0-dimensional array of flipper maps. To avoid this unpacking, pass

Flipper needs cdelt0 to be in decreasing order. This function ensures that, at the cost of losing the original orientation. Hence to_flipper followed by from_flipper does not give back an exactly identical map to the one on started with.

`pixell.enmap.`**`from_flipper`**`(`*imap*, *omap=None*`)`

Construct an enmap from a flipper map or array of flipper maps imap. If omap is specified, it must have the correct shape, and the data will be written there.

`pixell.enmap.`**`write_map`**`(`*fname*, *emap*, *fmt=None*, *address=None*, *extra={}*, *allow_modify=False*`)`

Writes an enmap to file. If fmt is not passed, the file type is inferred from the file extension, and can be either fits or hdf. This can be overriden by passing fmt with either 'fits' or 'hdf' as argument.

The other arguments are passed to write_fits and/or write_hdf.

`pixell.enmap.`**`read_map`**`(`*fname*, *fmt=None*, *sel=None*, *box=None*, *pixbox=None*, *geometry=None*, *wrap='auto'*, *mode=None*, *sel_threshold=10000000.0*, *wcs=None*, *hdu=None*, *delayed=False*, *verbose=False*, *address=None*`)`

Read an enmap from file. The file type is inferred from the file extension, unless fmt is passed. fmt must be one of 'fits' and 'hdf'.

The sel, box, pixbox, geometry, wrap, mode, and delayed arguments are all used by read_helper to (optionally) select a subregion of the map or change how it is wrapped on the sky.

The hdu and verbose arguments are only used for FITS (see read_fits). The address argument is only used for HDF (see read_hdf).

`pixell.enmap.`**`read_map_geometry`**`(`*fname*, *fmt=None*, *hdu=None*, *address=None*`)`

Read an enmap geometry from file. The file type is inferred from the file extension, unless fmt is passed. fmt must be one of 'fits' and 'hdf'.

`pixell.enmap.`**`write_map_geometry`**`(`*fname*, *shape*, *wcs*, *fmt=None*`)`

Write an enmap geometry to file. The file type is inferred from the file extension, unless fmt is passed. fmt must be one of 'fits' and 'hdf'. Only fits is supported for now, though.

`pixell.enmap.`**`write_fits`**`(`*fname*, *emap*, *extra={}*, *allow_modify=False*`)`

Write an enmap to a fits file.

`pixell.enmap.`**`write_fits_geometry`**`(`*fname*, *shape*, *wcs*`)`

Write just the geometry to a fits file that will only contain the header `pixell.enmap.`**`read_fits`**(*fname*, *hdu=None*, *sel=None*, *box=None*, *pixbox=None*, *geometry=None*, *wrap='auto'*, *mode=None*, *sel_threshold=10000000.0*, *wcs=None*, *delayed=False*, *verbose=False*)

Read an enmap from the specified fits file. By default, the map and coordinate system will be read from HDU 0. Use the hdu argument to change this. The map must be stored as a fits image. If sel is specified, it should be a slice that will be applied to the image before reading. This avoids reading more of the image than necessary. Instead of sel, a coordinate box [[yfrom,xfrom],[yto,xto]] can be specified.

`pixell.enmap.`**`read_fits_geometry`**(*fname*, *hdu=None*, *quick=True*)

Read an enmap wcs from the specified fits file. By default, the map and coordinate system will be read from HDU 0. Use the hdu argument to change this. The map must be stored as a fits image.

`pixell.enmap.`**`write_hdf`**(*fname*, *emap*, *address=None*, *extra={}*)

Write an enmap as an hdf file, preserving all the WCS metadata.

Parameters

- **fname** (*str or h5py.Group*) – Filename or open h5py handle.
- **emap** (ndmap) – Object to store.
- **address** (*str*) – Group address within the HDF file to place the result. If None, the data are written at root level after truncating the file.
- **extra** (*dict*) – additional data to write into the output.

Notes

If address is None, the output file will be replaced if it exists. If address is a string, and the target file exists, the file will not be reset but anything living at that address will be replaced with the encoded emap.

`pixell.enmap.`**`read_hdf`**(*fname*, *hdu=None*, *sel=None*, *box=None*, *pixbox=None*, *geometry=None*, *wrap='auto'*, *mode=None*, *sel_threshold=10000000.0*, *wcs=None*, *delayed=False*, *address=None*)

Read an enmap from the specified hdf file. Two formats are supported. The old enmap format, which simply used a bounding box to specify the coordinates, and the new format, which uses WCS properties. The latter is used if available. With the old format, plate carree projection is assumed. Note: some of the old files have a slightly buggy wcs, which can result in 1-pixel errors.

If address is a string, the map will be loaded from that group address within fname.

Note fname can be passed in as an h5py.Group (e.g. an open h5py.File) instead of a string, and the map will be read from that handle.

`pixell.enmap.`**`read_hdf_geometry`**(*fname*, *address=None*)

Read an enmap wcs from the specified hdf file.

`pixell.enmap.`**`read_npy`**(*fname*, *hdu=None*, *sel=None*, *box=None*, *pixbox=None*, *geometry=None*, *wrap='auto'*, *mode=None*, *sel_threshold=10000000.0*, *wcs=None*, *delayed=False*, *address=None*)

Read an enmap from the specified npy file. Only minimal support. No wcs information.

`pixell.enmap.`**`fix_python3`**(*s*)

Convert “bytes” to string in python3, while leaving other types unmolested. Python3 string handling is stupid.

`pixell.enmap.`**`read_helper`**(*data*, *sel=None*, *box=None*, *pixbox=None*, *geometry=None*, *wrap='auto'*, *mode=None*, *delayed=False*)

Helper function for map reading. Handles the slicing, sky-wrapping and capping, etc.

`class` `pixell.enmap.`**`ndmap_proxy`**(*shape*, *wcs*, *dtype*, *fname='<none>'*, *threshold=10000000.0*) property **ndim**

property **geometry**

property **npix**

submap(*box*, *mode=None*, *wrap='auto'*)

stamps(*pos*, *shape*, *aslist=False*)

area()

box(*npoint=10*, *corner=True*)

center()

distance_from(*points*, *omap=None*, *odomains=None*, *domains=False*, *method='cellgrid'*, *rmax=None*, *step=1024*)

extent(*method='auto'*, *signed=False*)

extract(*shape*, *wcs*, *omap=None*, *wrap='auto'*, *op=<function ndmap.<lambda>>*, *cval=0*, *iwcs=None*, *reverse=False*)

extract_pixbox(*pixbox*, *omap=None*, *wrap='auto'*, *op=<function ndmap.<lambda>>*, *cval=0*, *iwcs=None*, *reverse=False*)

lmap(*oversample=1*)

modlmap(*oversample=1*, *min=0*)

modrmap(*ref='center'*, *safe=True*, *corner=False*)

pix2sky(*pix*, *safe=True*, *corner=False*)

pixbox_of(*oshape*, *owcs*)

pixmap()

pixshape(*signed=False*)

pixshapemap(*separable='auto'*, *signed=False*)

pixsize()

pixsizemap(*separable='auto'*, *broadcastable=False*)

posmap(*safe=True*, *corner=False*, *separable='auto'*, *dtype=<class 'numpy.float64'>*)

sky2pix(*coords*, *safe=True*, *corner=False*)

class pixell.enmap.**ndmap_proxy_fits**(*hdu*, *wcs*, *fname='<none>'*, *threshold=10000000.0*, *verbose=False*)

class pixell.enmap.**ndmap_proxy_hdf**(*dset*, *wcs*, *fname='<none>'*, *threshold=10000000.0*)

pixell.enmap.**fix_endian**(*map*)

Make endianness of array map match the current machine. Returns the result.

`pixell.enmap.`**`get_stokes_flips`**`(`*hdu*`)`

Given a FITS HDU, parse its header to determine which, if any, axes need to have their sign flip to get them in the COSMO polarization convention. Returns an array of length ndim, with each entry being the index of the axis that should be flipped, or -1 if none should be flipped.

`pixell.enmap.`**`shift`**`(`*map*, *off*, *inplace=False*, *keepwcs=False*`)`

Cyclicly shift the pixels in map such that a pixel at position (i,j) ends up at position (i+off[0],j+off[1])

`pixell.enmap.`**`fractional_shift`**`(`*map*, *off*, *keepwcs=False*, *nofft=False*`)`

Shift map cyclically by a non-integer amount off [{y_off,x_off}]

`pixell.enmap.`**`fftshift`**`(`*map*, *inplace=False*`)`

`pixell.enmap.`**`ifftshift`**`(`*map*, *inplace=False*`)`

`pixell.enmap.`**`fillbad`**`(`*map*, *val=0*, *inplace=False*`)`

`pixell.enmap.`**`resample`**`(`*map*, *oshape*, *off=(0, 0)*, *method='fft'*, *mode='wrap'*, *corner=False*, *order=3*`)`

Resample the input map such that it covers the same area of the sky with a different number of pixels given by oshape.

`pixell.enmap.`**`resample_fft`**`(`*fimap*, *oshape*, *fomap=None*, *off=(0, 0)*, *corner=False*, *norm='pix'*, *op=<function <lambda>>*, *dummy=False*`)`

Like resample, but takes a fourier-space map as input and outputs a fourier-space map. unit specifies which fourier-space unit is used. "pix" corresponds to the standard enmap normalization (normalize=True in enmap.fft). "phys" corresponds to physical normalization (normalize="phys"). The fourier-units matter because some fourier-space units need rescaline when going from one resolution to another.

`pixell.enmap.`**`spin_helper`**`(`*spin*, *n*`)`

`pixell.enmap.`**`spin_pre_helper`**`(`*spin*, *pre*`)`

Like spin_helper, but also handles looping over pre-dimensions

`pixell.enmap.`**`padtiles`**`(`**maps*, *tshape=600*, *pad=60*, *margin=60*, *mode='auto'*, *start=0*, *step=1*`)`

Iterate over padded tiles in one or more maps. The tiling will have a logical tile shape of tshape, but each yielded tile will be expanded with some data from its neighbors. The extra area consists of two parts: The padding and the margin. For a read-iterator these are equivalent, but for a write-iterator the margin will be ignored (and so can be used for apodization), while the padding will be used for crossfading when mergin the tiles together.

Typical usage:

for itile, otile in padtiles(imap, imap, margin=60):
itile = apod(itile, 60) otile[:] = some_filter(itile)

This would iterate over tiles of imap and omap, with the default padding and a margin of 60 pixels. The margin region is used for apodization, and some filter is then applied to the tile, writing the result to the output tile. Note the use of [:] to actually write to otile instead of just rebinding the variable name!

It's also possible to iterate over fewer or more maps at once. See the "mode" argument.

If the tile shape does not evenly divide the map shape, then the last tile in each row and column will extend beyond the edge of the map. These pixels will be treated as enmap.extract does, with the potential of sky wrapping. Warning: Write-iterators for a map that goes all the way around the sky while the tile shape does not divide the map shape will incorrectly weight the wrapped tiles, so avoid this.

Parameters

- **`*maps`** (***) – The maps to iterate over. Must have the same pixel dimensions.
- **`tshape`** (***) – The tile shape. Either an integer or a (yshape,xshape) tuple. Default: 600 pixels.

- **pad** (*) – The padding. Either an integer or a (ypad,xpad) tuple. Used to implement context and crossfading. Cannot be larger than half of tshape. Default: 60 pixels.
- **margin** (*) – The margin size. Either an integer or a (ymargin,xmargin) tuple. Ignored in write-iterators, so suitable for apodization. Default 60 pixels.
- **mode** (*) – Specifies which maps should be read-iterated vs. write-iterated. A read-iterated map will yield padded tiles from the corresponding map. Writes to these tiles are discarded. A write-iterated map yields zeroed tiles of the same shape as the read-iterator. Writes to these tiles are used to update the corresponding map, including crossfading the overlapping regions (due to the padding) such that there aren't any sharp tile boundaries in the output map. mode can be either "auto" or a string of the same length as maps consisting of "r" and "w" characters. If the nth character is r/w then the corresponding map will be read/write-iterated. If the string is "auto", then the last map will be output-iterated and all the others input-iterated, unless there's only a single map in which case it will be input- iterated. Default: "auto".
- **start** (*) – Flattened tile offset to start at. Useful for mpi loops. Default: 0.
- **step** (*) – Flattened tile stride. Useful for mpi loops. Default: 1

class pixell.enmap.**Padtiler**(*tshape=600*, *pad=60*, *margin=60*, *start=0*, *step=1*)

Helper class used to implement padtiles. See its docstring for details.

read(*imap*)

write(*omap*)

3.2 2.2 fft - Fourier transforms

This is a convenience wrapper of pyfftw.

class pixell.fft.**numpy_FFTW**(*a*, *b*, *axes=-1*, *direction='FFTW_FORWARD'*, **args*, ***kwargs*)

Minimal wrapper of numpy in order to be able to provide it as an engine. Not a full-blown interface.

class pixell.fft.**ducc_FFTW**(*a*, *b*, *axes=(-1,)*, *direction='FFTW_FORWARD'*, *threads=1*, **args*, ***kwargs*)

Minimal wrapper of ducc in order to be able to provide it as an engine. Not a full-blown interface.

do_dct(*kind*, **args*, ***kwargs*)

pixell.fft.**numpy_empty_aligned**(*shape*, *dtype*, *n=None*)

This dummy function just skips the alignment, since numpy doesn't provide an easy way to get it.

class pixell.fft.**NumpyEngine**

class pixell.fft.**DuccEngine**

pixell.fft.**set_engine**(*eng*)

pixell.fft.**get_engine**(*eng*)

pixell.fft.**fft**(*tod*, *ft=None*, *nthread=0*, *axes=[-1]*, *flags=None*, *_direction='FFTW_FORWARD'*, *engine='auto'*)

Compute discrete fourier transform of tod, and store it in ft. What transform to do (real or complex, number of dimension etc.) is determined from the size and type of tod and ft. If ft is left out, a complex transform is assumed. The optional nthread argument specifies the number of theads to use in the fft. The default (0) uses the value specified by the OMP_NUM_THREAD environment varible if that is specified, or the total number of cores on the computer otherwise.

`pixell.fft.`**`ifft`**(*ft*, *tod=None*, *nthread=0*, *normalize=False*, *axes=[-1]*, *flags=None*, *engine='auto'*)

Compute inverse discrete fourier transform of ft, and store it in tod. What transform to do (real or complex, number of dimension etc.) is determined from the size and type of tod and ft. The optional nthread argument specifies the number of theads to use in the fft. The default (0) uses the value specified by the OMP_NUM_THREAD environment varible if that is specified, or the total number of cores on the computer otherwise. By default this is not normalized, meaning that fft followed by ifft will multiply the data by the length of the transform. By specifying the normalize argument, you can turn normalization on, though the normalization step will not use paralellization.

`pixell.fft.`**`rfft`**(*tod*, *ft=None*, *nthread=0*, *axes=[-1]*, *flags=None*, *engine='auto'*)

Equivalent to fft, except that if ft is not passed, it is allocated with appropriate shape and data type for a real-to-complex transform.

`pixell.fft.`**`irfft`**(*ft*, *tod=None*, *n=None*, *nthread=0*, *normalize=False*, *axes=[-1]*, *flags=None*, *engine='auto'*)

Equivalent to ifft, except that if tod is not passed, it is allocated with appropriate shape and data type for a complex-to-real transform. If n is specified, that is used as the length of the last transform axis of the output array. Otherwise, the length of this axis is computed assuming an even original array.

`pixell.fft.`**`dct`**(*tod*, *dt=None*, *nthread=0*, *normalize=False*, *axes=[-1]*, *flags=None*, *type='DCT-I'*, *engine='auto'*)

Compute discrete cosine transform of tod, and store it in dt. By default it will do a DCT-I trasnform, but this can be controlled with the type argument. Even the much less common discrete sine transforms are avialble by passing e.g. type="DST-I". Valid values are DCT-I, DCT-II, DCT-III, DCT-IV, DST-I, DST-II, DST-III and DST-IV, or the raw FFTW names the correspond to (e.g. FFTW_REDFT00). If dt is not passed, it will be allocated with the same shape and data type as tod.

The optional nthread argument specifies the number of theads to use in the fft. The default (0) uses the value specified by the OMP_NUM_THREAD environment varible if that is specified, or the total number of cores on the computer otherwise.

Note that DCTs and DSTs were only added to pyfftw in version 13.0. The function will fail with an Invalid scheme error for older versions.

`pixell.fft.`**`idct`**(*dt*, *tod=None*, *nthread=0*, *normalize=False*, *axes=[-1]*, *flags=None*, *type='DCT-I'*, *engine='auto'*)

Compute the inverse discrete cosine transform of dt, and store it in tod. By default it will do the inverse of a DCT-I trasnform, but this can be controlled with the type argument. Even the much less common discrete sine transforms are avialble by passing e.g. type="DST-I". Valid values are DCT-I, DCT-II, DCT-III, DCT-IV, DST-I, DST-II, DST-III and DST-IV, or the raw FFTW names the correspond to (e.g. FFTW_REDFT00). If tod is not passed, it will be allocated with the same shape and data type as tod.

By the default an unnormalized transform is performed. Pass normalize=True to get an actual inverse transform. This divides by a factor of 2*N+d for each axis the transform is performed along, where N is the length of the axis and d is -1 for DCT-1, +1 for DST-I and 0 for all the others. Usually it's faster to compute this factor once and combine it with other scalar factors in your math than to let this function do it, which is why it's turned off by default.

Note that this function already takes care of figuring out which transform is the appropriate inverse. E.g. the inverse of b = dct(a, type="DCT-II") is idct(b, type="DCT-II", normalize=True), not idct(b, type="DCT-III", normalize=True) even though DCT-III is the inverse of DCT-II.

The optional nthread argument specifies the number of theads to use in the fft. The default (0) uses the value specified by the OMP_NUM_THREAD environment varible if that is specified, or the total number of cores on the computer otherwise.

Note that DCTs and DSTs were only added to pyfftw in version 13.0. The function will fail with an Invalid scheme error for older versions.

`pixell.fft.`**`redft00`**`(`*a*, *b=None*, *nthread=0*, *normalize=False*, *flags=None*, *engine='auto'*`)`

Old brute-force work-around for missing dcts in pyfftw. Can be removed when newer versions of pyfftw become common. It's not very fast, sadly - about 5 times slower than an rfft. Transforms along the last axis.

`pixell.fft.`**`chebt`**`(`*a*, *b=None*, *nthread=0*, *flags=None*, *engine='auto'*`)`

The chebyshev transform of a, along its last dimension.

`pixell.fft.`**`ichebt`**`(`*a*, *b=None*, *nthread=0*, *engine='auto'*`)`

The inverse chebyshev transform of a, along its last dimension.

`pixell.fft.`**`fft_len`**`(`*n*, *direction='below'*, *factors=None*`)`

`pixell.fft.`**`asfcarray`**`(`*a*`)`

`pixell.fft.`**`empty`**`(`*shape*, *dtype*`)`

`pixell.fft.`**`fftfreq`**`(`*n*, *d=1.0*`)`

`pixell.fft.`**`rfftfreq`**`(`*n*, *d=1.0*`)`

`pixell.fft.`**`ind2freq`**`(`*n*, *i*, *d=1.0*`)`

`pixell.fft.`**`int2rfreq`**`(`*n*, *i*, *d=1.0*`)`

`pixell.fft.`**`freq2ind`**`(`*n*, *f*, *d=1.0*`)`

`pixell.fft.`**`rfreq2ind`**`(`*n*, *f*, *d=1.0*`)`

`pixell.fft.`**`shift`**`(`*a*, *shift*, *axes=None*, *nofft=False*, *deriv=None*, *engine='auto'*`)`

Shift the array a by a (possibly fractional) number of samples "shift" to the right, along the specified axis, which defaults to the last one. shift can also be an array, in which case multiple axes are shifted together.

`pixell.fft.`**`resample_fft`**`(`*fa*, *n*, *out=None*, *axes=-1*, *norm=1*, *op=<function <lambda>>*`)`

Given array fa[{dims}] which is the fourier transform of some array a, transform it so that that it corresponds to the fourier transform of a version of a with a different number of samples by padding or truncating the fourier space. The argument n controls the new number of samples. By default this is for the last axis, but this can be changed using the axes argument. Multiple axes can be resampled at once by specifying a tuple for axes and n.

The resulting array is multiplied by the argument norm. This can be used for normalization purposes. If norm is 1, then the multiplication is skipped.

The argument out can be used to specify an already allocated output array. If it is None (the default), then an array will be allocated automatically. Normally the output array is overwritten, but this can be controlled using the op argument, which should be a function (out,fa)->out

`pixell.fft.`**`fft_flat`**`(`*tod*, *ft*, *nthread=1*, *axes=[-1]*, *flags=None*, *_direction='FFTW_FORWARD'*`)`

Workaround for intel FFTW wrapper. Flattens appropriate dimensions of intput and output arrays to avoid crash that otherwise happens for arrays with ndim > N + 1, where N is the dimension of the transform. If 'axes' correspond to the last dimensions of the arrays, the workaround is essentially free. If *axes* correspond to other axes, copies are made when reshaping the arrays.

`pixell.fft.`**`ifft_flat`**`(`*ft*, *tod*, *nthread=1*, *axes=[-1]*, *flags=None*`)`

Same workaround as fft_flat but now for the inverse transform.

3.3 2.3 curvedsky - Curved-sky harmonic transforms

This module provides functions for taking into account the curvature of the full sky.

exception `pixell.curvedsky.`**`ShapeError`**

`pixell.curvedsky.`**`rand_map`**(*shape, wcs, ps, lmax=None, dtype=<class 'numpy.float64'>, seed=None, spin=[0, 2], method='auto', verbose=False*)

Generates a CMB realization with the given power spectrum for an enmap with the specified shape and WCS. This is identical to enlib.rand_map, except that it takes into account the curvature of the full sky. This makes it much slower and more memory-intensive. The map should not cross the poles.

`pixell.curvedsky.`**`pad_spectrum`**(*ps*, *lmax*)

`pixell.curvedsky.`**`rand_alm_healpy`**(*ps*, *lmax=None*, *seed=None*, *dtype=<class 'numpy.complex128'>*)

`pixell.curvedsky.`**`rand_alm`**(*ps*, *ainfo=None*, *lmax=None*, *seed=None*, *dtype=<class 'numpy.complex128'>*, *m_major=True*, *return_ainfo=False*)

This is a replacement for healpy.synalm. It generates the random numbers in l-major order before transposing to m-major order in order to allow generation of low-res and high-res maps that agree on large scales. It uses 2/3 of the memory of healpy.synalm, and has comparable speed.

`pixell.curvedsky.`**`alm2map`**(*alm*, *map*, *spin=[0, 2]*, *deriv=False*, *adjoint=False*, *copy=False*, *method='auto'*, *ainfo=None*, *verbose=False*, *nthread=None*, *epsilon=1e-06*, *pix_tol=1e-06*, *locinfo=None*, *tweak=False*)

Spherical harmonics synthesis. Transform from harmonic space to real space.

Parameters

- **`alm`**(*`complex64 or complex128 numpy array with shape [...,ncomp,nelem],`*) – [ncomp,nelem] or [nelem]. Spin transforms will be applied to the ncomp axis, controlled by the spin argument below.
- **`map`** (*`float32 or float64 enmap with shape [...,ncomp,ny,nx], [ncomp, ny,nx]`*) – or [ny,nx]. All but last two dimensions must match alm. Will be overwritten unless copy is True
- **`Options`** –
- **`-------`** –
- **`spin`** (*`list of spins. These describe how to handle the [ncomp] axis.`*) – 0: scalar transform. Consumes one element in the component axis not 0: spin transform. Consumes two elements from the component axis. For example, if you have a TEB alm [3,nelem] and want to transform it to a TQU map [3,ny,nx], you would use spin=[0,2] to perform a scalar transform for the T component and a spin-2 transform for the Q,U components. Another example. If you had an alm [5,nelem] and map [5,ny,nx] and the first element was scalar, the next pair spin-1 and the next pair spin-2, you woudl use spin=[0,1,2]. default:[0,2]
- **`deriv`** (*`If true, instead calculates the d/ddec and d/dra derivatives`*) – of the map corresponding to the alms. In this case the alm must have shape […,nelem] or [nelem] and the map must have shape […,2,ny,nx] or [2,ny,nx]. default: False
- **`adjoint`** (*`If true, instead calculates the adjoint of the`*) – alm2map operation. This reads from map and writes to alm. default: False
- **`copy`** (*`If true, writes to a copy of map instead of overwriting the`*) – map argument. The resulting map is returned.
- **`method`** (*`Select the spherical harmonics transform method:`*) –

"2d": Use ducc's "2d" transforms. These are fast and accurate, but
require full-sky CAR maps with one of a limited set of pixel layouts (CC, F1, MW, MWflip, DH, F2), see the ducc documentation. Maps with partial sky coverage compatible with these pixelizations will be temporarily padded to full sky before the transform. For other maps, this method will fail.

"cyl": Use ducc's standard transforms. These work for any cylindrical
projection where pixels are equi-spaced and evenly divide the sky along each horizontal line. Maps with partial sky coverage will be temporarily padded horizontally as necessary.

"general": Use ducc's general transforms. These work for any pixelization,
but are significantly more expensive, both in terms of time and memory.

"auto": Automatically choose "2d", "cyl" or "general". This is the default.,

- `ainfo` (*alm_info object containing information about the alm layout.*) – default: standard triangular layout,
- `verbose` (*If True, prints information about what's being done*) –
- `nthread` (*Number of threads to use. Defaults to OMP_NUM_THREADS.*) –
- `epsilon` (*The desired fractional accuracy. Used for interpolation*) – in the "general" method. Default: 1e-6.
- `pix_tol` (*Tolerance for matching a pixel layout with a predefined one,*) – in fractions of a pixel. Default: 1e-6.
- `locinfo` (*Information about the coordinates and validity of each pixel.*) – Only relevant for the "general" method. Computed via calc_locinfo if missing. If you're doing multiple transforms with the same geometry, you can speed things up by precomputing this and passing it in here.

Returns

- *The resulting map. This will be the same object as the map argument,*
- *or a copy if copy == True.*

`pixell.curvedsky.`**`alm2map_adjoint`**(*map*, *alm=None*, *spin=[0, 2]*, *deriv=False*, *copy=False*, *method='auto'*, *ainfo=None*, *verbose=False*, *nthread=None*, *epsilon=None*, *pix_tol=1e-06*, *locinfo=None*)

The adjoint of map2alm. Forwards to map2alm; see its docstring for details

`pixell.curvedsky.`**`alm2map_pos`**(*alm*, *pos=None*, *loc=None*, *ainfo=None*, *map=None*, *spin=[0, 2]*, *deriv=False*, *copy=False*, *verbose=False*, *adjoint=False*, *nthread=None*, *epsilon=None*)

Like alm2map, but operates directly on arbitrary positions instead of an enmap. The positions are given either with the pos argument or the loc argument.

pos: [{dec,ra},…] in radians loc: […,{codec,ra}] in radians. codec is pi/2 - dec, ra must be positive

The underlying implementation uses loc, so if pos is passed an internal loc will be built. See alm2map for the meaning of the other arguments.

`pixell.curvedsky.`**`map2alm`**(*map*, *alm=None*, *lmax=None*, *spin=[0, 2]*, *deriv=False*, *adjoint=False*, *copy=False*, *method='auto'*, *ainfo=None*, *verbose=False*, *nthread=None*, *niter=0*, *epsilon=None*, *pix_tol=1e-06*, *weights=None*, *locinfo=None*, *tweak=False*)

Spherical harmonics analysis. Transform from real space to harmonic space.

Parameters

- `map` (*float32 or float64 enmap with shape [...,ncomp,ny,nx], [ncomp, ny,nx]*) – or [ny,nx]. All but last two dimensions must match alm.

- **alm** (*complex64 or complex128 numpy array with shape [...,ncomp,nelem],*) – [ncomp,nelem] or [nelem]. Spin transforms will be applied to the ncomp axis, controlled by the spin argument below. Will be overwritten unless copy is True
- **Options** –
- **-------** –
- **spin** (*list of spins. These describe how to handle the [ncomp] axis.*) – 0: scalar transform. Consumes one element in the component axis not 0: spin transform. Consumes two elements from the component axis. For example, if you have a TEB alm [3,nelem] and want to transform it to a TQU map [3,ny,nx], you would use spin=[0,2] to perform a scalar transform for the T component and a spin-2 transform for the Q,U components. Another example. If you had an alm [5,nelem] and map [5,ny,nx] and the first element was scalar, the next pair spin-1 and the next pair spin-2, you woudl use spin=[0,1,2]. default:[0,2]
- **deriv** (*If true, instead calculates the d/ddec and d/dra derivatives*) – of the map corresponding to the alms. In this case the alm must have shape [...,nelem] or [nelem] and the map must have shape [...,2,ny,nx] or [2,ny,nx]. default: False
- **adjoint** (*If true, instead calculates the adjoint of the*) – map2alm operation. This reads from alm and writes to map. default: False
- **copy** (*If true, writes to a copy of map instead of overwriting the*) – map argument. The resulting map is returned.
- **method** (*Select the spherical harmonics transform method:*) –

  "2d": Use ducc's "2d" transforms. These are fast and accurate, but
  require full-sky CAR maps with one of a limited set of pixel layouts (CC, F1, MW, MWflip, DH, F2), see the ducc documentation. Maps with partial sky coverage compatible with these pixelizations will be temporarily padded to full sky before the transform. For other maps, this method will fail.

  "cyl": Use ducc's standard transforms. These work for any cylindrical
  projection where pixels are equi-spaced and evenly divide the sky along each horizontal line. Maps with partial sky coverage will be temporarily padded horizontally as necessary.

  "general": Use ducc's general transforms. These work for any pixelization,
  but are significantly more expensive, both in terms of time and memory.

  "auto": Automatically choose "2d", "cyl" or "general". This is the default.,
- **ainfo** (*alm_info object containing information about the alm layout.*) – default: standard triangular layout,
- **verbose** (*If True, prints information about what's being done*) –
- **nthread** (*Number of threads to use. Defaults to OMP_NUM_THREADS.*) –
- **niter** (*The number of Jacobi iteration steps to perform when*) – estimating the map2alm integral. Should ideally be controlled via epsilon, but is manual for now. Only relevant for the "cyl" and "general" methods. Time proportional to 1+2*niter. For a flat spectrum, niter=0 typically results in std(alm-alm_true)/std(alm_true) 1e-5, improving to 1e-8 by niter=3. Default: 0
- **epsilon** (*The desired fractional accuracy. Used for interpolation*) – in the "general" method. Default: 1e-6.
- **pix_tol** (*Tolerance for matching a pixel layout with a predefined one,*) – in fractions of a pixel. Default: 1e-6.

- **weights** (*Integration weights to use. Only used for methods "cyl" and "general".*) – Defaults to ducc's grid weights if available, otherwise the pixel area. Somewhat heavy to compute and store for the "general" method, so if you're performing multiple map2alm operations with the same geometry, consider precomputing them and passing them with this argument. Must have the same shape as locinfo.loc for the "general" method.
- **locinfo** (*Information about the coordinates and validity of each pixel.*) – Only relevant for the "general" method. Computed via calc_locinfo if missing. If you're doing multiple transforms with the same geometry, you can speed things up by precomputing this and passing it in here.

Returns

- *The resulting alm. This will be the same object as the alm argument,*
- *or a copy if copy == True.*

`pixell.curvedsky.`**`map2alm_adjoint`**(*alm*, *map*, *lmax=None*, *spin=[0, 2]*, *deriv=False*, *copy=False*, *method='auto'*, *ainfo=None*, *verbose=False*, *nthread=None*, *niter=0*, *epsilon=1e-06*, *pix_tol=1e-06*, *weights=None*, *locinfo=None*)

The adjoint of alm2map. Forwards to map2alm. See its docstring for details

`pixell.curvedsky.`**`alm2map_healpix`**(*alm*, *healmap=None*, *spin=[0, 2]*, *deriv=False*, *adjoint=False*, *copy=False*, *ainfo=None*, *nside=None*, *theta_min=None*, *theta_max=None*, *nthread=None*)

Projects the given alm[…,ncomp,nalm] onto the given healpix map healmap[…,ncomp,npix].

`pixell.curvedsky.`**`map2alm_healpix`**(*healmap*, *alm=None*, *ainfo=None*, *lmax=None*, *spin=[0, 2]*, *weights=None*, *deriv=False*, *copy=False*, *verbose=False*, *adjoint=False*, *niter=0*, *theta_min=None*, *theta_max=None*, *nthread=None*)

map2alm for healpix maps. Similar to healpy's map2alm. See the map2alm docstring for details.

`class pixell.curvedsky.`**`alm_info`**(*lmax=None*, *mmax=None*, *nalm=None*, *stride=1*, *layout='triangular'*)

`lm2ind`(*l*, *m*)

`get_map`()

Return the explicit [nelem,{l,m}] mapping this alm_info represents.

`transpose_alm`(*alm*, *out=None*)

In order to accomodate l-major ordering, which is not directly supported, this function efficiently transposes Alm into Aml. If the out argument is specified, the transposed result will be written there. In order to perform an in-place transpose, call this function with the same array as "alm" and "out". If the out argument is not specified, then a new array will be constructed and returned.

`alm2cl`(*alm*, *alm2=None*)

Computes the cross power spectrum for the given alm and alm2, which

must have the same dtype and broadcast. For example, to get the TEB,TEB cross spectra for a single map you would do

cl = ainfo.alm2cl(alm[:,None,:], alm[None,:,:])

To get the same TEB,TEB spectra crossed with a different map it would be

cl = ainfo.alm2cl(alm1[:,None,:], alm2[None,:,:])

In both these cases the output will be [{T,E,B},{T,E,B},nl].

The returned cls start at ell=0.

lmul(*alm*, *lmat*, *out=None*)

Computes res[a,lm] = lmat[a,b,l]*alm[b,lm], where lm is the position of the element with (l,m) in the alm array, as defined by this class.

pixell.curvedsky.**get_method**(*shape*, *wcs*, *minfo=None*, *pix_tol=1e-06*)

Return which method map2alm and alm2map will use for the given enmap geometry. Returns either “2d”, “cyl” or “general”.

pixell.curvedsky.**quad_weights**(*shape*, *wcs*, *pix_tol=1e-06*)

Return the quadrature weights to use for map2alm operations for the given geometry. Only valid for a limited number of cylindrical geometries recognized by ducc. Returns weights[ny] where ny is shape[-2]. For cases where quadrature weights aren’t available, it’s a pretty good approximation to just use the pixel area.

pixell.curvedsky.**profile2harm**(*br*, *r*, *lmax=None*, *oversample=1*, *left=None*, *right=None*)

This is an alternative to healpy.beam2bl. In my tests it’s a bit more accurate and about 3x faster, most of which is spent constructing the quadrature. It does use some interpolation internally, though, so there might be cases where it’s less accurate. Transforms the function br(r) to bl(l). br has shape […,nr], and the output will have shape […,nl]. Implemented using sharp SHTs with one pixel per row and mmax=0. r is in radians and must be in ascending order.

pixell.curvedsky.**harm2profile**(*bl*, *r*)

The inverse of profile2beam or healpy.beam2bl. *Much* faster than these (150x faster in my test case). Should be exact too.

pixell.curvedsky.**prof2alm**(*profile*, *dir=[0, 1.5707963267948966]*, *spin=0*, *geometry='CC'*, *nthread=None*, *norot=False*)

Calculate the alms for a 1d equispaced profile[…,n] oriented along the given [ra,dec] on the sky.

pixell.curvedsky.**npix2nside**(*npix*)

pixell.curvedsky.**prepare_healmap**(*healmap*, *nside=None*, *pre=()*, *dtype=<class 'numpy.float64'>*)

pixell.curvedsky.**apply_minfo_theta_lim**(*minfo*, *theta_min=None*, *theta_max=None*)

pixell.curvedsky.**fill_gauss**(*arr*, *bsize=65536*)

pixell.curvedsky.**prepare_ps**(*ps*, *ainfo=None*, *lmax=None*)

pixell.curvedsky.**rand_alm_white**(*ainfo*, *pre=None*, *alm=None*, *seed=None*, *dtype=<class 'numpy.complex128'>*, *m_major=True*)

pixell.curvedsky.**almxfl**(*alm*, *lfilter=None*, *ainfo=None*, *out=None*)

Filter alms isotropically. Unlike healpy (at time of writing), this function allows leading dimensions in the alm, and also allows the filter to be specified as a function instead of an array.

Parameters

- **alm** – (…,N) ndarray of spherical harmonic alms
- **lfilter** – either an array containing the 1d filter to apply starting with ell=0
- **delta_ell=1** (*and separated by*) –
- **the** (*or a function mapping multipole ell to*) –
- **expression.** (*filtering*) –
- **ainfo** – If ainfo is provided, it is an alm_info describing the layout
- **itself.** (*of the input alm. Otherwise it will be inferred from the alm*) –

Returns
The filtered alms a_{l,m} * lfilter(l)

Return type
falm

`pixell.curvedsky.`**`filter`**`(`*imap*, *lfilter*, *ainfo=None*, *lmax=None*`)`

Filter a map isotropically by a function. Returns alm2map(map2alm(alm * lfilt(ell),lmax))

Parameters

- **imap** – (. . . ,Ny,Nx) ndmap stack of enmaps.
- **lmax** – integer specifying maximum multipole beyond which the alms are zeroed
- **lfilter** – either an array containing the 1d filter to apply starting with ell=0
- **delta_ell=1** (*and separated by*) –
- **the** (*or a function mapping multipole ell to*) –
- **expression.** (*filtering*) –
- **ainfo** – If ainfo is provided, it is an alm_info describing the layout

of the input alm. Otherwise it will be inferred from the alm itself.

Returns
(. . . ,Ny,Nx) ndmap stack of filtered enmaps

Return type
omap

`pixell.curvedsky.`**`alm2cl`**`(`*alm*, *alm2=None*, *ainfo=None*`)`

Compute the power spectrum for alm, or if alm2 is given, the cross-spectrum between alm and alm2, which must broadcast.

Some example usage, where the notation a[{x,y,z},n,m] specifies that the array a has shape [3,n,m], and the 3 entries in the first axis should be interpreted as x, y and z respectively.

1. cl[nl] = alm2cl(alm[nalm]) This just computes the standard power spectrum of the given alm, resulting in a single 1d array.
2. cl[nl] = alm2cl(alm1[nalm], alm2[nalm]) This compues the 1d cross-spectrum between the 1d alms alm1 and alm2.
3. cl[{T,E,B},{T,E,B},nl] = alm2cl(alm[{T,E,B},None,nalm], alm[None,{T,E,B},nalm]) This computes the 3x3 polarization auto-spectrum for a 2d polarized alm.
4. cl[{T,E,B},{T,E,B},nl] = alm2cl(alm1[{T,E,B},None,nalm], alm2[None,{T,E,B},nalm]) As above, but gives the 3x3 polarization cross-spectrum between two 2d alms.

The output is in the shape one would expect from numpy broadcasting. For example, in the last example, the TE power spectrum would be found in cl[0,1], and the ET power spectrum (which is different for the cross-spectrum case) is in cl[1,0]. If a Healpix-style compressed spectrum is desired, use pixell.powspec.sym_compress.

`pixell.curvedsky.`**`rotate_alm`**`(`*alm*, *psi*, *theta*, *phi*, *lmax=None*, *method='auto'*, *nthread=None*, *inplace=False*`)`

Rotate the given alm[. . . ,:] via the zyz rotations given by psi, theta and phi. The underlying implementation is provided by ducc0 or healpy. This is controlled with the "method" argument, which can be "ducc0", "healpy" or "auto". For "auto" it uses ducc0 if available, otherwise healpy. The resulting alm is returned. If inplace=True, then the input alm will be modified in place (but still returned). The number of threads to use is controlled with the nthread argument. If this is 0 (the default), then the number of threads is given by the value of the OMP_NUM_THREADS variable.

`pixell.curvedsky.transfer_alm`(*iainfo*, *ialm*, *oainfo*, *oalm=None*, *op=<function <lambda>>*)

Copy data from ialm with layout given by iainfo to oalm with layout given by oainfo. If oalm is not passed, it will be allocated. In either case oalm is returned. If op is specified, then it defines out oalm is updated: oalm = op(ialm, oalm). For example, if op = lambda a,b:a+b, then ialm would be added to oalm instead of overwriting it.

`pixell.curvedsky.alm2map_2d`(*alm*, *map*, *ainfo=None*, *minfo=None*, *spin=[0, 2]*, *deriv=False*, *copy=False*, *verbose=False*, *adjoint=False*, *nthread=None*, *pix_tol=1e-06*)

Helper function for alm2map. See its docstring for details

`pixell.curvedsky.alm2map_cyl`(*alm*, *map*, *ainfo=None*, *minfo=None*, *spin=[0, 2]*, *deriv=False*, *copy=False*, *verbose=False*, *adjoint=False*, *nthread=None*, *pix_tol=1e-06*)

Helper function for alm2map. See its docstring for details

`pixell.curvedsky.alm2map_general`(*alm*, *map*, *ainfo=None*, *spin=[0, 2]*, *deriv=False*, *copy=False*, *verbose=False*, *adjoint=False*, *nthread=None*, *locinfo=None*, *epsilon=None*)

Helper function for alm2map. See its docstring for details

`pixell.curvedsky.map2alm_2d`(*map*, *alm=None*, *ainfo=None*, *minfo=None*, *lmax=None*, *spin=[0, 2]*, *deriv=False*, *copy=False*, *verbose=False*, *adjoint=False*, *nthread=None*, *pix_tol=1e-06*)

Helper function for map2alm. See its docsctring for details.

`pixell.curvedsky.map2alm_cyl`(*map*, *alm=None*, *ainfo=None*, *minfo=None*, *lmax=None*, *spin=[0, 2]*, *weights=None*, *deriv=False*, *copy=False*, *verbose=False*, *adjoint=False*, *nthread=None*, *pix_tol=1e-06*, *niter=0*)

Helper function for map2alm. See its docsctring for details.

`pixell.curvedsky.map2alm_general`(*map*, *alm=None*, *ainfo=None*, *minfo=None*, *lmax=None*, *spin=[0, 2]*, *weights=None*, *deriv=False*, *copy=False*, *verbose=False*, *adjoint=False*, *nthread=None*, *locinfo=None*, *epsilon=None*, *niter=0*)

Helper function for map2alm. See its docsctring for details.

`pixell.curvedsky.alm2map_raw_2d`(*alm*, *map*, *ainfo=None*, *spin=[0, 2]*, *deriv=False*, *copy=False*, *verbose=False*, *adjoint=False*, *nthread=None*)

Helper function for alm2map_2d. Usually not called directly. See the alm2map docstring for details.

`pixell.curvedsky.alm2map_raw_cyl`(*alm*, *map*, *ainfo=None*, *minfo=None*, *spin=[0, 2]*, *deriv=False*, *copy=False*, *verbose=False*, *adjoint=False*, *nthread=None*)

Helper function for alm2map_cyl. Usually not called directly. See the alm2map docstring for details.

`pixell.curvedsky.alm2map_raw_general`(*alm*, *map*, *loc*, *ainfo=None*, *spin=[0, 2]*, *deriv=False*, *copy=False*, *verbose=False*, *adjoint=False*, *nthread=None*, *epsilon=None*)

Helper function for alm2map_general. Usually not called directly. See the alm2map docstring for details.

`pixell.curvedsky.map2alm_raw_2d`(*map*, *alm=None*, *ainfo=None*, *lmax=None*, *spin=[0, 2]*, *deriv=False*, *copy=False*, *verbose=False*, *adjoint=False*, *nthread=None*)

Helper function for map2alm_2d. Usually not called directly. See the map2alm docstring for details.

`pixell.curvedsky.map2alm_raw_cyl`(*map*, *alm=None*, *ainfo=None*, *lmax=None*, *spin=[0, 2]*, *weights=None*, *deriv=False*, *copy=False*, *verbose=False*, *adjoint=False*, *niter=0*, *nthread=None*)

Helper function for map2alm_cyl. Usually not called directly. See the map2alm docstring for details.

`pixell.curvedsky.`**`map2alm_raw_general`**(*map*, *loc*, *alm=None*, *ainfo=None*, *lmax=None*, *spin=[0, 2]*, *weights=None*, *deriv=False*, *copy=False*, *verbose=False*, *adjoint=False*, *nthread=None*, *niter=0*, *epsilon=None*)

Helper function for map2alm_general. Usually not called directly. See the map2alm docstring for details.

`pixell.curvedsky.`**`jacobi_inverse`**(*forward*, *approx_backward*, *y*, *niter=0*)

Given y = forward(x), attempt to recover x using jacobi iteration with forward and it's approximate inverse approx_backward. niter controls the number of iterations. The number of calls to forward is niter. The number of calls to approx_backward is 1+niter.

See minres_inverse for a function with faster convergence and better stopping criterion. But Jacobi's quick startup time often means it's finished by the time minres has gotten started, so unless high accuracy is needed, Jacobi might be the best choice.

`pixell.curvedsky.`**`minres_inverse`**(*forward*, *approx_backward*, *y*, *epsilon=1e-06*, *maxiter=100*, *zip=None*, *unzip=None*, *verbose=False*)

Given y = forward(x), attempt to recover the maximum-likelihood solution of x = (P'N"P)"P'N"P using Minres iteration. Here forward = P and approx_backward = P'N". Both of these should be functions that takes a single argument and returns the result. Iterates until the desired accuracy given by epsilon is reached, or the maximum number of iterations given by maxiter is reached. If verbose is True, prints information about each step in the iteration.

This function converges more quickly than jacobi, and has a better defined stopping criterion, but uses more memory and has a higher startup cost. Effectively this function starts two iteration steps behind jacobi, and takes several more steps to catch up. It is therefore not the fastest choice when only moderate accuracy is needed.

`pixell.curvedsky.`**`nalm2lmax`**(*nalm*)

`pixell.curvedsky.`**`get_ring_info`**(*shape*, *wcs*, *dtype=<class 'numpy.float64'>*)

Return information about the horizontal rings of pixels in a cylindrical pixelization. Used in map2alm and alm2map with the "cyl" method.

`pixell.curvedsky.`**`get_ring_info_healpix`**(*nside*, *rings=None*)

`pixell.curvedsky.`**`get_ring_info_radial`**(*r*)

Construct a ring info for a case where there's just one pixel in each ring. This is useful for radially symmetric (mmax=0) transforms.

`pixell.curvedsky.`**`flip2slice`**(*flips*)

`pixell.curvedsky.`**`flip_geometry`**(*shape*, *wcs*, *flips*)

`pixell.curvedsky.`**`flip_array`**(*arr*, *flips*)

`pixell.curvedsky.`**`pad_geometry`**(*shape*, *wcs*, *pad*)

`pixell.curvedsky.`**`analyse_geometry`**(*shape*, *wcs*, *tol=1e-06*)

Pass in shape, wcs, and get out an info object that contains

case:
2d: can be passed directly to synthesis_2d cyl: can be passed directly to synthesis partial: can be passed to synthesis after ring-extension,

or synthesis_2d after full extension

general: only synthesis_general can be used

flip: [flipy,flipx] bools. Only relevant for 2d and cyl.
partial always needs slices, general never needs them.

ducc_geo: Matching ducc geometry. Most useful member is .name, which
can be “CC”, “F1”, “MW”, “MWflip” “DH”, “F2”. ducc_geo is None if this doesn’t correspond to a ducc geometry.

ypad: [npre,npost]. Only used when padding to 2d xpad: [npre,npost]. Used when case==”partial”

`pixell.curvedsky.`**`get_ducc_geo`**(*wcs*, *shape=None*, *tol=1e-06*)

Return the ducc geometry type for the given world coordinate system object. Returns a bunch(name, phi0) where name is one of “CC”, “F1”, “MW”, “MWflip”, “DH” and “F2”. “GL”: gauss-legendre is not supported by wcs. Returns None if the wcs doesn’t correspond to a ducc geometry.

`pixell.curvedsky.`**`get_ducc_maxlmax`**(*name*, *ny*)

`pixell.curvedsky.`**`calc_locinfo`**(*shape*, *wcs*, *bsize=1000*)

Calculate pixel position info in the format ducc needs

`pixell.curvedsky.`**`map2buffer`**(*map*, *flip*, *pad*, *obuf=False*)

Prepare a map for ducc operations by flipping and/or padding it, returning the resulting map.

`pixell.curvedsky.`**`buffer2map`**(*map*, *flip*, *pad*)

The inverse of map2buffer. Undoes flipping and padding

`pixell.curvedsky.`**`prepare_alm`**(*alm=None*, *ainfo=None*, *lmax=None*, *pre=()*, *dtype=<class 'numpy.float64'>*)

Set up alm and ainfo based on which ones of them are available.

`pixell.curvedsky.`**`prepare_raw`**(*alm*, *map*, *ainfo=None*, *lmax=None*, *deriv=False*, *verbose=False*, *nthread=None*, *pixdims=2*)

`pixell.curvedsky.`**`dangerous_dtype`**(*dtype*)

3.4 2.4 utils - General utilities

`pixell.utils.`**`l2ang`**(*l*)

Compute the angular scale roughly corresponding to a given multipole. Based on matching the number of alm degrees of freedom with map degrees of freedom.

`pixell.utils.`**`ang2l`**(*ang*)

Compute the multipole roughly corresponding to a given angular scale. Based on matching the number of alm degrees of freedom with map degrees of freedom.

`pixell.utils.`**`D`**(*f*, *eps=1e-10*)

Clever derivative operator for function f(x) from Ivan Yashchuck. Accurate to second order in eps. Only calls f(x) once to evaluate the derivative, but f must accept complex arguments. Only works for real x. Example usage: D(lambda x: x**4)(1) => 4.0

`pixell.utils.`**`lines`**(*file_or_fname*)

Iterates over lines in a file, which can be specified either as a filename or as a file object.

`pixell.utils.`**`touch`**(*fname*)

`pixell.utils.`**`listsplit`**(*seq*, *elem*)

Analogue of str.split for lists. listsplit([1,2,3,4,5,6,7],4) -> [[1,2],[3,4,5,6]].

`pixell.utils.`**`streq`**(*x*, *s*)

Check if x is the string s. This used to be simply “x is s”, but that now causes a warning. One can’t just do “x == s”, as that causes a numpy warning and will fail in the future.

`pixell.utils.`**`find`**(*array*, *vals*, *default=None*)

Return the indices of each value of vals in the given array.

`pixell.utils.`**`find_any`**(*array*, *vals*)

Like find, but skips missing entries

`pixell.utils.`**`nearest_ind`**(*arr*, *vals*, *sorted=False*)

Given array arr and values vals, return the index of the entry in arr with value closest to each entry in val

`pixell.utils.`**`contains`**(*array*, *vals*)

Given an array[n], returns a boolean res[n], which is True for any element in array that is also in vals, and False otherwise.

`pixell.utils.`**`common_vals`**(*arrs*)

Given a list of arrays, returns their intersection. For example

> common_vals([[1,2,3,4,5],[2,4,6,8]]) -> [2,4]

`pixell.utils.`**`common_inds`**(*arrs*)

Given a list of arrays, returns the indices into each of them of their common elements. For example

> common_inds([[1,2,3,4,5],[2,4,6,8]]) -> [[1,3],[0,1]]

`pixell.utils.`**`union`**(*arrs*)

Given a list of arrays, returns their union.

`pixell.utils.`**`inverse_order`**(*order*)

If order represents a reordering of an array, such as that returned by np.argsort, inverse_order(order) returns a new reordering that can be used to recover the old one.

Example

a = np.array([6,102,32,20,0,91,1910]]) order = np.argsort(a) print(a[order]) => [0,6,20,32,91,102,1910] invorder = inverse_order(order) print(a[order][inverse_order]) => [6,102,32,20,0,91,1910] # same as a

`pixell.utils.`**`complement_inds`**(*inds*, *n*)

Given a subset of range(0,n), return the missing values. E.g. complement_inds([0,2,4],7) => [1,3,5,6]

`pixell.utils.`**`dict_apply_listfun`**(*dict*, *function*)

Applies a function that transforms one list to another with the same number of elements to the values in a dictionary, returning a new dictionary with the same keys as the input dictionary, but the values given by the results of the function acting on the input dictionary's values. I.e. if f(x) = x[::-1], then dict_apply_listfun({"a":1,"b":2},f) = {"a":2,"b":1}.

`pixell.utils.`**`fallback`**(**args*)

`pixell.utils.`**`unwind`**(*a*, *period=6.283185307179586*, *axes=[-1]*, *ref=0*, *refmode='left'*, *mask_nan=False*)

Given a list of angles or other cyclic coordinates where a and a+period have the same physical meaning, make a continuous by removing any sudden jumps due to period-wrapping. I.e. [0.07,0.02,6.25,6.20] would become [0.07,0.02,-0.03,-0.08] with the default period of 2*pi.

`pixell.utils.`**`rewind`**(*a*, *ref=0*, *period=6.283185307179586*)

Given a list of angles or other cyclic corodinates, add or subtract multiples of the period in order to ensure that they all lie within the same period. The ref argument specifies the angle furthest away from the cut, i.e. the period cut will be at ref+period/2.

`pixell.utils.`**`cumsplit`**`(`*sizes*, *capacities*`)`

Given a set of sizes (of files for example) and a set of capacities (of disks for example), returns the index of the sizes for which each new capacity becomes necessary, assuming sizes can be split across boundaries. For example cumsplit([1,1,2,0,1,3,1],[3,2,5]) -> [2,5]

`pixell.utils.`**`mask2range`**`(`*mask*`)`

Convert a binary mask [True,True,False,True,. . .] into a set of ranges [:,{start,stop}].

`pixell.utils.`**`repeat_filler`**`(`*d*, *n*`)`

Form an array n elements long by repeatedly concatenating d and d[::-1].

`pixell.utils.`**`deslope`**`(`*d*, *w=1*, *inplace=False*, *axis=-1*, *avg=<function mean>*`)`

Remove a slope and mean from d, matching up the beginning and end of d. The w parameter controls the number of samples from each end of d that is used to determine the value to match up.

`pixell.utils.`**`ctime2mjd`**`(`*ctime*`)`

Converts from unix time to modified julian date.

`pixell.utils.`**`mjd2djd`**`(`*mjd*`)`

`pixell.utils.`**`djd2mjd`**`(`*djd*`)`

`pixell.utils.`**`mjd2jd`**`(`*mjd*`)`

`pixell.utils.`**`jd2mjd`**`(`*jd*`)`

`pixell.utils.`**`ctime2djd`**`(`*ctime*`)`

`pixell.utils.`**`djd2ctime`**`(`*djd*`)`

`pixell.utils.`**`ctime2jd`**`(`*ctime*`)`

`pixell.utils.`**`jd2ctime`**`(`*jd*`)`

`pixell.utils.`**`mjd2ctime`**`(`*mjd*`)`

Converts from modified julian date to unix time

`pixell.utils.`**`medmean`**`(`*x*, *axis=None*, *frac=0.5*`)`

`pixell.utils.`**`maskmed`**`(`*arr*, *mask=None*, *axis=-1*, *maskval=0*`)`

Median of array along the given axis, but ignoring entries with the given mask value.

`pixell.utils.`**`moveaxis`**`(`*a*, *o*, *n*`)`

`pixell.utils.`**`moveaxes`**`(`*a*, *old*, *new*`)`

`pixell.utils.`**`search`**`(`*a*, *v*, *side='left'*`)`

Like np.searchsorted, but searches a[. . . ,n] along the last axis for v[. . .] values, returning the inds[. . .] values. Does not perform a binary search, so less efficient on large arrays, but faster than my original idea of shoehorning the arrays into a single monotonic array, since that would have required touching all the values anyway.

`pixell.utils.`**`weighted_quantile`**`(`*map*, *ivar*, *quantile*, *axis=-1*`)`

Multidimensional weighted quantile. Takes the given quantile (scalar) along the given axis (default last axis) of the array "map". Each element along the axis is given weight from the corresponding element in "ivar". This is based on the weighted percentile method in https://en.wikipedia.org/wiki/Percentile.

Arguments: map: The array to find quantiles for. Must broadcast with ivar ivar: The weight array. Must broadcast with map quantiles: The quantiles to evaluate. axis: The axis to take the quantiles along.

If post-broadcast map and ivar have shape A when excluding the quantile-axis and quantile has shape B, then the result will have shape B+A.

`pixell.utils.`**`weighted_median`**(*map*, *ivar=1*, *axis=-1*)

Compute the multidimensional weghted median. See weighted_quantile for details

`pixell.utils.`**`partial_flatten`**(*a*, *axes=[-1]*, *pos=0*)

Flatten all dimensions of a except those mentioned in axes, and put the flattened one at the given position.

Example: if a.shape is [1,2,3,4], then partial_flatten(a,[-1],0).shape is [6,4].

`pixell.utils.`**`partial_expand`**(*a*, *shape*, *axes=[-1]*, *pos=0*)

Undo a partial flatten. Shape is the shape of the original array before flattening, and axes and pos should be the same as those passed to the flatten operation.

`pixell.utils.`**`addaxes`**(*a*, *axes*)

`pixell.utils.`**`delaxes`**(*a*, *axes*)

`class pixell.utils.`**`flatview`**(*array*, *axes=[]*, *mode='rwc'*, *pos=0*)

Produce a read/writable flattened view of the given array, via with flatview(arr) as farr:

> do stuff with farr

Changes to farr are propagated into the original array. Flattens all dimensions of a except those mentioned in axes, and put the flattened one at the given position.

`class pixell.utils.`**`nowarn`**

Use in with block to suppress warnings inside that block.

`pixell.utils.`**`dedup`**(*a*)

Removes consecutive equal values from a 1d array, returning the result. The original is not modified.

`pixell.utils.`**`interpol`**(*a*, *inds*, *order=3*, *mode='nearest'*, *mask_nan=False*, *cval=0.0*, *prefilter=True*)

Given an array a[{x},{y}] and a list of float indices into a, inds[len(y),{z}], returns interpolated values at these positions as [{x},{z}].

`pixell.utils.`**`interpol_prefilter`**(*a*, *npre=None*, *order=3*, *inplace=False*, *mode='nearest'*)

`pixell.utils.`**`interp`**(*x*, *xp*, *fp*, *left=None*, *right=None*, *period=None*)

Unlike utils.interpol, this is a simple wrapper around np.interp that extends it to support fp[…,n] instead of just fp[n]. It does this by looping over the other dimensions in python, and calling np.interp for each entry in the pre-dimensions. So this function does not save any time over doing that looping manually, but it avoid typing this annoying loop over and over.

`pixell.utils.`**`bin_multi`**(*pix*, *shape*, *weights=None*)

Simple multidimensional binning. Not very fast. Given pix[{coords},:] where coords are indices into an array with shape shape, count the number of hits in each pixel, returning map[shape].

`pixell.utils.`**`bincount`**(*pix*, *weights=None*, *minlength=0*)

Like numpy.bincount, but allows pre-dimensions, which must broadcast

`pixell.utils.`**`grid`**(*box*, *shape*, *endpoint=True*, *axis=0*, *flat=False*)

Given a bounding box[{from,to},ndim] and shape[ndim] in each direction, returns an array [ndim,shape[0],shape[1],…] array of evenly spaced numbers. If endpoint is True (default), then the end point is included. Otherwise, the last sample is one step away from the end of the box. For one dimension, this is similar to linspace:

> linspace(0,1,4) => [0.0000, 0.3333, 0.6667, 1.0000] grid([[0],[1]],[4]) => [[0,0000, 0.3333, 0.6667, 1.0000]]

`pixell.utils.`**`cumsum`**`(`*a*, *endpoint=False*, *axis=None*`)`

As numpy.cumsum for a 1d array a, but starts from 0. If endpoint is True, the result will have one more element than the input, and the last element will be the sum of the array. Otherwise (the default), it will have the same length as the array, and the last element will be the sum of the first n-1 elements.

`pixell.utils.`**`pixwin_1d`**`(`*f*, *order=0*`)`

Calculate the 1D pixel window for the dimensionless frequncy f corresponding to a pixel spacing of 1 (so the Nyquist frequncy is 0.5). The order argument controls the interpolation order to assume in the mapmaker. order = 0 corresponds to standard nearest-neighbor mapmking. order = 1 corresponds to linear interpolation. For a multidimensional (e.g. 2d) image, the full pixel window will be the outer product of this pixel window along each axis.

`pixell.utils.`**`nearest_product`**`(`*n*, *factors*, *direction='below'*`)`

Compute the highest product of positive integer powers of the specified factors that is lower than or equal to n. This is done using a simple, O(n) brute-force algorithm.

`pixell.utils.`**`mkdir`**`(`*path*`)`

`pixell.utils.`**`symlink`**`(`*src*, *dest*`)`

`pixell.utils.`**`decomp_basis`**`(`*basis*, *vec*`)`

`pixell.utils.`**`find_period`**`(`*d*, *axis=-1*`)`

`pixell.utils.`**`find_period_fourier`**`(`*d*, *axis=-1*`)`

This is a simple second-order estimate of the period of the assumed-periodic signal d. It finds the frequency with the highest power using an fft, and partially compensates for nonperiodicity by taking a weighted mean of the position of the top.

`pixell.utils.`**`find_period_exact`**`(`*d*, *guess*`)`

`pixell.utils.`**`find_sweeps`**`(`*az*, *tol=0.2*`)`

Given an array "az" that sweeps up and down between approximately constant minimum and maximum values, returns an array sweeps[:,{i1,i2}], which gives the start and end index of each such sweep. For example, if az starts at 0 at sample 0, increases to 1 at sample 1000 and then falls to -1 at sample 2000, increase to 1 at sample 2500 and then falls to 0.5 at sample 3000 where it ends, then the function will return [[0,1000],[1000,2000],[2000,2500],[2500,3000]]. The tol parameter determines how close to the extremum values of the array it will look for turnarounds. It shouldn't normally need to be ajusted.

`pixell.utils.`**`equal_split`**`(`*weights*, *nbin*`)`

Split weights into nbin bins such that the total weight in each bin is as close to equal as possible. Returns a list of indices for each bin.

`pixell.utils.`**`range_sub`**`(`*a*, *b*, *mapping=False*`)`

Given a set of ranges a[:,{from,to}] and b[:,{from,to}], return a new set of ranges c[:,{from,to}] which corresponds to the ranges in a with those in b removed. This might split individual ranges into multiple ones. If mapping=True, two extra objects are returned. The first is a mapping from each output range to the position in a it comes from. The second is a corresponding mapping from the set of cut a and b range to indices into a and b, with b indices being encoded as -i-1. a and b are assumed to be internally non-overlapping.

Example: utils.range_sub([[0,100],[200,1000]], [[1,2],[3,4],[8,999]], mapping=True) (array([[0, 1],

[2, 3], [4, 8], [999, 1000]]),

array([0, 0, 0, 1]), array([0, -1, 1, -2, 2, -3, 3]))

The last array can be interpreted as: Moving along the number line, we first encounter [0,1], which is a part of range 0 in c. We then encounter range 0 in b ([1,2]), before we hit [2,3] which is part of range 1 in c. Then comes range 1 in b ([3,4]) followed by [4,8] which is part of range 2 in c, followed by range 2 in b ([8,999]) and finally [999,1000] which is part of range 3 in c.

The same call without mapping: utils.range_sub([[0,100],[200,1000]], [[1,2],[3,4],[8,999]]) array([[0, 1],

[2, 3], [4, 8], [999, 1000]])

`pixell.utils.range_union`(*a*, *mapping=False*)

Given a set of ranges a[:,{from,to}], return a new set where all overlapping ranges have been merged, where to >= from. If mapping=True, then the mapping from old to new ranges is also returned.

`pixell.utils.range_normalize`(*a*)

Given a set of ranges a[:,{from,to}], normalize the ranges such that no ranges are empty, and all ranges go in increasing order. Decreasing ranges are interpreted the same way as in a slice, e.g. empty.

`pixell.utils.range_cut`(*a*, *c*)

Cut range list a at positions given by c. For example range_cut([[0,10],[20,100]],[0,2,7,30,200]) -> [[0,2],[2,7],[7,10],[20,30],[30,100]].

`pixell.utils.compress_beam`(*sigma*, *phi*)

`pixell.utils.expand_beam`(*irads*, *return_V=False*)

`pixell.utils.combine_beams`(*irads_array*)

`pixell.utils.regularize_beam`(*beam*, *cutoff=0.01*, *nl=None*, *normalize=False*)

Given a beam transfer function beam[…,nl], replace small values with an extrapolation that has the property that the ratio of any pair of such regularized beams is constant in the extrapolated region.

`pixell.utils.read_lines`(*fname*, *col=0*)

Read lines from file fname, returning them as a list of strings. If fname ends with :slice, then the specified slice will be applied to the list before returning.

`pixell.utils.loadtxt`(*fname*)

As numpy.loadtxt, but allows slice syntax.

`pixell.utils.atleast_3d`(*a*)

`pixell.utils.atleast_Nd`(*a*, *n*)

Prepend length-1 dimensions to array a to make it n-dimensional

`pixell.utils.to_Nd`(*a*, *n*, *axis=0*, *return_inverse=False*)

`pixell.utils.preflat`(*a*, *n*)

Flatten the first n dimensions of a. If n is negative, flatten all but the last -n dimensions.

`pixell.utils.postflat`(*a*, *n*)

Flatten the last n dimensions of a. If n is negative, flatten all but the last -n dimensions.

`pixell.utils.between_angles`(*a*, *range*, *period=6.283185307179586*)

`pixell.utils.hasoff`(*val*, *off*, *tol=1e-06*)

Return True if val's deviation from an integer value equals off to the given tolerance (default: 1e-6). Example. hasoff(17.3, 0.3) == True

`pixell.utils.same_array`(*a*, *b*)

Returns true if a and b are the same array `pixell.utils.`**`fix_zero_strides`**`(`*a*`)`

Given an array a, return the same array with any zero-stride along an axis with length one, such as those introduced by None-indexing, replaced with an equivalent value

`pixell.utils.`**`greedy_split`**`(`*data*, *n=2*, *costfun=<built-in function max>*, *workfun=<function <lambda>>*`)`

Given a list of elements data, return indices that would split them it into n subsets such that cost is approximately minimized. costfun specifies which cost to minimize, with the default being the value of the data themselves. workfun specifies how to combine multiple values. workfun(datum,workval) => workval. scorefun then operates on a list of the total workval for each group score = scorefun([workval,workval,....]).

Example: greedy_split(range(10)) => [[9,6,5,2,1,0],[8,7,4,3]]
greedy_split([1,10,100]) => [[2],[1,0]] greedy_split("012345",costfun=lambda x:sum([xi**2 for xi in x]),

workfun=lambda w,x:0 if x is None else int(x)+w) => [[5,2,1,0],[4,3]]

`pixell.utils.`**`greedy_split_simple`**`(`*data*, *n=2*`)`

Split array "data" into n lists such that each list has approximately the same sum, using a greedy algorithm.

`pixell.utils.`**`cov2corr`**`(`*C*`)`

Scale rows and columns of C such that its diagonal becomes one. This produces a correlation matrix from a covariance matrix. Returns the scaled matrix and the square root of the original diagonal.

`pixell.utils.`**`corr2cov`**`(`*corr*, *std*`)`

Given a matrix "corr" and an array "std", return a version of corr with each row and column scaled by the corresponding entry in std. This is the reverse of cov2corr.

`pixell.utils.`**`eigsort`**`(`*A*, *nmax=None*, *merged=False*`)`

Return the eigenvalue decomposition of the real, symmetric matrix A. The eigenvalues will be sorted from largest to smallest. If nmax is specified, only the nmax largest eigenvalues (and corresponding vectors) will be returned. If merged is specified, E and V will not be returned separately. Instead, Q=VE**0.5 will be returned, such that QQ' = VEV'.

`pixell.utils.`**`nodiag`**`(`*A*`)`

Returns matrix A with its diagonal set to zero.

`pixell.utils.`**`date2ctime`**`(`*dstr*`)`

`pixell.utils.`**`bounding_box`**`(`*boxes*`)`

Compute bounding box for a set of boxes [:,2,:], or a set of points [:,2]

`pixell.utils.`**`unpackbits`**`(`*a*`)`

`pixell.utils.`**`box2corners`**`(`*box*`)`

Given a [{from,to},:] bounding box, returns [ncorner,:] coordinates of of all its corners.

`pixell.utils.`**`box2contour`**`(`*box*, *nperedge=5*`)`

Given a [{from,to},:] bounding box, returns [npoint,:] coordinates definiting its edges. Nperedge is the number of samples per edge of the box to use. For nperedge=2 this is equal to box2corners. Nperegege can be a list, in which case the number indicates the number to use in each dimension.

`pixell.utils.`**`box_slice`**`(`*a*, *b*`)`

Given two boxes/boxarrays of shape [{from,to},dims] or [:,{from,to},dims], compute the bounds of the part of each b that overlaps with each a, relative to the corner of a. For example box_slice([[2,5],[10,10]],[[0,0],[5,7]]) -> [[0,0],[3,2]].

`pixell.utils.`**`box_area`**`(`*a*`)`

Compute the area of a [{from,to},ndim] box, or an array of such boxes.

`pixell.utils.`**`box_overlap`**`(`*a*, *b*`)`

Given two boxes/boxarrays, compute the overlap of each box with each other box, returning the area of the overlaps. If a is [2,ndim] and b is [2,ndim], the result will be a single number. if a is [n,2,ndim] and b is [2,ndim], the result will be a shape [n] array. If a is [n,2,ndim] and b is [m,2,ndim], the result will' be [n,m] areas.

`pixell.utils.`**`widen_box`**`(`*box*, *margin=0.001*, *relative=True*`)`

`pixell.utils.`**`pad_box`**`(`*box*, *padding*`)`

How I should have implemented widen_box from the beginning. Simply pads a box by an absolute amount. The only complication is the sign stuff that handles descending axes in the box.

`pixell.utils.`**`unwrap_range`**`(`*range*, *nwrap=6.283185307179586*`)`

Given a logically ordered range[{from,to},…] that may have been exposed to wrapping with period nwrap, undo the wrapping so that range[1] > range[0] but range[1]-range[0] is as small as possible. Also makes the range straddle 0 if possible.

Unlike unwind and rewind, this function will not turn a very wide range into a small one because it doesn't assume that ranges are shorter than half the sky. But it still shortens ranges that are longer than a whole wrapping period.

`pixell.utils.`**`sum_by_id`**`(`*a*, *ids*, *axis=0*`)`

`pixell.utils.`**`pole_wrap`**`(`*pos*`)`

Given pos[{lat,lon},…], normalize coordinates so that lat is always between -pi/2 and pi/2. Coordinates outside this range are mirrored around the poles, and for each mirroring a phase of pi is added to lon.

`pixell.utils.`**`allreduce`**`(`*a*, *comm*, *op=None*`)`

Convenience wrapper for Allreduce that returns the result rather than needing an output argument.

`pixell.utils.`**`reduce`**`(`*a*, *comm*, *root=0*, *op=None*`)`

`pixell.utils.`**`allgather`**`(`*a*, *comm*`)`

Convenience wrapper for Allgather that returns the result rather than needing an output argument.

`pixell.utils.`**`allgatherv`**`(`*a*, *comm*, *axis=0*`)`

Perform an mpi allgatherv along the specified axis of the array a, returning an array with the individual process arrays concatenated along that dimension. For example allgatherv([[1,2]],comm) on one task and allgatherv([[3,4],[5,6]],comm) on another task results in [[1,2],[3,4],[5,6]] for both tasks.

`pixell.utils.`**`send`**`(`*a*, *comm*, *dest=0*, *tag=0*`)`

Faster version of comm.send for numpy arrays. Avoids slow pickling. Used with recv below.

`pixell.utils.`**`recv`**`(`*comm*, *source=0*, *tag=0*`)`

Faster version of comm.recv for numpy arrays. Avoids slow pickling. Used with send above.

`pixell.utils.`**`tuplify`**`(`*a*`)`

`pixell.utils.`**`resize_array`**`(`*arr*, *size*, *axis=None*, *val=0*`)`

Return a new array equal to arr but with the given axis reshaped to the given sizes. Inserted elements will be set to val.

`pixell.utils.`**`redistribute`**`(`*iarrs*, *iboxes*, *oboxes*, *comm*, *wrap=0*`)`

Given the array iarrs[[{pre},{dims}]] which represents slices garr[…,narr,ibox[0,0]:ibox[0,1]:ibox[0,2],ibox[1,0]:ibox[1,1]:ibox[1, of some larger, distributed array garr, returns a different slice of the global array given by obox.

`pixell.utils.`**`sbox_intersect`**`(`*a*, *b*, *wrap=0*`)`

Given two Nd sboxes a,b […,ndim,{start,end,step}] into the same array, compute an sbox representing their intersection. The resulting sbox will have positive step size. The result is a possibly empty list of sboxes - it is empty if there is no overlap. If wrap is specified, then it should be a list of length ndim of pixel wraps, each of which can be zero to disable wrapping in that direction.

`pixell.utils.`**`sbox_intersect_1d`**`(`*a*, *b*, *wrap=0*`)`

Given two 1d sboxes into the same array, compute an sbox representing their intersecting area. The resulting sbox will have positive step size. The result is a list of intersection sboxes. This can be empty if there is no intersection, such as between [0,n,2] and [1,n,2]. If wrap is not 0, then it should be an integer at which pixels repeat, so i and i+wrap would be equivalent. This can lead to more intersections than one would usually get.

`pixell.utils.`**`sbox_div`**`(`*a*, *b*, *wrap=0*`)`

Find c such that arr[a] = arr[b][c].

`pixell.utils.`**`sbox_mul`**`(`*a*, *b*`)`

Find c such that arr[c] = arr[a][b]

`pixell.utils.`**`sbox_flip`**`(`*sbox*`)`

`pixell.utils.`**`sbox2slice`**`(`*sbox*`)`

`pixell.utils.`**`sbox_size`**`(`*sbox*`)`

Return the size […,n] of an sbox […,{start,end,step}]. The end must be a whole multiple of step away from start, like as with the other sbox functions.

`pixell.utils.`**`sbox_fix0`**`(`*sbox*`)`

`pixell.utils.`**`sbox_fix`**`(`*sbox*`)`

`pixell.utils.`**`sbox_wrap`**`(`*sbox*, *wrap=0*, *cap=0*`)`

"Given a single sbox […,{from,to,step?}] representing a slice of an N-dim array, wraps and caps the sbox, returning a list of sboxes for each contiguous section of the slice.

The wrap argument, which can be scalar or a length N array-like, indicates the wrapping length along each dimension. Boxes that extend beyond the wrapping length will be split into two at the wrapping position, with the overshooting part wrapping around to the beginning of the array. The speical value 0 disables wrapping for that dimension.

The cap argument, which can also be a scalar or length N array-like, indicates the physical length of each array dimension. The sboxes will be truncated to avoid accessing any data beyond this length, after wrapping has been taken into account.

The function returns a list of the form [(ibox1,obox1),(ibox2,obox2)…], where the iboxes are sboxes representing slices into the input array (the array the original sbox refers to), while the oboxes represent slices into the output array. These sboxes can be turned into actual slices using sbox2slice.

A typical example of the use of this function would be a sky map that wraps horizontally after 360 degrees, where one wants to support extracting subsets that straddle the wrapping point.

`pixell.utils.`**`gcd`**`(`*a*, *b*`)`

Greatest common divisor of a and b

`pixell.utils.`**`lcm`**`(`*a*, *b*`)`

Least common multiple of a and b

`pixell.utils.`**`uncat`**`(`*a*, *lens*`)`

Undo a concatenation operation. If a = np.concatenate(b) and lens = [len(x) for x in b], then uncat(a,lens) returns b.

`pixell.utils.`**`ang2rect`**(*angs*, *zenith=False*, *axis=0*)

Convert a set of angles [{phi,theta},. . .] to cartesian coordinates [{x,y,z},. . .]. If zenith is True, the theta angle will be taken to go from 0 to pi, and measure the angle from the z axis. If zenith is False, then theta goes from -pi/2 to pi/2, and measures the angle up from the xy plane.

`pixell.utils.`**`rect2ang`**(*rect*, *zenith=False*, *axis=0*, *return_r=False*)

The inverse of ang2rect.

`pixell.utils.`**`angdist`**(*a*, *b*, *zenith=False*, *axis=0*)

Compute the angular distance between a[{ra,dec},. . .] and b[{ra,dec},. . .] using a Vincenty formula that's stable both for small and large angular separations. a and b must broadcast correctly.

`pixell.utils.`**`vec_angdist`**(*v1*, *v2*, *axis=0*)

Use Kahan's version of Heron's formula to compute a stable angular distance between to vectors v1 and v2, which don't have to be unit vectors. See https://scicomp.stackexchange.com/a/27694

`pixell.utils.`**`rotmatrix`**(*ang*, *raxis*, *axis=-1*, *dtype=None*)

Construct a 3d rotation matrix representing a rotation of ang degrees around the specified rotation axis raxis, which can be "x", "y", "z" or 0, 1, 2. If ang is a scalar, the result will be [3,3]. Otherwise, it will be ang.shape[:axis] + (3,3) + ang.shape[axis:]. Negative axis is interpreted as ang.ndim+1+axis, such that the (3,3) part ends at the end for axis=-1

`pixell.utils.`**`label_unique`**(*a*, *axes=()*, *rtol=1e-05*, *atol=1e-08*)

Given an array of values, return an array of labels such that all entries in the array with the same label will have approximately the same value. Labels count contiguously from 0 and up. axes specifies which axes make up the subarray that should be compared for equality. For scalars, use axes=().

`pixell.utils.`**`transpose_inds`**(*inds*, *nrow*, *ncol*)

Given a set of flattened indices into an array of shape (nrow,ncol), return the indices of the corresponding elemens in a transposed array.

`pixell.utils.`**`rescale`**(*a*, *range=[0, 1]*)

Rescale a such that min(a),max(a) -> range[0],range[1]

`pixell.utils.`**`split_by_group`**(*a*, *start*, *end*)

Split string a into non-group and group sections, where a group is defined as a set of characters from a start character to a corresponding end character.

`pixell.utils.`**`split_outside`**(*a*, *sep*, *start='([{'*, *end=')]}'*)

Split string a at occurences of separator sep, except when it occurs inside matching groups of start and end characters.

`pixell.utils.`**`find_equal_groups`**(*a*, *tol=0*)

Given a[nsamp,. . .], return groups[ngroup][{ind,ind,ind,. . . }] of indices into a for which all the values in the second index of a is the same. find_equal_groups([[0,1],[1,2],[0,1]]) -> [[0,2],[1]].

`pixell.utils.`**`find_equal_groups_fast`**(*vals*)

Group 1d array vals[n] into equal groups. Returns uvals, order, edges Using these, group #i is made up of the values with index order[edges[i]:edges[i+1]], and all these elements correspond to value uvals[i]. Accomplishes the same basic task as find_equal_groups, but 1. Only works on 1d arrays 2. Only works with exact quality, with no support for approximate equality 3. Returns 3 numpy arrays instead of a list of lists.

`pixell.utils.`**`label_multi`**(*valss*)

Given the argument valss[:][n], which is a list of 1d arrays of the same length n but potentially different data types, return a single 1d array labels[n] of integers such that unique lables correspond to unique valss[:]. More precisely, valss[:][labels[i]] == valss[:][labels[j]] only if labels[i] == labels[j]. The purpose of this is to go from having a heterogenous label like (1, "foo", 1.24) to having a single integer as the label.

Example: label_multi([[0,0,1,1,2],[“a”,”b”,”b”,”b”,”b”]]) → [0,1,2,2,3]

`pixell.utils.`**`pathsplit`**(*path*)

Like os.path.split, but for all components, not just the last one. Why did I have to write this function? It should have been in os already!

`pixell.utils.`**`minmax`**(*a*, *axis=None*)

Shortcut for np.array([np.min(a),np.max(a)]), since I do this a lot.

`pixell.utils.`**`broadcast_shape`**(**shapes*, *at=0*)

Return the shape resulting from broadcasting arrays with the given shapes. “at” controls how new axes are added. at=0 adds them at the beginning, which matches how numpy broadcasting works. at=1 would add them after the first element, etc. -1 adds them at the end.

`pixell.utils.`**`broadcast_arrays`**(**arrays*, *npre=0*, *npost=0*, *at=0*)

Like np.broadcast_arrays, but allows arrays to be None, in which case they are passed just passed through as None without affecting the rest of the broadcasting. The argument npre specifies the number of dimensions at the beginning of the arrays to exempt from broadcasting. This can be either an integer or a list of integers.

`pixell.utils.`**`point_in_polygon`**(*points*, *polys*)

Given a points[…,2] and a set of polys[…,nvertex,2], return inside[…]. points[…,0] and polys[…,0,0] must broadcast correctly.

Examples: utils.point_in_polygon([0.5,0.5],[[0,0],[0,1],[1,1],[1,0]]) -> True utils.point_in_polygon([[0.5,0.5],[2,1]],[[0,0],[0,1],[1,1],[1,0]]) -> [True, False]

`pixell.utils.`**`poly_edge_dist`**(*points*, *polygons*)

Given points […,2] and a set of polygons […,nvertex,2], return dists[…], which represents the distance of the points from the edges of the corresponding polygons. This means that the interior of the polygon will not be 0. points[…,0] and polys[…,0,0] must broadcast correctly.

`pixell.utils.`**`block_mean_filter`**(*a*, *width*)

Perform a binwise smoothing of a, where all samples in each bin of the given width are replaced by the mean of the samples in that bin.

`pixell.utils.`**`block_reduce`**(*a*, *bsize*, *axis=-1*, *off=0*, *op=<function mean>*, *inclusive=True*)

Replace each block of length bsize along the given axis of a with an aggregate value given by the operation op. op must accept op(array, axis), just like np.sum or np.mean. a need not have a whole number of blocks. In that case, the last block will have fewer than bsize samples in it. If off is specified, it gives an offset from the start of the array for the start of the first block; anything before that will be treated as an incomplete block, just like anything left over at the end. Pass the same value of off to block_expand to undo this.

`pixell.utils.`**`block_expand`**(*a*, *bsize*, *osize*, *axis=-1*, *off=0*, *op='nearest'*, *inclusive=True*)

The opposite of block_reduce. Where block_reduce averages (by default) this function duplicates (by default) to recover the original shape. If op=”linear”, then linear interpolation will be done instead of duplication. NOTE: Currently axis and orr are not supported for linear interpolation, which will always be done along the last axis.

`pixell.utils.`**`ctime2date`**(*timestamp*, *tzone=0*, *fmt='%Y-%m-%d'*)

`pixell.utils.`**`tofinite`**(*arr*, *val=0*)

Return arr with all non-finite values replaced with val.

`pixell.utils.`**`parse_ints`**(*s*)

`pixell.utils.`**`parse_floats`**(*s*)

`pixell.utils.`**`parse_numbers`**(*s*, *dtype=None*) `pixell.utils.`**`parse_box`**`(`*desc*`)`

Given a string of the form from:to,from:to,from:to,… returns an array [{from,to},:]

`pixell.utils.`**`triangle_wave`**`(`*x*, *period=1*`)`

Return a triangle wave with amplitude 1 and the given period.

`pixell.utils.`**`calc_beam_area`**`(`*beam_profile*`)`

Calculate the beam area in steradians given a beam profile[{r,b},npoint]. r is in radians, b should have a peak of 1..

`pixell.utils.`**`planck`**`(`*f*, *T=2.72548*`)`

Return the Planck spectrum at the frequency f and temperature T in Jy/sr

`pixell.utils.`**`blackbody`**`(`*f*, *T=2.72548*`)`

Return the Planck spectrum at the frequency f and temperature T in Jy/sr

`pixell.utils.`**`iplanck_T`**`(`*f*, *I*`)`

The inverse of planck with respect to temperature

`pixell.utils.`**`dplanck`**`(`*f*, *T=2.72548*`)`

The derivative of the planck spectrum with respect to temperature, evaluated at frequencies f and temperature T, in units of Jy/sr/K.

`pixell.utils.`**`graybody`**`(`*f*, *T=10*, *beta=1*`)`

Return a graybody spectrum at the frequency f and temperature T in Jy/sr

`pixell.utils.`**`tsz_spectrum`**`(`*f*, *T=2.72548*`)`

The increase in flux due to tsz in Jy/sr per unit of y. This is just the first order approximation, but it's good enough for realistic values of y, i.e. y << 1

`pixell.utils.`**`flux_factor`**`(`*beam_area*, *freq*, *T0=2.72548*`)`

Compute the factor A that when multiplied with a linearized temperature increment dT around T0 (in K) at the given frequency freq in Hz and integrated over the given beam_area in steradians, produces the corresponding flux = A*dT. This is useful for converting between point source amplitudes and point source fluxes.

For uK to mJy use flux_factor(beam_area, freq)/1e3

`pixell.utils.`**`noise_flux_factor`**`(`*beam_area*, *freq*, *T0=2.72548*`)`

Compute the factor A that converts from white noise level in K sqrt(steradian) to uncertainty in Jy for the given beam area in steradians and frequency in Hz. This assumes white noise and a gaussian beam, so that the area of the real-space squared beam is just half that of the normal beam area.

For uK arcmin to mJy, use noise_flux_factor(beam_area, freq)*arcmin/1e3

`pixell.utils.`**`gnfw`**`(`*x*, *xc*, *alpha*, *beta*, *gamma*`)`

`pixell.utils.`**`tsz_profile_raw`**`(`*x*, *xc=0.497*, *alpha=1.0*, *beta=-4.65*, *gamma=-0.3*`)`

Dimensionless radial (3d) cluster thermal pressure profile from arxiv:1109.3711. That article used a different definition of beta, beta' = 4.35 such that beta=gamma-alpha*beta'. I've translated it to follow the standard gnfw formula here. The numbers correspond to z=0, and M200 = 1e14 solar masses. They change slightly with mass and significantly with distance. But the further away they are, the smaller they get and the less the shape matters, so these should be good defaults.

The full dimensions are this number times P0*G*M200*200*rho_cr(z)*f_b/(2*R200) where P0=18.1 at z=0 and M200=1e14. To get the dimensionful electron pressure, further scale by (2+2*Xh)/(3+5*Xh), where Xh=0.76 is the hydrogen fraction. But if one is working in units of y, then the dimensionless version is enough.

x = r/R200. That is, it is the distance from the cluster center in units of the radius inside which the mean density is 200x as high as the critical density rho_c.

`pixell.utils.`**`tsz_profile_los`**(*x*, *xc=0.497*, *alpha=1.0*, *beta=-4.65*, *gamma=-0.3*, *zmax=100000.0*, *npoint=100*, *x1=1e-08*, *x2=10000.0*, *_a=8*, *cache=None*)

Fast, highly accurate approximate version of tsz_profile_los_exact. Interpolates the exact function in log-log space, and caches the interpolator. With the default settings, it's accurate to better than 1e-5 up to at least x = 10000, and building the interpolator takes about 25 ms. After that, each evaluation takes 50-100 ns per data point. This makes it about 10000x faster than tsz_profile_los_exact. See tsz_profile_raw for the units.

`pixell.utils.`**`tsz_profile_los_exact`**(*x*, *xc=0.497*, *alpha=1.0*, *beta=-4.65*, *gamma=-0.3*, *zmax=100000.0*, *_a=8*)

Line-of-sight integral of the cluster_pressure_profile. See tsz_profile_raw for the meaning of the arguments. Slow due to the use of quad and the manual looping this requires. Takes about 1 ms per data point. The argument _a controls a change of variable used to improve the speed and accuracy of the integral, and should not be necessary to change from the default value of 8.

See tsz_profile_raw for the units and how to scale it to something physical. Without scaling, the profile has a peak of about 0.5 and a FWHM of about 0.12 with the default parameters.

Instead of using this function directly, consider using tsz_profile_los instead. It's 10000x faster and more than accurate enough.

`pixell.utils.`**`tsz_tform`**(*r200=0.0002908882086657216*, *l=None*, *lmax=40000*, *xc=0.497*, *alpha=1.0*, *beta=-4.65*, *gamma=-0.3*, *zmax=100000.0*)

Return the radial spherical harmonic coefficients b(l) of the tSZ profile with the parameters xc, alpha, beta, gamma. Scale controls the angular size of the profile on the sky. r200 is the cluster's angular R200 size, in radians (default=1 arcmin).

If l (which can be multidimensional) is specified, the tsz coefficients will be evaluated at these ls. Otherwise l = np.arange(lmax+1) will be used.

The 2d-but-radially-symmetric fourier integral and cuspy nature of the tSZ profile are both handled via a fast hankel transform.

`pixell.utils.`**`edges2bins`**(*edges*)

`pixell.utils.`**`bins2edges`**(*bins*)

`pixell.utils.`**`linbin`**(*n*, *nbin=None*, *nmin=None*, *bsize=None*)

Given a number of points to bin and the number of approximately equal-sized bins to generate, returns [nbin_out,{from,to}]. nbin_out may be smaller than nbin. The nmin argument specifies the minimum number of points per bin, but it is not implemented yet. nbin defaults to the square root of n if not specified.

`pixell.utils.`**`expbin`**(*n*, *nbin=None*, *nmin=8*, *nmax=0*)

Given a number of points to bin and the number of exponentially spaced bins to generate, returns [nbin_out,{from,to}]. nbin_out may be smaller than nbin. The nmin argument specifies the minimum number of points per bin. nbin defaults to n**0.5

`pixell.utils.`**`bin_data`**(*bins*, *d*, *op=<function mean>*)

Bin the data d into the specified bins along the last dimension. The result has shape d.shape[:-1] + (nbin,).

`pixell.utils.`**`bin_expand`**(*bins*, *bdata*)

`pixell.utils.`**`is_int_valued`**(*a*)

`pixell.utils.`**`solve`**(*A*, *b*, *axes=[-2, -1]*, *masked=False*)

Solve the linear system Ax=b along the specified axes for A, and axes[0] for b. If masked is True, then entries where A00 along the given axes is zero will be skipped.

pixell.utils.**eigpow**(*A*, *e*, *axes=[-2, -1]*, *rlim=None*, *alim=None*)

Compute the e'th power of the matrix A (or the last two axes of A for higher-dimensional A) by exponentiating the eigenvalues. A should be real and symmetric.

When e is not a positive integer, negative eigenvalues could result in a complex result. To avoid this, negative eigenvalues are set to zero in this case.

Also, when e is not positive, tiny eigenvalues dominated by numerical errors can be blown up enough to drown out the well-measured ones. To avoid this, eigenvalues smaller than 1e-13 for float64 or 1e-4 for float32 of the largest one (rlim), or with an absolute value less than 2e-304 for float64 or 1e-34 for float32 (alim) are set to zero for negative e. Set alim and rlim to 0 to disable this behavior.

pixell.utils.**build_conditional**(*ps*, *inds*, *axes=[0, 1]*)

Given some covariance matrix ps[n,n] describing a set of n Gaussian distributed variables, and a set of indices inds[m] specifying which of these variables are already known, return matrices A[n-m,m], cov[m,m] such that the conditional distribution for the unknown variables is x_unknown ~ normal(A x_known, cov). If ps has more than 2 dimensions, then the axes argument indicates which dimensions contain the matrix.

Example:

C = np.array([[10,2,1],[2,8,1],[1,1,5]]) vknown = np.linalg.cholesky(C[:1,:1]).dot(np.random.standard_normal(1))
A, cov = lensing.build_conditional(C, v0) vrest = A.dot(vknown) + np.linalg.cholesky(cov).dot(np.random_standard_normal(2))

vtot = np.concatenate([vknown,vrest]) should have the same distribution as a sample drawn directly from the full C.

pixell.utils.**nint**(*a*)

Return a rounded to the nearest integer, as an integer.

pixell.utils.**ceil**(*a*)

Return a rounded to the next integer, as an integer.

pixell.utils.**floor**(*a*)

Return a rounded to the previous integer, as an integer.

pixell.utils.**format_to_glob**(*format*)

Given a printf format, construct a glob pattern that will match its outputs. However, since globs are not very powerful, the resulting glob will be much more premissive than the input format, and you will probably want to filter the results further.

pixell.utils.**format_to_regex**(*format*)

Given a printf format, construct a regex that will match its outputs.

class pixell.utils.**Printer**(*level=1*, *prefix=''*)

write(*desc*, *level*, *exact=False*, *newline=True*, *prepend=''*)

push(*desc*)

time(*desc*, *level*, *exact=False*, *newline=True*)

pixell.utils.**ndigit**(*num*)

Returns the number of digits in non-negative number num

pixell.utils.**contains_any**(*a*, *bs*)

Returns true if any of the strings in list bs are found in the string a

pixell.utils.**build_legendre**(*x*, *nmax*) pixell.utils.**build_cossin**(*x*, *nmax*)

pixell.utils.**load_ascii_table**(*fname*, *desc*, *sep=None*, *dsep=None*)

Load an ascii table with heterogeneous columns. fname: Path to file desc: whitespace-separated list of name:typechar pairs, or | for columns that are to be ignored. desc must cover every column present in the file

pixell.utils.**count_variable_basis**(*bases*)

Counts from 0 and up through a variable-basis number, where each digit has a different basis. For example, count_variable_basis([2,3]) would yield [0,0], [0,1], [0,2], [1,0], [1,1], [1,2].

pixell.utils.**list_combination_iter**(*ilist*)

Given a list of lists of values, yields every combination of one value from each list.

pixell.utils.**expand_slice**(*sel*, *n*, *nowrap=False*)

Expands defaults and negatives in a slice to their implied values. After this, all entries of the slice are guaranteed to be present in their final form. Note, doing this twice may result in odd results, so don't send the result of this into functions that expect an unexpanded slice. Might be replacable with slice.indices().

pixell.utils.**split_slice**(*sel*, *ndims*)

Splits a numpy-compatible slice "sel" into sub-slices sub[:], such that a[sel] = s[sub[0]][:,sub[1]][:,:,sub[2]][. . .], This is useful when implementing arrays with heterogeneous indices. Ndims indicates the number of indices to allocate to each split, starting from the left. Also expands all ellipsis.

pixell.utils.**split_slice_simple**(*sel*, *ndims*)

Helper function for split_slice. Splits a slice in the absence of ellipsis.

pixell.utils.**parse_slice**(*desc*)

pixell.utils.**slice_downgrade**(*d*, *s*, *axis=-1*)

Slice array d along the specified axis using the Slice s, but interpret the step part of the slice as downgrading rather than skipping.

pixell.utils.**outer_stack**(*arrays*)

Example. outer_stack([[1,2,3],[10,20]]) -> [[[1,1],[2,2],[3,3]],[[10,20],[10,20],[10,2]]]

pixell.utils.**tform_to_profile**(*bl*, *theta*, *normalize=False*)

Given the transform b(l) of a beam, evaluate its real space angular profile at the given radii theta.

pixell.utils.**beam_transform_to_profile**(*bl*, *theta*, *normalize=False*)

Given the transform b(l) of a beam, evaluate its real space angular profile at the given radii theta.

class pixell.utils.**RadialFourierTransform**(*lrange=None*, *rrange=None*, *n=512*, *pad=256*)

real2harm(*rprof*)

Perform a forward (real -> harmonic) transform, taking us from the provided real-space radial profile rprof(r) to a harmonic-space profile lprof(l). rprof can take two forms: 1. A function rprof(r) that can be called to evalute the profile at

arbitrary points.

2. An array rprof[self.r] that provides the profile evaluated at the points given by this object's .r member.

The transform is done along the last axis of the profile. Returns lprof[self.l]. This includes padding, which can be removed using self.unpad

harm2real(*lprof*)

Perform a backward (harmonic -> real) transform, taking us from the provided harmonic-space radial profile lprof(l) to a real-space profile rprof(r). lprof can take two forms: 1. A function lprof(l) that can be called to evalute the profile at

arbitrary points.

2. An array lprof[self.l] that provides the profile evaluated at the points given by this object's .l member.

The transform is done along the last axis of the profile. Returns rprof[self.r]. This includes padding, which can be removed using self.unpad

unpad(**arrs*)

Remove the padding from arrays used by this object. The values in the padded areas of the output of the transform have unreliable values, but they're not cropped automatically to allow for round-trip transforms. Example:

r = unpad(r_padded) r, l, vals = unpad(r_padded, l_padded, vals_padded)

lind(*l*)

rind(*r*)

pixell.utils.**profile_to_tform_hankel**(*profile_fun*, *lmin=0.1*, *lmax=10000000.0*, *n=512*, *pad=256*)

Transform a radial profile given by the function profile_fun(r) to sperical harmonic coefficients b(l) using a Hankel transform. This approach is good at handling cuspy distributions due to using logarithmically spaced points. n points from 10**logrange[0] to 10**logrange[1] will be used. Returns l, bl. l will not be equi-spaced, so you may want to interpolate the results. Note that unlike other similar functions in this module and the curvedsky module, this function uses the flat sky approximation, so it should only be used for profiles up to a few degrees in size.

pixell.utils.**fix_dtype_mpi4py**(*dtype*)

Work around mpi4py bug, where it refuses to accept dtypes with endian info

pixell.utils.**native_dtype**(*dtype*)

Return the native version of dtype. E.g. if the input is big-endian float32, returns plain float32

pixell.utils.**decode_array_if_necessary**(*arr*)

Given an arbitrary numpy array arr, decode it if it is of type S and we're in a version of python that doesn't like that

pixell.utils.**encode_array_if_necessary**(*arr*)

Given an arbitrary numpy array arr, encode it if it is of type S and we're in a version of python that doesn't like that

pixell.utils.**to_sexa**(*x*)

Given a number in decimal degrees x, returns (sign,deg,min,sec). Given this x can be reconstructed as sign*(deg+min/60+sec/3600).

pixell.utils.**from_sexa**(*sign*, *deg*, *min*, *sec*)

Reconstruct a decimal number from the sexagesimal representation.

pixell.utils.**format_sexa**(*x*, *fmt='%(deg)+03d:%(min)02d:%(sec)06.2f'*)

pixell.utils.**jname**(*ra*, *dec*, *fmt='J%(ra_H)02d%(ra_M)02d%(ra_S)02d%(dec_d)+02d%(dec_m)02d%(dec_s)02d'*, *tag=None*, *sep=' '*)

Build a systematic object name for the given ra/dec in degrees. The format is specified using the format string fmt. The default format string is 'J%(ra_H)02d%(ra_M)02d%(ra_S)02d%(dec_d)+02d%(dec_m)02d%(dec_s)02d'. This is not fully compliant with the IAU specification, but it's what is used in ACT. Formatting uses standard python string interpolation. The available variables are ra: right ascension in decimal degrees dec: declination in decimal degrees ra_d, ra_m, ra_s: sexagesimal degrees, arcmins and arcsecs of right ascensions dec_d, dec_m, dec_s: sexagesimal degrees, arcmins and arcsecs of declination ra_H, ra_M, ra_S: hours, minutes and seconds of right ascension dec_H, rec_M, dec_S: hours, minutes and seconds of declination (doesn't make much sense)

tag is prefixed to the format, with sep as the separator. This lets one prefix the survey name without needing to rewrite the whole format string.

`pixell.utils.`**`ang2chord`**`(`*ang*`)`

Converts from the angle between two points on a circle to the length of the chord between them

`pixell.utils.`**`chord2ang`**`(`*chord*`)`

Inverse of ang2chord.

`pixell.utils.`**`crossmatch`**`(`*pos1*, *pos2*, *rmax*, *mode='closest'*, *coords='auto'*`)`

Find close matches between positions given by pos1[:,ndim] and pos2[:,ndim], up to a maximum distance of rmax (in the same units as the positions).

The argument "coords" controls how the coordinates are interpreted. If it is "cartesian", then they are assumed to be cartesian coordinates. If it is "radec" or "phitheta", then the coordinates are assumed to be angles in radians, which will be transformed to cartesian coordinates internally before being used. "radec" is equator-based while "phitheta" is zenith-based. The default, "auto", will assume "radec" if ndim == 2, and "cartesian" otherwise.

It's possible that multiple objects from the catalogs are within rmax of each other. The "mode" argument controls how this is handled. mode == "all":

> Returns a list of pairs of indices into the two lists, one for each pair of objects that are close enough to each other, regardless of the presence of any other matches. Any given object can be mentioned multiple times in this list.

mode == "closest":
Like "all", but only the closest time an index appears in a pair is kept, the others are discarded.

mode == "first":
Like "all", but only the first time an index appears in a pair is kept, the others are discarted. This can be useful if some objects should be given higher priority than others. For example, one could sort pos1 and pos2 by brightness and then use mode == "first" to prefer bright objects in the match.

`pixell.utils.`**`real_dtype`**`(`*dtype*`)`

Return the closest real (non-complex) dtype for the given dtype

`pixell.utils.`**`complex_dtype`**`(`*dtype*`)`

Return the closest complex dtype for the given dtype

`pixell.utils.`**`ascomplex`**`(`*arr*`)`

`pixell.utils.`**`default_M`**`(`*x*`)`

`pixell.utils.`**`default_dot`**`(`*a*, *b*`)`

class `pixell.utils.`**`CG`**`(`*A*, *b*, *x0=None*, *M=<function default_M>*, *dot=<function default_dot>*`)`

A simple Preconditioner Conjugate gradients solver. Solves the equation system Ax=b.

This improves on the one in scipy in several ways. It allows one to specify one's own dot product operator, which is necessary for handling distributed degrees of freedom, where each mpi task only stores parts of the full solution. It is also reentrant, meaning that one can do nested CG if necessary.

`step()`

Take a single step in the iteration. Results in .x, .i and .err being updated. To solve the system, call step() in a loop until you are satisfied with the accuracy. The result can then be read off from .x.

`save(`*fname*`)`

Save the volatile internal parameters to hdf file fname. Useful for later restoring cg iteration

`load(`*fname*`)`

Load the volatile internal parameters from the hdf file fname. Useful for restoring a saved cg state, after first initializing the object normally.

`class pixell.utils.Minres(`*A*, *b*, *x0=None*, *dot=<function default_dot>*`)`

A simple Minres solver. Solves the equation system Ax=b.

`step()`

Take a single step in the iteration. Results in .x, .i and .err being updated. To solve the system, call step() in a loop until you are satisfied with the accuracy. The result can then be read off from .x.

`pixell.utils.nditer(`*shape*`)`

`pixell.utils.first_importable(`**args*`)`

Given a list of module names, return the name of the first one that can be imported.

`pixell.utils.glob(`*desc*`)`

Like glob.glob, but without nullglob turned on. This is useful for not just silently throwing away arguments with spelling mistakes.

`pixell.utils.cache_get(`*cache*, *key*, *op*`)`

`pixell.utils.replace(`*istr*, *ipat*, *repl*`)`

`pixell.utils.remove_nan(`*a*`)`

Sets nans and infs to 0 in an array in-place. Should have no memory overhead. Also returns the array for convenience.

`pixell.utils.without_nan(`*a*`)`

Returns a copy of a with nans and infs set to 0. The original array is not modified.

`pixell.utils.primes(`*n*`)`

Simple prime factorization of the positive integer n. Uses the brute force algorithm, but it's quite fast even for huge numbers.

`pixell.utils.res2nside(`*res*`)`

`pixell.utils.nside2res(`*nside*`)`

`pixell.utils.split_esc(`*string*, *delim*, *esc='\\'*`)`

Split string by the delimiter except when escaped by the given escape character, which defaults to backslash. Consumes one level of escapes. Yields the tokens one by one as an iterator.

`pixell.utils.getenv(`*name*, *default=None*`)`

Return the value of the named environment variable, or default if it's not set `pixell.utils.`**`setenv`**(*name*, *value*, *keep=False*)

Set the named environment variable to the given value. If keep==False (the default), existing values are overwritten. If the value is None, then it's deleted from the environment. If keep==True, then this function does nothing if the variable already has a value.

`pixell.utils.`**`zip2`**(**args*)

Variant of python's zip that calls next() the same number of times on all arguments. This means that it doesn't give up immediately after getting the first StopIteration exception, but continues on until the end of the row. This can be useful for iterators that want to do cleanup after hte last yield.

`pixell.utils.`**`call_help`**(*fun*, **args*, ***kwargs*)

`pixell.utils.`**`arg_help`**(*arg*)

3.5 2.5 reproject - Map reprojection

`pixell.reproject.`**`thumbnails`**(*imap*, *coords*, *r=0.0014544410433286808*, *res=None*, *proj=None*, *apod=0.0005817764173314432*, *order=3*, *oversample=4*, *pol=None*, *oshape=None*, *owcs=None*, *extensive=False*, *verbose=False*, *filter=None*, *pixwin=False*, *pixwin_order=0*)

Given an enmap [...,ny,nx] and a set of coordinates in a numpy array coords with shape (n,2) and ordering [n,{dec,ra}], extract a set of thumbnail images [n,...,thumby,thumbx] centered on each set of coordinates, where n is the number of images. When proj is 'tan', each of these thumbnail images is projected onto a local tangent plane, removing the effect of size and shape distortions in the input map.

If oshape, owcs are specified, then the thumbnails will have this geometry, which should be centered on [0,0]. Otherwise, a geometry with the given projection (defaults to "tan" = gnomonic projection) will be constructed, going up to a maximum radius of r. FIXME: Defaults to "car" instead while enmap.pixsizemap is buggy for non-cylindrical projections.

The reprojection involved in this operation implies interpolation. The default is to use fft rescaling to oversample the input pixels by the given pixel, and then use bicubic spline interpolation to read off the values at the output pixel centers. The fft oversampling can be controlled with the oversample argument. Values <= 1 turns this off. The other interpolation step is controlled using the "order" argument. 0/1/3 corresponds to nearest neighbor, bilinear and bicubic spline interpolation respectively.

If pol == True, then Q,U will be rotated to take into account the change in the local northward direction impled in the reprojection. The default is to do polarization rotation automatically if the input map has a compatible shape, e.g. at least 3 axes and a length of 3 for the 3rd last one. TODO: I haven't tested this yet.

If extensive == True (not the default), then the map is assumed to contain an extensive field rather than an intensive one. An extensive field is one where the values in the pixels depend on the size of the pixel. For example, if the inverse variance in the map is given per pixel, then this ivar map will be extensive, but if it's given in units of inverse variance per square arcmin then it's intensive.

For reprojecting inverse variance maps, consider using the wrapper thumbnails_ivar, which makes it easier to avoid common pitfalls.

If pixwin is True, the pixel window will be deconvolved.

`pixell.reproject.`**`thumbnails_ivar`**(*imap*, *coords*, *r=0.0014544410433286808*, *res=None*, *proj=None*, *oshape=None*, *owcs=None*, *order=1*, *extensive=True*, *verbose=False*)

Like thumbnails, but for hitcounts, ivars, masks, and other quantities that should stay positive and local. Remember to set extensive to True if you have an extensive quantity, i.e. if the values in each pixel would go up if multiple pixels combined. An example of this is a hitcount map or ivar per pixel. Conversely, if you have an intensive quantity like ivar per arcmin you should set extensive=False.

pixell.reproject.**map2healpix**(*imap*, *nside=None*, *lmax=None*, *out=None*, *rot=None*, *spin=[0, 2]*, *method='harm'*, *order=1*, *extensive=False*, *bsize=100000*, *nside_mode='pow2'*, *boundary='constant'*, *verbose=False*, *niter=0*)

Reproject from an enmap to healpix, optionally including a rotation.

imap: The input enmap[. . . ,ny,nx]. Stokes along the -3rd axis if
present.

nside: The nside of the healpix map to generate. Not used if
an output map is passed. Otherwise defaults to the same resolution as the input map.

lmax: The highest multipole to use in any harmonic-space
operations. Defaults to the input maps' Nyquist limit.

out: An optional array [. . . ,npix] to write the output map to.
The . . . part must match the input map, as must the data type.

rot: An optional coordinate rotation to apply. Either a string
"isys,osys", where isys is the system to transform from, and osys is the system to transform to. Currently the values "cel"/"equ" and "gal" are recognized. Alternatively, a tuple of 3 euler zyz euler angles can be passed, in the same convention as healpy.rotate_alm.

spin: A description of the spin of the entries along the stokes
axis. Defaults to [0,2], which means that the first entry is spin-0, followed by a spin-2 pair (any non-zero spin covers a pair of entries). If the axis is longer than what's covered in the description, then it is repeated as necessary. Pass spin=[0] to disable any special treatment of this axis.

method: How to interpolate between the input and output
pixelizations. Can be "harm" (default) or "spline". "harm" maps between them using spherical harmonics transforms. This preserves the power spectrum (so no window function is introduced), and averages noise down when the output pixels are larger than the input pixels. However, it can suffer from ringing around very bright features, and an all-positive input map may end up with small negative values. "spline" instead uses spline interpolation to look up the value in the intput map corresponding to each pixel center in the output map. The spline order is controlled with the "order" argument. Overall "harm" is best suited for normal sky maps, while "spline" with order = 0 or 1 is best suited for hitcount maps and masks.

order: The spline order to use when method="spline".
0 corresponds to nearest neighbor interpolation. 1 corresponds to bilinear interpolation (default) 3 corresponds to bicubic spline interpolation. 0 and 1 are local and do not introduce values outside the input range, but introduce some aliasing and loss of power. 3 has less power loss, but still non-zero, and is vulnerable to ringing.

extensive: Whether the map represents an extensive (as opposed to
intensive) quantity. Extensive quantities have values proportional to the pixel size, unlike intensive quantities. Hitcount per pixel is an extensive quantity. Hitcount per square degree is an intensive quantity, as is a temperature map. Defaults to False.

bsize: The spline method operates on batches of pixels to save memory.
This controls the batch size, in pixels. Defaults to 100000.

nside_mode: Controls which restrictions apply to nside in the case where
it has to be inferred automatically. Can be "pow2", "mul32" and "any". "pow2", the default, results in nside being a power of two, as required by the healpix standard. "mul32" relaxes this requirement, making a map where nside is a multiple of 32. This is compatible with most healpix operations, but not with ud_grade or the nest pixel ordering. "any" allows for any integer nside.

boundary: The boundary conditions assumed for the input map when
method="spline". Defaults to "constant", which assumes that anything outsize the map has a constant value of 0. Another useful value is "wrap", which assumes that the right side wraps over to the left, and

the top to the bottom. See scipy.ndimage.distance_transform's documentation for other, less useful values. method="harm" always assumes "constant" regardless of this setting.

verbose: Whether to print information about what it's doing.
Defaults to False, which doesn't print anything.

Typical usage: * map_healpix = map2healpix(map, rot="cel,gal") * ivar_healpix = map2healpix(ivar, rot="cel,gal", method="spline", spin=[0], extensive=True)

`pixell.reproject.`**`healpix2map`**(*iheal*, *shape=None*, *wcs=None*, *lmax=None*, *out=None*, *rot=None*, *spin=[0, 2]*, *method='harm'*, *order=1*, *extensive=False*, *bsize=100000*, *verbose=False*, *niter=0*)

Reproject from healpix to an enmap, optionally including a rotation.

iheal: The input healpix map […,npix]. Stokes along the -2nd axis if
present.

shape: The (…,ny,nx) shape of the output map. Only the last two entries
are used, the rest of the dimensions are taken from iheal. Mandatory unless an output map is passed.

wcs
[The world woordinate system object the output map.] Mandatory unless an output map is passed.

lmax: The highest multipole to use in any harmonic-space
operations. Defaults to 3 times the nside of iheal.

out: An optional enmap […,ny,nx] to write the output map to.
The … part must match iheal, as must the data type.

rot: An optional coordinate rotation to apply. Either a string
"isys,osys", where isys is the system to transform from, and osys is the system to transform to. Currently the values "cel"/"equ" and "gal" are recognized. Alternatively, a tuple of 3 euler zyz euler angles can be passed, in the same convention as healpy.rotate_alm.

spin: A description of the spin of the entries along the stokes
axis. Defaults to [0,2], which means that the first entry is spin-0, followed by a spin-2 pair (any non-zero spin covers a pair of entries). If the axis is longer than what's covered in the description, then it is repeated as necessary. Pass spin=[0] to disable any special treatment of this axis.

method: How to interpolate between the input and output
pixelizations. Can be "harm" (default) or "spline". "harm" maps between them using spherical harmonics transforms. This preserves the power spectrum (so no window function is introduced), and averages noise down when the output pixels are larger than the input pixels. However, it can suffer from ringing around very bright features, and an all-positive input map may end up with small negative values. "spline" instead uses spline interpolation to look up the value in the intput map corresponding to each pixel center in the output map. The spline order is controlled with the "order" argument. Overall "harm" is best suited for normal sky maps, while "spline" with order = 0 or 1 is best suited for hitcount maps and masks.

order: The spline order to use when method="spline".
0 corresponds to nearest neighbor interpolation. 1 corresponds to bilinear interpolation (default) Higher order interpolation is not supported - use method="harm" for that.

extensive: Whether the map represents an extensive (as opposed to
intensive) quantity. Extensive quantities have values proportional to the pixel size, unlike intensive quantities. Hitcount per pixel is an extensive quantity. Hitcount per square degree is an intensive quantity, as is a temperature map. Defaults to False.

bsize: The spline method operates on batches of pixels to save memory.
This controls the batch size, in pixels. Defaults to 100000.

verbose: Whether to print information about what it's doing.
Defaults to False, which doesn't print anything.

Typical usage: * map = healpix2map(map_healpix, shape, wcs, rot="gal,cel") * ivar = healpix2map(ivar_healpix, shape, wcs, rot="gal,cel", method="spline", spin=[0], extensive=True)

`pixell.reproject.`**`rot2euler`**`(`*rot*`)`

Given a coordinate rotation description, return the [rotz,roty,rotz] euler angles it corresponds to. The rotation desciption can either be those angles directly, or a string of the form isys,osys

`pixell.reproject.`**`inv_euler`**`(`*euler*`)`

`pixell.reproject.`**`restrict_nside`**`(`*nside*, *mode='mul32'*, *round='ceil'*`)`

Given an arbitrary Healpix nside, return one that's restricted in various ways according to the "mode" argument:

"pow2": Restrict to a power of 2. This is required for compatibility
with the rarely used "nest" pixel ordering in Healpix, and is the standard in the Healpix world.

"mul32": Restrict to multiple of 32, unless 12*nside2<=1024.**
This is enough to make the maps writable by healpy.

"any": No restriction

The "round" argument controls how any rounding is done. This can be one of the strings "ceil" (default), "round" or "floor", or you can pass in a custom function(nside) -> nside.

In all cases, the final nside is converted to an integer and capped to 1 below.

`pixell.reproject.`**`centered_map`**`(`*imap*, *res*, *box=None*, *pixbox=None*, *proj='car'*, *rpix=None*, *width=None*, *height=None*, *width_multiplier=1.0*, *rotate_pol=True*, ***kwargs*`)`

Reproject a map such that its central pixel is at the origin of a given projection system (default: CAR).

imap – (Ny,Nx) enmap array from which to extract stamps TODO: support leading dimensions res – width of pixel in radians box – optional bounding box of submap in radians pixbox – optional bounding box of submap in pixel numbers proj – coordinate system for target map; default is 'car'; can also specify 'cea' or 'gnomonic' rpix – optional pre-calculated pixel positions from get_rotated_pixels()

`pixell.reproject.`**`healpix_from_enmap_interp`**`(`*imap*, ***kwargs*`)`

`pixell.reproject.`**`healpix_from_enmap`**`(`*imap*, *lmax*, *nside*`)`

Convert an ndmap to a healpix map such that the healpix map is band-limited up to lmax. Only supports single component (intensity) currently. The resulting map will be band-limited. Bright sources and sharp edges could cause ringing. Use healpix_from_enmap_interp if you are worried about this (e.g. for a mask), but that routine will not ensure power to be correct to some lmax.

Parameters

- **`imap`** – ndmap of shape (Ny,Nx)
- **`lmax`** – integer specifying maximum multipole of map
- **`nside`** – integer specifying nside of healpix map

Returns
(Npix,) healpix map as array

Return type
retmap

`pixell.reproject.`**`enmap_from_healpix`**`(`*hp_map*, *shape*, *wcs*, *ncomp=1*, *unit=1*, *lmax=0*, *rot='gal,equ'*, *first=0*, *is_alm=False*, *return_alm=False*, *f_ell=None*`)`

Convert a healpix map to an ndmap using harmonic space reprojection. The resulting map will be band-limited. Bright sources and sharp edges could cause ringing. Use enmap_from_healpix_interp if you are worried about this (e.g. for a mask), but that routine will not ensure power to be correct to some lmax.

Parameters

- **hp_map** – an (Npix,) or (ncomp,Npix,) healpix map, or alms, or a string containing
- **disk** (*the path to a healpix map on*) –
- **shape** – the shape of the ndmap geometry to project to
- **wcs** – the wcs object of the ndmap geometry to project to
- **ncomp** – the number of components in the healpix map (either 1 or 3)
- **unit** – a unit conversion factor to divide the map by
- **lmax** – the maximum multipole to include in the reprojection
- **rot** – comma separated string that specify a coordinate rotation to
- **"gal** (*perform. Use None to perform no rotation. e.g. default*) –
- **equ"** –
- **equatorial** (*to rotate a Planck map in galactic coordinates to the*) –
- **ndmaps.** (*coordinates used in*) –
- **first** – if a filename is provided for the healpix map, this specifies
- **field** (*the index of the first FITS*) –
- **is_alm** – if True, interprets hp_map as alms
- **return_alm** – if True, returns alms also
- **f_ell** – optionally apply a transfer function f_ell(ell) – this should be
- **e.g.** (*a function of a single variable ell.*) – exp(-x**2/2/sigma**2)
- **x** (*lambda*) – exp(-x**2/2/sigma**2)

Returns
the reprojected ndmap or the a tuple (ndmap,alms) if return_alm is True

Return type
res

`pixell.reproject.`**enmap_from_healpix_interp**(*hp_map*, *shape*, *wcs*, *rot='gal,equ'*, *interpolate=False*)

Project a healpix map to an enmap of chosen shape and wcs. The wcs is assumed to be in equatorial (ra/dec) coordinates. No coordinate systems other than equatorial or galactic are currently supported. Only intensity maps are supported.

Parameters

- **hp_map** – an (Npix,) healpix map
- **shape** – the shape of the ndmap geometry to project to
- **wcs** – the wcs object of the ndmap geometry to project to
- **rot** – comma separated string that specify a coordinate rotation to
- **"gal** (*perform. Use None to perform no rotation. e.g. default*) –
- **equ"** –

- **equatorial** (*to rotate a Planck map in galactic coordinates to the*) –
- **ndmaps.** (*coordinates used in*) –
- **interpolate** – if True, bilinear interpolation using 4 nearest neighbours
- **done.** (*is*) –

`pixell.reproject.`**`ivar_hp_to_cyl`**(*hmap*, *shape*, *wcs*, *rot=False*, *do_mask=True*, *extensive=True*)

`pixell.reproject.`**`gnomonic_pole_wcs`**(*shape*, *res*)

`pixell.reproject.`**`gnomonic_pole_geometry`**(*width*, *res*, *height=None*)

`pixell.reproject.`**`rotate_map`**(*imap*, *shape_target=None*, *wcs_target=None*, *shape_source=None*, *wcs_source=None*, *pix_target=None*, ***kwargs*)

`pixell.reproject.`**`get_rotated_pixels`**(*shape_source*, *wcs_source*, *shape_target*, *wcs_target*, *inverse=False*, *pos_target=None*, *center_target=None*, *center_source=None*)

Given a source geometry (shape_source,wcs_source) return the pixel positions in the target geometry (shape_target,wcs_target) if the source geometry were rotated such that its center lies on the center of the target geometry.

WARNING: Only currently tested for a rotation along declination from one CAR geometry to another CAR geometry.

`pixell.reproject.`**`cutout`**(*imap*, *width=None*, *ra=None*, *dec=None*, *pad=1*, *corner=False*, *res=None*, *npix=None*, *return_slice=False*, *sindex=None*)

`pixell.reproject.`**`rect_box`**(*width*, *center=(0.0, 0.0)*, *height=None*)

`pixell.reproject.`**`get_pixsize_rect`**(*shape*, *wcs*)

Return the exact pixel size in steradians for the rectangular cylindrical projection given by shape, wcs. Returns area[ny], where ny = shape[-2] is the number of rows in the image. All pixels on the same row have the same area.

`pixell.reproject.`**`rect_geometry`**(*width*, *res*, *height=None*, *center=(0.0, 0.0)*, *proj='car'*)

`pixell.reproject.`**`distribute`**(*N*, *nmax*)

Distribute N things into cells as equally as possible such that no cell has more than nmax things.

`pixell.reproject.`**`populate`**(*shape*, *wcs*, *ofunc*, *maxpixy=400*, *maxpixx=400*)

Loop through tiles in a new map of geometry (shape,wcs) with tiles that have maximum allowed shape (maxpixy,maxpixx) such that each tile is populated with the result of ofunc(oshape,owcs) where oshape,owcs is the geometry of each tile.

`pixell.reproject.`**`postage_stamp`**(*inmap*, *ra_deg*, *dec_deg*, *width_arcmin*, *res_arcmin*, *proj='gnomonic'*, *return_cutout=False*, *npad=3*, *rotate_pol=True*, ***kwargs*)

3.6 2.6 resample - Map resampling

This module handles resampling of time-series and similar arrays.

`pixell.resample.`**`resample`**`(`*d*, *factors=[0.5]*, *axes=None*, *method='fft'*`)`

`pixell.resample.`**`resample_bin`**`(`*d*, *factors=[0.5]*, *axes=None*`)`

`pixell.resample.`**`downsample_bin`**`(`*d*, *steps=[2]*, *axes=None*`)`

`pixell.resample.`**`upsample_bin`**`(`*d*, *steps=[2]*, *axes=None*`)`

`pixell.resample.`**`resample_fft`**`(`*d*, *n*, *axes=None*`)`

Resample numpy array d via fourier-reshaping. Requires periodic data. n indicates the desired output lengths of the axes that are to be resampled. By default the last len(n) axes are resampled, but this can be controlled via the axes argument.

`pixell.resample.`**`resample_fft_simple`**`(`*d*, *n*, *ngroup=100*`)`

Resample 2d numpy array d via fourier-reshaping along last axis.

`pixell.resample.`**`make_equispaced`**`(`*d*, *t*, *quantile=0.1*, *order=3*, *mask_nan=False*`)`

Given an array d[…,nt] of data that has been sampled at times t[nt], return an array that has been resampled to have a constant sampling rate.

3.7 2.7 lensing - Lensing

`pixell.lensing.`**`lens_map`**`(`*imap*, *grad_phi*, *order=3*, *mode='spline'*, *border='cyclic'*, *trans=False*, *deriv=False*, *h=1e-07*`)`

Lens map imap[{pre},ny,nx] according to grad_phi[2,ny,nx], where phi is the lensing potential, and grad_phi, which can be computed as enmap.grad(phi), simply is the coordinate displacement for each pixel. order, mode and border specify details of the interpolation used. See enlib.interpol.map_coordinates for details. If trans is true, the transpose operation is performed. This is NOT equivalent to delensing.

If the same lensing field needs to be reused repeatedly, then higher efficiency can be gotten from calling displace_map directly with precomputed pixel positions.

`pixell.lensing.`**`delens_map`**`(`*imap*, *grad_phi*, *nstep=3*, *order=3*, *mode='spline'*, *border='cyclic'*`)`

The inverse of lens_map, such that delens_map(lens_map(imap, dpos), dpos) = imap for well-behaved fields. The inverse does not always exist, in which case the equation above will only be approximately fulfilled. The inverse is computed by iteration, with the number of steps in the iteration controllable through the nstep parameter. See enlib.interpol.map_coordinates for details on the other parameters.

`pixell.lensing.`**`delens_grad`**`(`*grad_phi*, *nstep=3*, *order=3*, *mode='spline'*, *border='cyclic'*`)`

Helper function for delens_map. Attempts to find the undisplaced gradient given one that has been displaced by itself.

`pixell.lensing.`**`displace_map`**`(`*imap*, *pix*, *order=3*, *mode='spline'*, *border='cyclic'*, *trans=False*, *deriv=False*`)`

Displace map m[{pre},ny,nx] by pix[2,ny,nx], where pix indicates the location in the input map each output pixel should get its value from (float). The output is [{pre},ny,nx].

`pixell.lensing.`**`lens_map_flat`**`(`*cmb_map*, *phi_map*`)` `pixell.lensing.`**`phi_to_kappa`**(*phi_alm*, *phi_ainfo=None*)

Convert lensing potential alms phi_alm to lensing convergence alms kappa_alm, i.e. phi_alm * l * (l+1) / 2

Parameters

- **phi_alm** – (. . . ,N) ndarray of spherical harmonic alms of lensing potential
- **phi_ainfo** – If ainfo is provided, it is an alm_info describing the layout

of the input alm. Otherwise it will be inferred from the alm itself.

Returns
The filtered alms phi_alm * l * (l+1) / 2

Return type
kappa_alm

`pixell.lensing.`**`lens_map_curved`**(*shape, wcs, phi_alm, cmb_alm, phi_ainfo=None, maplmax=None, dtype=<class 'numpy.float64'>, spin=[0, 2], output='l', geodesic=True, verbose=False, delta_theta=None*)

`pixell.lensing.`**`rand_alm`**(*ps_lensinput*, *lmax=None*, *dtype=<class 'numpy.float64'>*, *seed=None*, *phi_seed=None*, *verbose=False*, *ncomp=None*)

`pixell.lensing.`**`rand_map`**(*shape, wcs, ps_lensinput, lmax=None, maplmax=None, dtype=<class 'numpy.float64'>, seed=None, phi_seed=None, spin=[0, 2], output='l', geodesic=True, verbose=False, delta_theta=None*)

`pixell.lensing.`**`offset_by_grad`**(*ipos*, *grad*, *geodesic=True*, *pol=None*)

Given a set of coordinates ipos[{dec,ra},. . .] and a gradient grad[{ddec,dphi/cos(dec)},. . .] (as returned by curvedsky.alm2map(deriv=True)), returns opos = ipos + grad, while properly parallel transporting on the sphere. If geodesic=False is specified, then an much faster approximation is used, which is still very accurate unless one is close to the poles.

`pixell.lensing.`**`offset_by_grad_helper`**(*ipos*, *grad*, *pol*)

Find the new position and induced rotation from offseting the input positions ipos[2,nsamp] by grad[2,nsamp].

`pixell.lensing.`**`pole_wrap`**(*pos*)

Handle pole wraparound.

3.8 2.8 pointsrcs - Point Sources

Point source parameter I/O. In order to simulate a point source as it appears on the sky, we need to know its position, amplitude and local beam shape (which can also absorb an extendes size for the source, as long as it's gaussian). While other properties may be nice to know, those are the only ones that matter for simulating it. This module provides functions for reading these minimal parameters from various data files.

The standard parameters are [nsrc,nparam]:
dec (radians) ra (radians) [T,Q,U] amplitude at center of gaussian (uK) beam sigma (wide axis) (radians) beam sigma (short axis) (radians) beam orientation (wide axis from dec axis) (radians)

What do I really need to simulate a source?

1. Physical source on the sky (pos,amps,shape)
2. Telescope response (beam in focalplane)

For a point source 1.shape would be a point. But clusters and nearby galaxies can have other shapes. In general many profiles are possible. Parametrizing them in a standard format may be difficult.

pixell.pointsrcs.**sim_objects**(*shape*, *wcs*, *poss*, *amps*, *profile*, *prof_ids=None*, *omap=None*, *vmin=None*, *rmax=None*, *op='add'*, *pixwin=False*, *pixwin_order=0*, *separable='auto'*, *transpose=False*, *prof_equi='auto'*, *cache=None*, *return_times=False*)

Simulate radially symmetric objects with arbitrary profiles and amplitudes. Arguments: * shape, wcs: The geometry of the patch to simulate. Only shape[-2:]

is used. amps determines the pre-dimensions

- poss: The positions of the objects. [{dec,ra},nobj] in radians.
- amps: The central amplitudes of the objects. […,nobj]. Not the same as the flux.
- profile: The profiles to use. Either [{r,b(r)},nsamp] or a list of such.

Optional arguments: * prof_ids: Which profile to use for each source. Defaults to use

the first profile for all. Only necessary to specify if you want to simulate objects with varying profiles. If specified, it should be [nobj] indices into the profile list.

- omap: Update this map instead of constructing a new one. MUST BE float32 AND C CONTIGUOUS and have shape […,ny,nx] where … matches amps.
- vmin: The lowest values to bother simulating. To avoid being terribly slow, profiles aren't evaluated out to infinite distance, but only once they drop down to a sufficiently low level, given by vmin. This takes into account the peak amplitud of each object, so it should be in map units. For example, it might be reasonable to have a vmin a few times lower than the noise level of the map, e.g. 0.1 µK. If not specified, then it defaults to min(abs(amps))*1e-3.
- rmax: The maximum radius to use, in radians. Acts as a cap on the radius calculated from vmin. Not applied if None or 0.
- op: What operation to use when combining the input map with each object. "add": Add linearly [default] "max": Keep the max value in each pixel "min": Keep the min value in each pixel
- pixwin: Whether to apply a pixel window after simulating the objects. This assumes periodic boundary consitions, so objects at the very edge will be wrong.
- separable: Whether the coordinate system's coordinate axes are indpendent, such that one only needs to know y in order to calculate dec, and x to calculate ra. This allows for much faster calculation of the pixel coordinates. Default "auto": True for cylindrical coordinates, False otherwise.
- cache: Dictionary to use for caching pixel coordinates. Can be useful if you're doing repeated simulations on the same geometry with non-separable geometry, to avoid having to recalculate the pixel coordinates all the time.

Returns the resulting map. If omap was specified, then the same object will be returned (after being updated of course). In this case, the simulated sources will have been added (or maxed etc. depending on op) into the map. Otherwise, the only signal in the map will be the objects.

pixell.pointsrcs.**is_equi**(*r*)

Estimate whether the values r[:] = arange(n)*delta, allowing for fast index calculations. This is just a heuristic, but it is hopefully reliable enough.

pixell.pointsrcs.**radial_sum**(*map*, *poss*, *bins*, *oprofs=None*, *separable='auto'*, *prof_equi='auto'*, *cache=None*, *return_times=False*)

Sum the signal in map into radial bins around a set of objects, returning one radial sum-profile per object. Arguments: * map: The map to read data from. […,ny,nx] * poss: The positions of the objects. [{dec,ra},nobj] in radians. * bins: The bin edges. [nbin+1]. Faster if equi-spaced with first at 0

Optional arguments: * oprofs: [obj,…,nbin] array to write result to. MUST BE float32 AND C CONTIGUOUS * separable: Whether the coordinate system's coordinate axes are indpendent,

> such that one only needs to know y in order to calculate dec, and x to calculate ra. This allows for much faster calculation of the pixel coordinates. Default "auto": True for cylindrical coordinates, False otherwise.

- cache: Dictionary to use for caching pixel coordinates. Can be useful if you're doing repeated simulations on the same geometry with non-separable geometry, to avoid having to recalculate the pixel coordinates all the time.

Returns the resulting profiles. If oprof was specified, then the same object will be returned (after being updated of course).

`pixell.pointsrcs.`**`radial_bin`**(*map*, *poss*, *bins*, *weights=None*, *separable='auto'*, *prof_equi='auto'*, *cache=None*, *return_times=False*)

Average the signal in map into radial bins for a set of objects, returning a radial profile for each object. Arguments: * map: The map to read data from. […,ny,nx] * poss: The positions of the objects. [{dec,ra},nobj] in radians. * bins: The bin edges. [nbin+1]. Faster if equi-spaced with first at 0

Optional arguments: * oprofs: [obj,…,nbin] array to write result to. MUST BE float32 AND C CONTIGUOUS * separable: Whether the coordinate system's coordinate axes are indpendent,

> such that one only needs to know y in order to calculate dec, and x to calculate ra. This allows for much faster calculation of the pixel coordinates. Default "auto": True for cylindrical coordinates, False otherwise.

- cache: Dictionary to use for caching pixel coordinates. Can be useful if you're doing repeated simulations on the same geometry with non-separable geometry, to avoid having to recalculate the pixel coordinates all the time.

Returns the resulting profiles. If oprof was specified, then the same object will be returned (after being updated of course).

`pixell.pointsrcs.`**`sim_srcs`**(*shape*, *wcs*, *srcs*, *beam*, *omap=None*, *dtype=None*, *nsigma=5*, *rmax=None*, *vmin=None*, *smul=1*, *return_padded=False*, *pixwin=False*, *pixwin_order=0*, *op=<ufunc 'add'>*, *wrap='auto'*, *verbose=False*, *cache=None*, *separable='auto'*, *method='c'*)

Backwards compatibility wrapper that exposes the speed of the new sim_objects function using the old sim_srcs interface. For most users this should result in a transparent speedup of O(10x), but sim_objects does not implement 100% of the sim_srcs functionality, so the old python method is also available by specifying method = "python".

Limitations of the new version: * only float32, C-contiguous maps supported * smul not supported * padding not supported, which impacts objects at the very edge of the map if

> pixwin is used

- only add, max and min supported for 'op'

Unlike sim_srcs, sim_objects supports simulating objects with multiple different profiles at once, but this functionality isn't available through the sim_srcs interface.

I recommend using sim_objects directly instead of relying on this wrapper in most cases.

`pixell.pointsrcs.`**`sim_srcs_python`**(*shape*, *wcs*, *srcs*, *beam*, *omap=None*, *dtype=None*, *nsigma=5*, *rmax=None*, *smul=1*, *return_padded=False*, *pixwin=False*, *pixwin_order=0*, *op=<ufunc 'add'>*, *wrap='auto'*, *verbose=False*, *cache=None*, *separable='auto'*)

Simulate a point source map in the geometry given by shape, wcs for the given srcs[nsrc,{dec,ra,T…}], using the beam[{r,val},npoint], which must be equispaced. If omap is specified, the sources will be added to it in place. All angles are in radians. The beam is only evaluated up to the point where it reaches exp(-0.5*nsigma**2) unless rmax is specified, in which case this gives the maximum radius. smul gives a factor to multiply the resulting source model by. This is mostly useful in conction with omap.

The source simulation is sped up by using a source lookup grid.

`pixell.pointsrcs.`**`eval_srcs_loop`**(*posmap*, *poss*, *amps*, *beam*, *cres*, *nhit*, *cell_srcs*, *dtype=<class 'numpy.float64'>*, *op=<ufunc 'add'>*, *verbose=False*)

`pixell.pointsrcs.`**`sim_srcs_dist_transform`**(*shape*, *wcs*, *srcs*, *beam*, *omap=None*, *dtype=None*, *nsigma=4*, *rmax=None*, *smul=1*, *pixwin=False*, *ignore_outside=False*, *op=<ufunc 'add'>*, *verbose=False*)

Simulate a point source map in the geometry given by shape, wcs for the given srcs[nsrc,{dec,ra,T…}], using the beam[{r,val},npoint], which must be equispaced. Unlike sim_srcs, overalpping point sources are not supported. If omap is specified, the sources will be added to it in place. All angles are in radians. The beam is only evaluated up to the point where it reaches exp(-0.5*nsigma**2) unless rmax is specified, in which case this gives the maximum radius. smul gives a factor to multiply the resulting source model by. This is mostly useful in conction with omap.

`pixell.pointsrcs.`**`expand_beam`**(*beam*, *nsigma=5*, *rmax=None*, *nper=400*)

`pixell.pointsrcs.`**`nsigma2rmax`**(*beam*, *nsigma*)

`pixell.pointsrcs.`**`build_src_cells`**(*cbox*, *srcpos*, *cres*, *unwind=False*, *wrap=None*)

`pixell.pointsrcs.`**`build_src_cells_helper`**(*cbox*, *cshape*, *cres*, *srcpos*, *nmax=0*, *wrap=None*)

`pixell.pointsrcs.`**`cellify`**(*map*, *res*)

Given a map […,ny,nx] and a cell resolution [ry,rx], return map reshaped into a cell grid […,ncelly,ncellx,ry,rx]. The map will be truncated if necessary

`pixell.pointsrcs.`**`uncellify`**(*cmap*)

`pixell.pointsrcs.`**`crossmatch`**(*srcs1*, *srcs2*, *tol=0.0002908882086657216*, *safety=4*)

Cross-match two source catalogs based on position. Each source in one catalog is associated with the closest source in the other catalog, as long as the distance between them is less than the tolerance. The catalogs must be [:,{ra,dec,…}] in radians. Returns [nmatch,2], with the last index giving the index in the first and second catalog for each match.

`pixell.pointsrcs.`**`read`**(*fname*, *format='auto'*)

`pixell.pointsrcs.`**`read_nemo`**(*fname*)

Reads the nemo ascii catalog format, and returns it as a recarray.

`pixell.pointsrcs.`**`read_simple`**(*fname*)

`pixell.pointsrcs.`**`read_dory_fits`**(*fname*, *hdu=1*)

`pixell.pointsrcs.`**`read_dory_txt`**(*fname*)

`pixell.pointsrcs.`**`read_fits`**(*fname*, *hdu=1*, *fix=True*) `pixell.pointsrcs.`**`format_sauron`**`(`*cat*`)`

`pixell.pointsrcs.`**`write_sauron`**`(`*ofile*, *cat*`)`

`pixell.pointsrcs.`**`read_sauron`**`(`*ifile*`)`

`pixell.pointsrcs.`**`write_sauron_fits`**`(`*ofile*, *cat*`)`

`pixell.pointsrcs.`**`read_sauron_fits`**`(`*fname*`)`

`pixell.pointsrcs.`**`write_sauron_txt`**`(`*ofile*, *cat*`)`

`pixell.pointsrcs.`**`read_sauron_txt`**`(`*ifile*, *ncomp=3*`)`

`pixell.pointsrcs.`**`translate_dtype_keys`**`(`*d*, *translation*`)`

`pixell.pointsrcs.`**`src2param`**`(`*srcs*`)`

Translate recarray srcs into the source fromat used for tod-level point source operations.

3.9 2.9 interpol - Interpolation

`pixell.interpol.`**`map_coordinates`**`(`*idata*, *points*, *odata=None*, *mode='spline'*, *order=3*, *border='cyclic'*, *trans=False*, *deriv=False*, *prefilter=True*`)`

An alternative implementation of scipy.ndimage.map_coordinates. It is slightly slower (20-30%), but more general. Basic usage is

odata[{pre},{pdims}] = map_coordinates(idata[{pre},{dims}], points[ndim,{pdims}])

where {foo} means a (possibly empty) shape. For example, if idata has shape (10,20) and points has shape (2,100), then the result will have shape (100,), and if idata has shape (10,20,30,40) and points has shape (3,1,2,3,4), then the result will have shape (10,1,2,3,4). Except for the presence of {pre}, this is the same as how map_coordinates works.

It is also possible to pass the output array as an argument (odata), which must have the same data type as idata in that case.

The function differs from ndimage in the meaning of the optional arguments. mode specifies the interpolation scheme to use: “conv”, “spline” or “lanczos”. “conv” is polynomial convolution, which is commonly used in image processing. “spline” is spline interpolation, which is what ndimage uses. “lanczos” convolutes with a lanczos kernerl, which approximates the optimal sinc kernel. This is slow, and the quality is not much better than spline.

order specifies the interpolation order, its exact meaning differs based on mode.

border specifies the handling of boundary conditions. It can be “zero”, “nearest”, “cyclic” or “mirror”/”reflect”. The latter corresponds to ndimage’s “reflect”. The others do not match ndimage due to ndimage’s inconsistent treatment of boundary conditions in spline_filter vs. map_coordiantes.

trans specifies whether to perform the transpose operation or not. The interpolation performed by map_coordinates is a linear operation, and can hence be expressed as out = A*data, where A is a matrix. If trans is true, then what will instead be performed is data = A.T*in. For this to work, the odata argument must be specified. This will be read from, while idata will be written to.

Normally idata is read and odata is written to, but when trans=True, idata is written to and odata is read from.

If deriv is True, then the function will compute the derivative of the interpolation operation with respect to the position, resulting in odata[ndim,{pre},{pdims}] `pixell.interpol.spline_filter`(*data*, *order=3*, *border='cyclic'*, *ndim=None*, *trans=False*)

Apply a spline filter to the given array. This is normally done on-the-fly internally in map_coordinates when using spline interpolation of order > 1, but since it's an operation that applies to the whole input array, it can be a big overhead to do this for every call if only a small number of points are to be interpolated. This overhead can be avoided by manually filtering the array once, and then passing in the filtered array to map_coordinates with prefilter=False to turn off the internal filtering.

`pixell.interpol.get_core`(*dtype*)

`pixell.interpol.build`(*func*, *interpolator*, *box*, *errlim*, *maxsize=None*, *maxtime=None*, *return_obox=False*, *return_status=False*, *verbose=False*, *nstart=None*, **args*, ***kwargs*)

Given a function func([nin,. . .]) => [nout,. . .] and an interpolator class interpolator(box,[nout,. . .]), (where the input array is regularly spaced in each direction), which provides __call__([nin,. . .]) => [nout,. . .], automatically polls func and constructs an interpolator object that has the required accuracy inside the provided bounding box.

`class pixell.interpol.Interpolator`(*box*, *y*, **args*, ***kwargs*)

`class pixell.interpol.ip_ndimage`(*box*, *y*, **args*, ***kwargs*)

`class pixell.interpol.ip_linear`(*box*, *y*, **args*, ***kwargs*)

`class pixell.interpol.ip_grad`(*box*, *y*, **args*, ***kwargs*)

Gradient interpolation. Faster but less accurate than bilinear

`pixell.interpol.lin_derivs_forward`(*y*, *npre=0*)

Given an array y with npre leading dimensions and n following dimensions, compute all combinations of the 0th and 1st derivatives along the n last dimensions, returning an array of shape (2,)*n+(:,)*npre+(:-1,)*n. That is, it is one shorter in each direction along which the derivative is taken. Derivatives are computed using forward difference.

`pixell.interpol.grad_forward`(*y*, *npre=0*)

Given an array y with npre leading dimensions and n following dimensions, the gradient along the n last dimensions, returning an array of shape (n,)+y.shape. Derivatives are computed using forward difference.

3.10 2.10 coordinates - Coordinate Transformation

`class pixell.coordinates.default_site`

`lat = -22.9585`

`lon = -67.7876`

`alt = 5188.0`

`T = 273.15`

`P = 550.0`

`hum = 0.2`

`freq = 150.0`

`lapse = 0.0065`

`base_tilt = 0.0107693`

`base_az = -114.9733961`

`pixell.coordinates.transform`(*from_sys*, *to_sys*, *coords*, *time=55500*, *site=<class 'pixell.coordinates.default_site'>*, *pol=None*, *mag=None*, *bore=None*)

Transforms coords[2,. . .] from system from_sys to system to_sys, where systems can be "hor", "cel" or "gal". For transformations involving "hor", the optional arguments time (in modified julian days) and site (which must contain .lat (rad), .lon (rad), .P (pressure, mBar), .T (temperature, K), .hum (humidity, 0.2 by default), .alt (altitude, m)). Returns an array with the same shape as the input. The coordinates are in ra,dec-ordering.

`pixell.coordinates.transform_meta`(*transfun*, *coords*, *fields=['ang', 'mag']*, *offset=5e-07*)

Computes metadata for the coordinate transformation functor transfun applied to the coordinate array coords[2,. . .], such as the induced rotation, magnification.

Currently assumes that input and output coordinates are in non-zenith polar coordinates. Might generalize this later.

`pixell.coordinates.transform_raw`(*from_sys*, *to_sys*, *coords*, *time=None*, *site=<class 'pixell.coordinates.default_site'>*, *bore=None*)

Transforms coords[2,. . .] from system from_sys to system to_sys, where systems can be "hor", "cel" or "gal". For transformations involving "hor", the optional arguments time (in modified julian days) and site (which must contain .lat (rad), .lon (rad), .P (pressure, mBar), .T (temperature, K), .hum (humidity, 0.2 by default), .alt (altitude, m)). Returns an array with the same shape as the input. The coordinates are in ra,dec-ordering.

coords and time will be broadcast such that the result has the same shape as coords*time[None].

`pixell.coordinates.transform_astropy`(*from_sys*, *to_sys*, *coords*)

As transform, but only handles the systems supported by astropy.

`pixell.coordinates.transform_euler`(*euler*, *coords*, *pol=None*, *mag=None*)

Like transform, but for a set of zyz euler angles instead

`pixell.coordinates.hor2cel`(*coord*, *time*, *site*, *copy=True*)

`pixell.coordinates.cel2hor`(*coord*, *time*, *site*, *copy=True*)

`pixell.coordinates.tele2hor`(*coord*, *site*, *copy=True*)

`pixell.coordinates.hor2tele`(*coord*, *site*, *copy=True*)

`pixell.coordinates.tele2bore`(*coord*, *bore*, *copy=True*)

Transforms coordinates [{ra,dec},. . .] to boresight-relative coordinates given by the boresight pointing [{ra,dec},. . .] with the same shape as coords. After the rotation, the boresight will be at the zenith; things above the boresight will be at 'ra'=180 and things below will be 'ra'=0.

`pixell.coordinates.bore2tele`(*coord*, *bore*, *copy=True*)

Transforms coordinates [{ra,dec},. . .] from boresight-relative coordinates given by the boresight pointing [{ra,dec},. . .] with the same shape as coords. After the rotation, the coordinates will be in telescope coordinates, which are similar to horizontal coordinates.

`pixell.coordinates.euler_mat`(*euler_angles*, *kind='zyz'*)

Defines the rotation matrix M for a ABC euler rotation, such that M = A(alpha)B(beta)C(gamma), where euler_angles = [alpha,beta,gamma]. The default kind is ABC=ZYZ.

`pixell.coordinates.euler_rot`(*euler_angles*, *coords*, *kind='zyz'*)

`pixell.coordinates.recenter`(*angs*, *center*, *restore=False*)

Recenter coordinates "angs" (as ra,dec) on the location given by "center", such that center moves to the north pole.

`pixell.coordinates.`**`decenter`**`(`*angs*, *center*, *restore=False*`)`

Inverse operation of recenter.

`pixell.coordinates.`**`nohor`**`(`*sys*`)`

`pixell.coordinates.`**`getsys`**`(`*sys*`)`

`pixell.coordinates.`**`get_handedness`**`(`*sys*`)`

Return the handedness of the coordinate system sys, as seen from inside the celestial sphere, in the standard IAU convention.

`pixell.coordinates.`**`getsys_full`**`(`*sys*, *time=None*, *site=<class 'pixell.coordinates.default_site'>*, *bore=None*`)`

Handles our expanded coordinate system syntax: base[:ref[:refsys]]. This allows a system to be recentered on a given position or object. The argument can either be a string of the above format (with [] indicating optional parts), or a list of [base, ref, refsys]. Returns a parsed and expanded version, where the systems have been replaced by full system objects (or None), and the reference point has been expanded into coordinates (or None), and rotated into the base system. Coordinates are separated by _.

Example: Horizontal-based coordinates with the Moon centered at [0,0] would be hor:Moon/0_0.

Example: Put celestial coordinates ra=10, dec=20 at horizontal coordinates az=0, el=0: hor:10_20:cel/0_0:hor. Yes, this is horrible.

Used to be sys:center_on/center_at:sys_of_center_coordinates. But much more flexible to do sys:center_on:sys/center_at:sys. This syntax would be backwards compatible, though it's starting to get a bit clunky.

Big hack: If the system is "sidelobe", then we will use sidelobe-oriented centering instead of object-oriented centering. This will result in a coordinate system where the boresight has the zenith-mirrored position of what the object would have in zenith-relative coordinates.

`pixell.coordinates.`**`ephem_pos`**`(`*name*, *mjd*`)`

Given the name of an ephemeris object from pyephem and a time in modified julian date, return its position in ra, dec in radians in equatorial coordinates.

`pixell.coordinates.`**`interpol_pos`**`(`*from_sys*, *to_sys*, *name_or_pos*, *mjd*, *site=<class 'pixell.coordinates.default_site'>*, *dt=10*`)`

Given the name of an ephemeris object or a [ra,dec]-type position in radians in from_sys, compute its position in the specified coordinate system for each mjd. The mjds are assumed to be sampled densely enough that interpolation will work. For ephemeris objects, positions are computed in steps of 10 seconds by default (controlled by the dt argument).

`pixell.coordinates.`**`make_mapping`**`(`*dict*`)`

3.11 2.11 wcsutils - World Coordinate Sytem utilities

This module defines shortcuts for generating WCS instances and working with them. The bounding boxes and shapes used in this module all use the same ordering as WCS, i.e. column major (so {ra,dec} rather than {dec,ra}). Coordinates are assigned to pixel centers, as WCS does natively, but bounding boxes include the whole pixels, not just their centers, which is where the 0.5 stuff comes from.

`pixell.wcsutils.`**`streq`**`(`*x*, *s*`)`

`pixell.wcsutils.`**`explicit`**`(`*naxis=2*, ***args*`)` pixell.wcsutils.**describe**(*wcs*)

Since astropy.wcs.WCS objects do not have a useful str implementation, this function provides a relpacement.

pixell.wcsutils.**equal**(*wcs1*, *wcs2*, *flags=1*, *tol=1e-14*)

pixell.wcsutils.**nobcheck**(*wcs*)

pixell.wcsutils.**is_compatible**(*wcs1*, *wcs2*, *tol=0.001*)

Checks whether two world coordinate systems represent (shifted) versions of the same pixelizations, such that every pixel center in wcs1 correspond to a pixel center in wcs2. For now, they also have to have the pixels going in the same direction.

pixell.wcsutils.**is_plain**(*wcs*)

Determines whether the given wcs represents plain, non-specific, non-wrapping coordinates or some angular coordiante system.

pixell.wcsutils.**is_cyl**(*wcs*)

Returns True if the wcs represents a cylindrical coordinate system

pixell.wcsutils.**get_proj**(*wcs*)

pixell.wcsutils.**scale**(*wcs*, *scale=1*, *rowmajor=False*, *corner=False*)

Scales the linear pixel density of a wcs by the given factor, which can be specified per axis. This is the same as dividing the pixel size by the same number.

pixell.wcsutils.**expand_res**(*res*, *default_dirs=[1, -1]*)

pixell.wcsutils.**plain**(*pos*, *res=None*, *shape=None*, *rowmajor=False*, *ref=None*)

Set up a plain coordinate system (non-cyclical)

pixell.wcsutils.**car**(*pos*, *res=None*, *shape=None*, *rowmajor=False*, *ref=None*)

Set up a plate carree system. See the build function for details.

pixell.wcsutils.**cea**(*pos*, *res=None*, *shape=None*, *rowmajor=False*, *lam=None*, *ref=None*)

Set up a cylindrical equal area system. See the build function for details.

pixell.wcsutils.**mer**(*pos*, *res=None*, *shape=None*, *rowmajor=False*, *ref=None*)

Set up a mercator system. See the build function for details.

pixell.wcsutils.**zea**(*pos*, *res=None*, *shape=None*, *rowmajor=False*, *ref=None*)

Setups up an oblate Lambert's azimuthal equal area system. See the build function for details. Don't use this if you want a polar projection.

pixell.wcsutils.**air**(*pos*, *res=None*, *shape=None*, *rowmajor=False*, *rad=None*, *ref=None*)

Setups up an Airy system. See the build function for details.

pixell.wcsutils.**tan**(*pos*, *res=None*, *shape=None*, *rowmajor=False*, *ref=None*)

Set up a gnomonic (tangent plane) system. See the build function for details.

pixell.wcsutils.**build**(*pos*, *res=None*, *shape=None*, *rowmajor=False*, *system='cea'*, *ref=None*, ***kwargs*)

Set up the WCS system named by the "system" argument. pos can be either a [2] center position or a [{from,to},2] bounding box. At least one of res or shape must be specified. If res is specified, it must either be a number, in which the same resolution is used in each direction, or [2]. If shape is specified, it must be [2]. All angles are given in degrees.

pixell.wcsutils.**validate**(*pos*, *res*, *shape*, *rowmajor=False*, *default_dirs=[1, -1]*)

`pixell.wcsutils.`**`finalize`**`(`*w*, *pos*, *res*, *shape*, *ref=None*`)`

Common logic for the various wcs builders. Fills in the reference pixel and resolution.

`pixell.wcsutils.`**`angdist`**`(`*lon1*, *lat1*, *lon2*, *lat2*`)`

`pixell.wcsutils.`**`fix_wcs`**`(`*wcs*, *axis=0*`)`

Returns a new WCS object which has had the reference pixel moved to the middle of the possible pixel space.

`pixell.wcsutils.`**`fix_cdelt`**`(`*wcs*`)`

Return a new wcs with pc and cd replaced by cdelt

3.12 2.12 powspec - CMB power spectra utilities

`pixell.powspec.`**`sym_compress`**`(`*mat*, *which=None*, *n=None*, *scheme=None*, *axes=[0, 1]*, *combined=False*`)`

Extract the unique elements of a symmetric matrix, and return them as a flat array. For multidimensional arrays, the extra dimensions keep their shape. The optional argument 'which' indicates the compression scheme, as returned by compressed_order. The optional argument 'n' indicates the number of elements to keep (the default is to keep all unique elements). The 'axes' argument indicates which axes to operate on.

`pixell.powspec.`**`sym_expand`**`(`*mat*, *which=None*, *ncomp=None*, *scheme=None*, *axis=0*, *combined=False*`)`

The inverse of sym_compress. Expands a flat array of numbers into a symmetric matrix with ncomp components using the given mapping which (or construct one using the given scheme).

`pixell.powspec.`**`sym_expand_camb_full_lens`**`(`*a*`)`

`pixell.powspec.`**`compressed_order`**`(`*n*, *scheme=None*`)`

Surmise the order in which the unique elements of a symmetric matrix are stored, based on the number of such elements. Three different schemes are supported. The best one is the "stable" scheme because it can be truncated without the entries changing their meaning. However, the default in healpy is "diag", so that is the default here too.

stable:
00 00 11 00 11 01 00 11 01 22 00 11 01 22 02 00 11 01 22 02 12 …

diag:
00 00 11 00 11 01 00 11 22 01 00 11 22 01 12 00 11 22 01 12 02 …

row:
00 00 11 00 01 11 00 01 11 22 00 01 02 11 22 00 01 02 11 12 22 …

`pixell.powspec.`**`expand_inds`**`(`*x*, *y*`)`

`pixell.powspec.`**`scale_spectrum`**`(`*a*, *direction*, *extra=0*, *l=None*`)`

`pixell.powspec.`**`scale_camb_scalar_phi`**`(`*a*, *direction*, *l=None*`)`

`pixell.powspec.`**`read_spectrum`**`(`*fname*, *inds=True*, *scale=True*, *expand='diag'*, *ncol=None*, *ncomp=None*`)`

Read a power spectrum from disk and return a dense array cl[nspec,lmax+1]. Unless scale=False, the spectrum will be multiplied by 2pi/l/(l+1) when being read. Unless inds=False, the first column in the file is assumed to be the indices. If expand!=None, it can be one of the valid expansion schemes from compressed_order, and will cause the returned array to be cl[ncomp,ncomp,lmax+1] instead.

`pixell.powspec.`**`read_phi_spectrum`**`(`*fname*, *coloff=0*, *inds=True*, *scale=True*, *expand='diag'*`)` pixell.powspec.**read_camb_scalar**(*fname*, *inds=True*, *scale=True*, *expand=True*, *ncmb=3*)

Read the information in the camb scalar outputs. This contains the cmb and lensing power spectra, but not their correlation. They are therefore returned as two separate arrays.

pixell.powspec.**read_camb_full_lens**(*fname*, *inds=True*, *scale=True*, *expand=True*, *ncmb=3*)

Reads the CAMB lens_potential_output spectra, which contain l TT EE BB TE dd dT dE. These are rescaled appropriately is scale is True, and returned as [d,T,E,B] if expand is True.

pixell.powspec.**write_spectrum**(*fname*, *spec*, *inds=True*, *scale=True*, *expand='diag'*)

pixell.powspec.**spec2corr**(*spec*, *pos*, *iscos=False*, *symmetric=True*)

Compute the correlation function sum(2l+1)/4pi Cl Pl(cos(theta)) corresponding to the given power spectrum at the given positions.

3.13 2.13 enplot - Producing plots from ndmaps

class pixell.enplot.**Printer**(*level=1*, *prefix=''*)

write(*desc*, *level*, *exact=False*, *newline=True*, *prepend=''*)

push(*desc*)

time(*desc*, *level*, *exact=False*, *newline=True*)

pixell.enplot.**plot**(**arglist*, ***args*)

The main plotting function in this module. Plots the given maps/files, returning them as a list of plots, one for each separate image. This function has two equivalent interfaces: 1. A command-line like interface, where plotting options are specified with strings like "-r 500:50 -m 0 -t 2". 2. A python-like interface, where plotting options are specified with keyword arguments, like range="500:50", mask=0, ticks=2. These interfaces can be mixed and matched.

Input maps are specified either as part of the option strings, as separate file names, or as enmaps passed to the function. Here are some examples:

plot(mapobj):
Plots the given enmap object mapobj. If mapobj is a scalar map, the a length-1 list containing a single plot will be returned. If mapobj is e.g. a 3-component TQU map, then a length-3 list of plots will be returned.

plot((mapobj,"foo")):

If a tuple is given, the second element specifies the name tag to use. This tag will be used to populate the plot.name attribute for each output plot, which can be useful when plotting and writing the maps.

plot("file.fits"):
Reads a map from file.fits, and plots it. This sets the tag to "file", so that the result can easily be written to "file.png" (or "file_0.png" etc).

plot(["a.fits","b.fits",(mapobj,"c"),(mapobj,"d.fits")])
Reads a.fits and plots it to a.png, reads b.fits and plots it to b.png, plots the first mapobj to c.png and the second one to d.png (yes, the extension in the filename specified in the tuple is ignored. This is because that filename actually supplies the *input* filename that the output filename should be computed from).

plot("foo*.fits")
Reads and plots every file matching the glob foo*.fits.

plot("a.fits -r 500:50 -m 0 -d 4 -t 4")
Reads and plots the file a.fits to a.png, using a color range of +-500 for the first field in the map (typically the total intensity), and +-50 for the remaining fields (typically Q and U). The maps are downsampled by a factor of 4, and plotted with a grid spacing of 4.

Here is a list of the individual options plot recognizes. The short and long ones are recognized when used in the argument string syntax, while the long ones (with - replaced by _) also work as keyword arguments.

See plot_iterator for an iterator version of this function.

-h, --help show this help message and exit

-o ONAME, --oname ONAME The format to use for the output name. Default is {dir}{pre}{base}{suf}{comp}{layer}.{ext}

-c COLOR, --color COLOR The color scheme to use, e.g. planck, wmap, gray, hotcold, etc., or a colors pecification in the form val:rrggbb,val:rrggbb,. . . . Se enlib.colorize for details.

-r RANGE, --range RANGE The symmetric color bar range to use. If specified, colors in the map will be truncated to [-range,range]. To give each component in a multidimensional map different color ranges, use a colon-separated list, for example -r 250:100:50 would plot the first component with a range of 250, the second with a range of 100 and the third and any subsequent component with a range of 50.

--min MIN The value at which the color bar starts. See –range.

--max MAX The value at which the color bar ends. See –range.

-q QUANTILE, --quantile QUANTILE Which quantile to use when automatically determining the color range. If specified, the color bar will go from [quant(q),quant(1-q)].

-v Verbose output. Specify multiple times to increase verbosity further.

-u UPGRADE, -s UPGRADE, --upgrade UPGRADE, --scale UPGRADE Upscale the image using nearest neighbor interpolation by this amount before plotting. For example, 2 would make the map twice as large in each direction, while 4,1 would make it 4 times as tall and leave the width unchanged.

--verbosity VERBOSITY Specify verbosity directly as an integer.

--method METHOD Which colorization implementation to use: auto, fortran or python.

--slice SLICE Apply this numpy slice to the map before plotting.

--sub SUB Slice a map based on dec1:dec2,ra1:ra2.

-H HDU, --hdu HDU Header unit of the fits file to use

--op OP Apply this general operation to the map before plotting. For example, 'log(abs(m))' would give you a lograithmic plot.

-d DOWNGRADE, --downgrade DOWNGRADE Downsacale the map by this factor before plotting. This is done by averaging nearby pixels. See –upgrade for syntax.

--prefix PREFIX Specify a prefix for the output file. See –oname.

--suffix SUFFIX Specify a suffix for the output file. See –oname.

--odir ODIR Override the output directory. See –oname.

--ext EXT Specify an extension for the output file. This will determine the file type of the resulting image. Can be anything PIL recognizes. The default is png.

-m MASK, --mask MASK Mask this value, making it transparent in the output image. For example -m 0 would mark all values exactly equal to zero as missing.

--mask-tol MASK_TOL The tolerance to use with –mask.

-g, --grid Toggle the coordinate grid. Disabling it can make plotting much faster when plotting many small maps.

--grid-color GRID_COLOR The RGBA color to use for the grid.

--grid-width GRID_WIDTH The line width to use for the grid.

-t TICKS, --ticks TICKS The grid spacing in degrees. Either a single number to be used for both axis, or ty,tx.

--tick-unit TICK_UNIT, --tu TICK_UNIT Units for tick axis. Can be the unit size in degrees, or the word 'degree', 'arcmin' or 'arcsec' or the shorter 'd','m','s'.

--nolabels Disable the generation of coordinate labels outside the map when using the grid.

--nstep NSTEP The number of steps to use when drawing grid lines. Higher numbers result in smoother curves.

--subticks SUBTICKS Subtick spacing. Only supported by matplotlib driver.

-b, --colorbar Whether to draw the color bar or not

--font FONT The font to use for text.

--font-size FONT_SIZE Font size to use for text.

--font-color FONT_COLOR Font color to use for text.

-D DRIVER, --driver DRIVER The driver to use for plotting. Can be pil (the default) or mpl. pil cleanly maps input pixels to output pixels, and has better coordiante system support, but doesn't have as pretty grid lines or axis labels.

--mpl-dpi MPL_DPI The resolution to use for the mpl driver.

--mpl-pad MPL_PAD The padding to use for the mpl driver.

--rgb Enable RGB mode. The input maps must have 3 components, which will be interpreted as red, green and blue channels of a single image instead of 3 separate images as would be the case without this option. The color scheme is overriden in this case.

--rgb-mode RGB_MODE The rgb mode to use. Can be direct or direct_colorcap. These only differ in whether colors are preserved when too high or low colors are capped. direct_colorcap preserves colors, at the cost of noise from one noisy component leaking into others during capping.

--reverse-color Reverse the color scale. For example, a black-to-white scale will become a white-to-black sacle.

-a, --autocrop Automatically crop the image by removing expanses of uniform color around the edges. This is done jointly for all components in a map, making them directly comparable, but is done independently for each input file.

-A, --autocrop-each As –autocrop, but done individually for each component in each map.

-L, --layers Output the individual layers that make up the final plot (such as the map itself, the coordinate grid, the axis labels, any contours and lables) as individual files instead of compositing them into a final image.

--no-image Skip the main image plotting. Useful for getting a pure contour plot, for example.

-C CONTOURS, --contours CONTOURS Enable contour lines. For example -C 10 to place a contour at every 10 units in the map, -C 5:10 to place it at every 10 units, but starting at 5, and 1,2,4,8 or similar to place contours at manually chosen locations.

--contour-type CONTOUR_TYPE The type of the contour specification. Only used when the contours specification is a list of numbers rather than a string (so not from the command line interface). 'uniform': the list is [interval] or [base, interval]. 'list': the list is an explicit list of the values the contours should be at.

--contour-color CONTOUR_COLOR The color scheme to use for contour lines. Either a single rrggbb, a val:rrggbb,val:rrggbb,… specification or a color scheme name, such as planck, wmap or gray.

--contour-width CONTOUR_WIDTH The width of each contour line, in pixels.

--annotate ANNOTATE Annotate the map with text, lines or circles. Should be a text file with one entry per line, where an entry can be: c[ircle] lat lon dy dx [rad [width [color]]] t[ext] lat lon dy dx text [size [color]] l[ine] lat lon dy dx lat lon dy dx [width [color]] dy and dx are pixel-unit offsets from the specified lat/lon. Alternatively, from python one can pass in a list of lists containig the same information, e.g. [["circle", 5.10,222.3,0,0,32,3,"black"]]

--annotate-maxrad ANNOTATE_MAXRAD Assume that annotations do not extend further than this from their center, in pixels. This is used to prune which annotations to attempt to draw, as they can be a bit slow. The special value 0 disables this.

--stamps STAMPS Plot stamps instead of the whole map. Format is srcfile:size:nmax, where the last two are optional. srcfile is a file with [ra dec] in degrees, size is the size in pixels of each stamp, and nmax is the max number of stamps to produce.

--tile TILE Stack components vertically and horizontally. –tile 5,4 stacks into 5 rows and 4 columns. –tile 5 or –tile 5,-1 stacks into 5 rows and however many columns are needed. –tile -1,5 stacks into 5 columns and as many rows are needed. –tile -1 allocates both rows and columns to make the result as square as possible.

The result is treated as a single enmap, so the wcs will only be right for one of the tiles.

--tile-transpose Transpose the ordering of the fields when tacking. Normally row-major stacking is used. This sets column- major order instead.

–tile-dims TILE_DIMS -S, –symmetric Treat the non-pixel axes as being asymmetric matrix,

and only plot a non-redundant triangle of this matrix.

-z, --zenith Plot the zenith angle instead of the declination.

-F, --fix-wcs Fix the wcs for maps in cylindrical projections where the reference point was placed too far away from the map center.

`pixell.enplot.`**`pshow`**`(`**arglist*, *method='auto'*, ***args*`)`

Convenience function to both build plots and show them. pshow(. . .) is equivalent to show(plot(. . .)).

`pixell.enplot.`**`get_plots`**`(`**arglist*, ***args*`)`

This function is identical to enplot.plot

`pixell.enplot.`**`plot_iterator`**`(`**arglist*, ***kwargs*`)`

Iterator that yields the plots for each input map/file. Each yield will be a plot object, with fields .type: The type of the plot. Can be "pil" or "mpl". Usually "pil". .img: The plot image object, of the given .type. .name: Suggested file name These plots objects can be written to disk using enplot.write. See the plot function documentation for a description of the arguments

`pixell.enplot.`**`write`**`(`*fname*, *plot*`)`

Write the given plot or plots to file. If plot is a single plot, then it will simply be written to the specified filename. If plot is a list of plots, then it fname will be interpreted as a prefix, and the plots will be written to prefix + plot.name for each individual plot. If name was specified during plotting, then plot.name will either be ".png" for scalar maps or "_0.png", "_1.png", etc. for vector maps. It's also possible to pass in plain images (either PIL or matplotlib), which will be written to the given filename.

`pixell.enplot.`**`define_arg_parser`**`(`*nodefault=False*`)`

`pixell.enplot.`**`parse_args`**`(`*args=['-T', '-b', 'html', '-d', '_build/doctrees', '-D', 'language=en', '.', '/home/docs/checkouts/readthedocs.org/user_builds/pixell/checkouts/latest/_readthedocs//html']*, *noglob=False*, *nodef=False*`)`

`pixell.enplot.`**`extract_arg`**`(`*args*, *name*, *default*`)`

`pixell.enplot.`**`check_args`**`(`*kwargs*`)`

`pixell.enplot.`**`get_map`**`(`*ifile*, *args*, *return_info=False*, *name=None*`)`

Read the specified map, and massage it according to the options in args. Relevant ones are sub, autocrop, slice, op, downgrade, scale, mask. Retuns with shape [:,ny,nx], where any extra dimensions have been flattened into a single one.

`pixell.enplot.`**`extract_stamps`**`(`*map*, *args*`)`

Given a map, extract a set of identically sized postage stamps based on args.stamps. Returns a new map consisting of a stack of these stamps, along with a list of each of these' wcs object.

`pixell.enplot.`**`get_cache`**`(`*cache*, *key*, *fun*`)` `pixell.enplot.`**`draw_map_field`**`(`*map*, *args*, *crange=None*, *return_layers=False*, *return_info=False*, *printer=<pixell.enplot.Printer object>*, *cache=None*`)`

Draw a single map field, resulting in a single image. Adds a coordinate grid and lables as specified by args. If return_layers is True, an array will be returned instead of an image, wich each entry being a component of the image, such as the base image, the coordinate grid, the labels, etc. If return_bounds is True, then the

`pixell.enplot.`**`draw_colorbar`**`(`*crange*, *width*, *args*`)`

`pixell.enplot.`**`draw_map_field_mpl`**`(`*map*, *args*, *crange=None*, *printer=<pixell.enplot.Printer object>*`)`

Render a map field using matplotlib. Less tested and maintained than draw_map_field, and supports fewer features. Returns an object one can call savefig on to draw.

`pixell.enplot.`**`parse_range`**`(`*desc*, *n*`)`

`pixell.enplot.`**`parse_list`**`(`*desc*, *dtype=<class 'float'>*, *sep=', '*`)`

`pixell.enplot.`**`get_color_range`**`(`*map*, *args*`)`

Compute an appropriate color bare range from map[:,ny,nx] given the args. Relevant members are range, min, max, quantile.

`pixell.enplot.`**`get_num_digits`**`(`*n*`)`

`pixell.enplot.`**`split_file_name`**`(`*fname*`)`

Split a file name into directory, base name and extension, such that fname = dirname + "/" + basename + "." + ext.

`pixell.enplot.`**`map_to_color`**`(`*map*, *crange*, *args*`)`

Compute an [{R,G,B},ny,nx] color map based on a map[1 or 3, ny,nx] map and a corresponding color range crange[{min,max}]. Relevant args fields: color, method, rgb. If rgb is not true, only the first element of the input map will be used. Otherwise 3 will be used.

`pixell.enplot.`**`calc_gridinfo`**`(`*shape*, *wcs*, *args*`)`

Compute the points making up the grid lines for the given map. Depends on args.ticks and args.nstep.

`pixell.enplot.`**`draw_grid`**`(`*ginfo*, *args*`)`

Return a grid based on gridinfo. args.grid_color controls the color the grid will be drawn with.

`pixell.enplot.`**`draw_grid_labels`**`(`*ginfo*, *args*`)`

Return an image with a coordinate grid, along with abounds of this image relative to the coordinate shape stored in ginfo. Depends on the following args members: args.font, args.font_size, args.font_color

`pixell.enplot.`**`calc_contours`**`(`*crange*, *args*`)`

Returns a list of values at which to place contours based on the valure range of the map crange[{from,to}] and the contour specification in args.

Contour specifications:
base:step or val,val,val…

base: number step: number (explicit), -number (relative)

`pixell.enplot.`**`draw_contours`**`(`*map*, *contours*, *args*`)`

`pixell.enplot.`**`parse_annotations`**`(`*afile*`)`

`pixell.enplot.`**`draw_annotations`**`(`*map*, *annots*, *args*`)`

Draw a set of annotations on the map. These are specified as a list of ["type",param,param,…]. The recognized formats are:

c[ircle] lat lon dy dx [rad [width [color]]] t[ext] lat lon dy dx text [size [color]] l[ine] lat lon dy dx lat lon dy dx [width [color]] r[ect] lat lon dy dx lat lon dy dx [width [color]]

dy and dx are pixel-unit offsets from the specified lat/lon. This is useful for e.g. placing text next to circles.

`pixell.enplot.`**`standardize_images`**`(`*tuples*`)`

Given a list of (img,bounds), composite them on top of each other (first at the bottom), and return the total image and its new bounds.

`pixell.enplot.`**`merge_images`**`(`*images*`)`

Stack all images into an alpha composite. Images must all have consistent extent before this. Use standardize_images to achieve this.

`pixell.enplot.`**`merge_plots`**`(`*plots*`)`

`pixell.enplot.`**`prepare_map_field`**`(`*map*, *args*, *crange=None*, *printer=<pixell.enplot.Printer object>*`)`

`pixell.enplot.`**`makefoot`**`(`*n*`)`

`pixell.enplot.`**`contour_widen`**`(`*cmap*, *width*`)`

`pixell.enplot.`**`draw_ellipse`**`(`*image*, *bounds*, *width=1*, *outline='white'*, *antialias=1*`)`

Improved ellipse drawing function, based on PIL.ImageDraw. Improved from http://stackoverflow.com/questions/32504246/draw-ellipse-in-python-pil-with-line-thickness

`pixell.enplot.`**`hwexpand`**`(`*m*, *nrow=-1*, *ncol=-1*, *transpose=False*, *dims=None*`)`

Stack the maps in mflat[n,ny,nx] into a single flat map mflat[nrow,ncol,ny,nx]

`pixell.enplot.`**`hwstack`**`(`*mexp*`)`

`exception` `pixell.enplot.`**`BackendError`**

`pixell.enplot.`**`show`**`(`*img*, *title=None*, *method='auto'*`)`

`pixell.enplot.`**`show_ipython`**`(`*img*, *title=None*`)`

`pixell.enplot.`**`show_tk`**`(`*img*, *title=None*`)`

`pixell.enplot.`**`show_wx`**`(`*img*, *title=None*`)`

`pixell.enplot.`**`show_qt`**`(`*img*, *title=None*`)`

3.14 2.14 tilemap - Tiled maps where only some tiles are stored

`pixell.tilemap.`**`zeros`**`(`*tile_geom*, *dtype=<class 'numpy.float64'>*`)`

Construct a zero-initialized TileMap with the given TileGeometry and data type

`pixell.tilemap.`**`empty`**`(`*tile_geom*, *dtype=<class 'numpy.float64'>*`)`

Construct a zero-initialized TileMap with the given TileGeometry and data type

`pixell.tilemap.`**`full`**`(`*tile_geom*, *val*, *dtype=<class 'numpy.float64'>*`)`

Construct a zero-initialized TileMap with the given TileGeometry and data type

`pixell.tilemap.`**`from_tiles`**`(`*tiles*, *tile_geom*`)`

Construct a TileMap from a set of a full list of tiles, both active and inactive. Inactive tiles are indicated with None entries. The active information in tile_geom is ignored, as is the non-pixel part of tile_geom.shape, which is instead inferred from the tiles.

`pixell.tilemap.`**`from_active_tiles`**`(`*tiles*, *tile_geom*`)`

Construct a TileMap from a list of active tiles that should match the active list in the provided tile geometry. The non-pixel part of tile_geom is ignored, and is instead inferred from the tile shapes.

`class` `pixell.tilemap.`**`TileMap`**`(`*arr*, *tile_geom*`)`

Implements a sparse tiled map, as described by a TileGeometry. This is effectively a large enmap that has been split into tiles, of which only a subset is stored. This is implemented as a subclass of ndarray instead of a list of tiles to allow us to transparently perform math operations on it. The maps are stored stored as a single array with all tiles concatenated along a flattened pixel dimension, in the same order as in tile_geom.active.

Example: A TileMap covering a logical area with shape (3,100,100) with (10,10) tiles and active tiles [7,5] will have a shape of (3,200=10*10*2) when accessed directly. When accessed through the .tiles view, .tiles[5] will return a view of an (3,10,10) enmap, as will .tiles[7]. For all other indices, .tiles[x] will return None. The same two tiles can be accessed as .active_tiles[1] and .active_tiles[0] respecitvely.

Slicing the TileMap using the [] operator works. For all but the last axis, this does what you would expect. E.g. for the above example, tile_map[0].tiles[5] would return a view of a (10,10) enmap (so the first axis is gone). If the last axis, which represents a flattened view of the pixels and tiles, is sliced, then the returned object will be a plain numpy array.

`contig``()`

`property pre`

`copy``(`*order='C'*`)`

Return a copy of the array.

Parameters

order (*{'C', 'F', 'A', 'K'}, optional*) – Controls the memory layout of the copy. 'C' means C-order, 'F' means F-order, 'A' means 'F' if *a* is Fortran contiguous, 'C' otherwise. 'K' means match the layout of *a* as closely as possible. (Note that this function and `numpy.copy()` are very similar but have different default values for their order= arguments, and this function always passes sub-classes through.)

See also:

`numpy.copy`
Similar function with different default behavior

`numpy.copyto`

Notes

This function is the preferred method for creating an array copy. The function `numpy.copy()` is similar, but it defaults to using order 'K', and will not pass sub-classes through by default.

Examples

```
>>> x = np.array([[1,2,3],[4,5,6]], order='F')
```

```
>>> y = x.copy()
```

```
>>> x.fill(0)
```

```
>>> x
array([[0, 0, 0],
       [0, 0, 0]])
```

```
>>> y
array([[1, 2, 3],
       [4, 5, 6]])
```

```
>>> y.flags['C_CONTIGUOUS']
True
```

property tiles

property active_tiles

with_tiles(*other*, *strict=False*)

If a and b are TileMaps with the same overall tiling but different active tile sets, then c = a.with_tiles(b) will make c a TileMap with the union of the active tiles of a and b and the data from a (new tiles are zero-initialized).

If strict==True, then c will have exactly the active tiles of b, in exactly that order. Binary operations on strictly compatible arrays should be considerably faster.

property active

property lookup

property nactive

property ntile

property tile_shape

insert(*imap*, *op=<function TileMap.<lambda>>*)

class pixell.tilemap.**TileView**(*tile_map*, *active=True*)

Helper class used to implement access to the individual tiles that make up a TileMap object

property ndim

property shape

pixell.tilemap.**make_binop**(*op*, *is_inplace=False*)

pixell.tilemap.**insert**(*omap*, *imap*, *op=<function <lambda>>*)

Insert imap into omap, returning the result. Equivalent to enmap.insert, but with the following important differences: * omap is not modified. Use the result is returned. (enmap both modifies and returns) * The maps must have the same geometry, only differing by the active tiles. This may be generalized in the future.

`pixell.tilemap.`**`map_mul`**`(`*mat*, *vec*`)`

Elementwise matrix multiplication mat*vec. Result will have the same shape as vec. Multiplication happens along the last non-pixel indices.

`pixell.tilemap.`**`samegeo`**`(`*arr*, **args*`)`

Returns arr with the same geometry information as the first tilemap among args. If no matches are found, arr is returned as is. Will reference, rather than copy, the underlying array data whenever possible.

`pixell.tilemap.`**`geometry`**`(`*shape*, *wcs*, *tile_shape=(500, 500)*, *active=[]*`)`

TileGeometry constructor. shape, wcs: The enmap geometry of the full space the tiling covers. tile_shape: The (ny,nx) vertical and horizontal tile shape in pixels. active: The list of active tile indices.

`class pixell.tilemap.`**`TileGeometry`**`(`*shape*, *wcs*, *tile_shape*, *grid_shape*, *tile_shapes*, *npixs*, *active*, *lookup*`)`

`grid2ind``(`*ty*, *tx*`)`

Get the index of the tile wiht grid coordinates ty,tx in the full tiling

`ind2grid``(`*i*`)`

Get the tile grid coordinates ty, tx for tile #i in the full tiling

`copy``(`*pre=None*, *active=None*, *add_active=None*`)`

`property pre`

`property nactive`

`property size`

`property tiles`

Allow us to get the enmap geometry of tile #i by writing tile_geom.tiles[i]

`compatible``(`*other*`)`

Return our compatibility with binary operations with other. The return value can be 2, 1 or 0: 2: Strictly compatible. Both the logical geometry (shape, wcs), tile shape

> and active tiles match. This allows for direct numpy operations without any manual looping over tiles.

1: Loosely compatible. The logical and tile geometry match, but not the active tiles. 0. Not compatible.

`pixell.tilemap.`**`to_enmap`**`(`*tile_map*`)`

`pixell.tilemap.`**`redistribute`**`(`*imap*, *comm*, *active=None*, *omap=None*`)`

Redistirbute the data in the mpi-distributed tiles in imap into the active tiles in omap, using the given communicator. If a tile is active in multiple tasks in imap, it will be reduced. If it is active in multiple tiles in omap, it will be duplicated.

`pixell.tilemap.`**`tree_reduce`**`(`*imap*, *comm*, *plan=None*`)`

Given a tilemap imap that's distributed over the communicator comm, and where each tile is potentially present in multiple tasks, sum the duplicate tiles and assign them to a single task, such that in the end each tile is present in at most one task. Exactly which tiles end up in which tasks is determined automatically but deterministically based on the tile ownership pattern in imap.

`pixell.tilemap.`**`get_active_distributed`**`(`*tile_map*, *comm*`)`

`pixell.tilemap.`**`reduce`**`(`*tile_map*, *comm*, *root=0*`)`

Given a distributed TileMap tile_map, collect all the tiles on the task with rank root (default is rank 0), and return it. Multiply owned tiles are reduced. Returns a TileMap with no active tiles for other tasks than root.

`pixell.tilemap.`**`write_map`**(*fname*, *tile_map*, *comm*)

Write a distributed tile_map to disk as a single enmap. Collects all the data on a single task before writing.

CHAPTER

FOUR

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

4.1 Types of Contributions

4.1.1 Report Bugs

Report bugs at https://github.com/simonsobs/pixell/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

4.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

4.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

4.1.4 Write Documentation

pixell could always use more documentation, whether as part of the official pixell docs, in docstrings, or even on the web in blog posts, articles, and such.

4.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/simonsobs/pixell/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

4.2 Get Started!

Ready to contribute? Here's how to set up *pixell* for local development.

1. Fork the *pixell* repo on GitHub.
2. Clone your fork locally:

   ```
   $ git clone git@github.com:your_name_here/pixell.git
   ```

3. Install your local copy for development:

   ```
   $ cd pixell/
   $ python setup.py build_ext -i
   ```

and add the cloned directory to your Python path so that changes you make in any python file are immediately reflected. e.g., in your `.bashrc` file:

```
export PYTHONPATH=$PYTHONPATH:/path/to/cloned/pixell/directory
```

4. Create a branch for local development:

   ```
   $ git checkout -b name-of-your-bugfix-or-feature
   ```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests:

   ```
   $ flake8 pixell tests
   $ py.test
   ```

   To get flake8, just pip install it into your virtualenv.

6. Commit your changes and push your branch to GitHub:

   ```
   $ git add .
   $ git commit -m "Your detailed description of your changes."
   $ git push origin name-of-your-bugfix-or-feature
   ```

7. Submit a pull request through the GitHub website.

4.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 3.6, 3.7 and 3.8. Check https://github.com/simonsobs/pixell/actions and make sure that the tests pass for all supported Python versions.

4.4 Deploying

Only maintainers, who have access to the master branch, are able to deploy the package. This is accomplished by associating a tag, of the form vX.y.z, to the relevant commit in the master branch. We use bumpversion for this, in a way that is compatible with versioneer. Before initiating the release, be sure to update HISTORY.rst with the differences since last version (not required while we're still in 0.y.z) and commit the changes. Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Github Actions will then deploy to PyPI if tests pass.

The role of versioneer is to automatically embed version information in the distributed source code or installed package, based on the github tags. The role of bumpversion (in our configuration) is to generate sequential version numbers and create github corresponding git tags. The bumpversion and versioneer configurations are in `setup.cfg`.

CHAPTER

FIVE

CREDITS

5.1 Development Leads

- Sigurd Naess (@amaurea)
- Mathew Madhavacheril (@msyriac)
- Matthew Hasselfield (@mhasself)

5.2 Other Contributors

- Adri Duivenvoorden
- Alex van Engelen
- Xavier Garrido
- Matt Hilton
- Adrien La Posta
- Simon Foreman

CHAPTER

SIX

HISTORY

6.1 0.21.0 (2023-10-19)

Changes relative to 0.19.2 include:

- More miscellaneous fixes after transition to ducc0
- More work on wavelets
- We now provide MacOS x86_64 wheels
- Improved build system that respects choices of CC, CXX, and FC

6.2 0.19.2 (2023-08-18)

Changes relative to 0.19.0 include:

- Important bugfixes for the migration from libsharp2 to ducc0
- Improved SHT unit tests

6.3 0.19.0 (2023-07-14)

Changes relative to 0.17.3 include:

- Migrate fully from libsharp2 to ducc0 (for curved sky functions)
- Temporary suspension of MacOS pip binaries (use *pip install pixell==0.17.3* for Macs in the meantime)
- Miscellaneous fixes

6.4 0.17.3 (2023-03-17)

Changes relative to 0.17.2 include:

- More support for fejer1

6.5 0.17.2 (2023-02-21)

Changes relative to 0.17.1 include:

- Build for Python 3.11

6.6 0.17.1 (2023-01-26)

Changes relative to 0.16.0 include:

- Bilinear map-making pixel window function
- Miscellaneous new functions and API improvements
- Miscellaneous bug fixes
- Fixes for Apple Silicon

6.7 0.16.0 (2022-06-08)

Changes relative to 0.15.3 include:

- Wavelet analysis
- Fast C-based source simulation
- Fast vectorized radial profile binning
- Fixes and improvements to HDF5 I/O
- Fixes to OSX support

6.8 0.15.3 (2022-02-12)

Changes relative to 0.15.1 include:

- New wheels that fix numpy binary incompatibility errors

6.9 0.15.1 (2022-01-23)

Changes relative to 0.14.3 include:

- More flexible enmap.read_map_geometry
- Add Python 3.10 support, drop Python 3.6 support

6.10 0.14.3 (2021-12-13)

Changes relative to 0.14.2 include:

- Updates to enmap.insert, UHTs and WCS string accuracy

6.11 0.14.2 (2021-11-23)

Changes relative to 0.14.1 include:

- An important bugfix for enmap.downgrade when the *op* argument is passed. This bug has been present since v0.14.0 and in commits on master since Aug 12, 2021.

6.12 0.14.1 (2021-11-16)

Changes relative to 0.13.2 include:

- A breaking change to map2alm where it no longer approximates WCS if ring weights are unavailable
- Miscellaneous bug fixes
- ducc0 FFT support and fast rotate_alm
- Tiled map support
- New healpix <-> rectpix reprojection API

6.13 0.13.2 (2021-07-16)

Changes relative to 0.13.1 include:

- Added binaries for MacOS 11 Big Sur

6.14 0.13.1 (2021-07-08)

Changes relative to 0.13.0 include:

- Fixes to the MacOS wheel building

6.15 0.13.0 (2021-07-08)

Changes relative to 0.12.1 include:

- Matched filtering in a new analysis module
- Conjugate gradients solver
- Discrete cosine transforms
- Miscellaneous bug fixes

6.16 0.12.1 (2021-04-30)

Changes relative to 0.12.0 include:

- Patch to fix numpy binary incompatibility issues caused by changes to the numpy C API. We now require numpy >1.20.

6.17 0.12.0 (2021-04-13)

Changes relative to 0.11.2 include:

- We now use libsharp2 instead of libsharp, which has signficantly faster SHTs
- Major breaking change: the meaning of the "iau" flag has been corrected and reversed. The default behaviour of map2harm and other functions using this flag will be different.
- Unified harmonic transforms module
- postage_stamp removed in favor of thumbnails
- Adjoint harmonic transforms

6.18 0.11.2 (2021-02-04)

Changes relative to 0.11.0 include:

- Bug-fix for distance_transform when using rmax

6.19 0.11.0 (2021-02-02)

Changes relative to 0.10.3 include:

- Bug-fix for enmap.project that led to crashes
- enplot improvements
- Improvements to fft and ifft overhead
- alm filtering API improvements
- Changes to CMB dipole parameter
- Allow lmax!=mmax in curvedsky routines
- Python 3.9 builds and Github actions instead of Travis

6.20 0.10.3 (2020-06-26)

Changes relative to 0.10.2 include:

- Bug fix for automatic IAU -> COSMO, recognizes POLCCONV instead of POLCONV.

6.21 0.10.2 (2020-06-26)

Changes relative to 0.9.6 include:

- Automatically converts maps recognized to be in IAU polarization convention (through the FITS header) to COSMO convention by flipping the sign of U
- Fixes a centering issue in reproject.thumbnails
- Optimizes posmap for separable projections and pixsizemap for cylindrical projections making these functions orders of magnitude faster for CAR (and other projections)
- A test script test-pixell is distributed with the package

6.22 0.9.6 (2020-06-22)

Changes relative to 0.6.0 include:

- Ability to read compressed FITS images
- Fixed a bug to make aberration and modulation accurate to all orders
- Expanded alm2cl to handle full cross-spectra and broadcasting

6.23 0.6.0 (2019-09-18)

Changes relative to 0.5.2 include:

- Improvements in accuracy for map extent, area and Fourier wavenumbers
- Spherical harmonic treatment consistent with healpy
- Additional helper functions, e.g enmap.insert
- Helper arguments, e.g. physical normalization for enmap.fft
- Bug fixes e.g. in rand_alm
- Improved installation procedure and documentation

6.24 0.5.2 (2019-01-22)

- API for most modules is close to converged
- Significant number of bug fixes and new features
- Versioning system implemented through versioneer and bumpversion
- Automated pixel level tests for discovering effects of low-level changes

6.25 0.1.0 (2018-06-15)

- First release on PyPI.

CHAPTER SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX