pixell Documentation

Release 0.7.0+24.g58cae0f.dirty

Simons Observatory Collaboration Analysis Library Task Force

Mar 27, 2020

Contents:

CHAPTER 1

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 libsharp) 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>=2.7 or Python>=3.4
- gcc/gfortran or Intel compilers (clang might not work out of the box)
- libsharp (downloaded and installed)
- automake (for libsharp compilation)
- healpy, Cython, astropy, numpy, scipy, matplotlib, pyyaml, h5py, Pillow (Python Image Library)

1.2 Installing

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

For installation instructions specific to MacOS X, see MACOSX (h/t Thibaut Louis).

For all other, below are general instructions.

To install, clone this repository and run:

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

To test the installation, you can run:

```
$ py.test
```

You may need to install pytest for the above to work (with *pip install pytest –user*).

1.2.1 Existing `libsharp` installation (optional)

Libsharp is installed automatically by setup.py. If instead you want to use an existing `libsharp` installation, you can do so by symlinking the `libsharp` directory into a directory called `_deps` in the root directory, such that the file `pixell/_deps/libsharp/libsharp/sharp.c` exists.

1.2.2 Intel compilers

Intel compilers might require a two step installation as follows

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

1.2.3 Development workflow (recommended)

If you are a developer, run:

```
$ 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
```

If you also need non-Python code to be recompiled, run:

```
$ python setup.py clean
```

before the above steps.

To test the installation under development mode, you can run:

```
$ py.test
```

This requires the pytest Python package to be installed.

1.3 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 2

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'
>>> # Uh oh, the WCS information was removed by some_function
>>> omap = enmap.enmap(omap,wcs) # restore the wcs
>>> omap = enmap.samewcs(omap,imap) # another way to restore the wcs
```

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 3

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**

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.)

See also:

`numpy.copy(), numpy.copyto()`

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*)

box(*corner=True*)

pixbox_of(*oshape*, *owcs*)

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

pixmap()

lmap(*oversample=1*)

lform(*shift=True*)

modlmap(*oversample=1*)

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

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

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

area()

pixsize()

pixshape(*signed=False*)

pixsizemap()

pixshapemap()

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

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

npix

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 <lambda>>*, *cval=0*, *iwcs=None*, *reverse=False*)

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

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

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

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

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='bubble'*, *rmax=None*, *step=1024*)

distance_transform(*omap=None*, *rmax=None*)

labeled_distance_transform(*omap=None*, *odomains=None*, *rmax=None*)

plain

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

center()

downgrade(*factor*)

upgrade(*factor*)

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]] bounding box to select. The resulting map will have a bounding box 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]] bounding box to select. The resulting map will have a bounding box 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 bounding box provided into a pixel units. Assumes rectangular coordinates.

When translated to box 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*)

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

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

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

scale(*scale*)

downgrade(*factor*)

copy()

pixell.enmap.**box**(*shape*, *wcs*, *npoint=10*, *corner=True*)
Compute a bounding box for the given geometry.

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=False*, *dtype=<type 'numpy.float64'>*, *bsize=1000000.0*)
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.

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

pixell.enmap.**posaxes**(*shape*, *wcs*, *safe=True*, *corner=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*)
Given an array of corner-based pixel coordinates [{y,x},...], return sky coordinates in the same ordering.

pixell.enmap.**sky2pix**(*shape*, *wcs*, *coords*, *safe=True*, *corner=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.**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.**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,2] 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 area of a single pixel, in steradians.

pixell.enmap.**pixshape**(*shape*, *wcs*, *signed=False*)
Returns the height and width of a single pixel, in radians.

pixell.enmap.**pixshapemap**(*shape*, *wcs*, *bsize=1000*)
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.**pixsizemap**(*shape*, *wcs*, *bsize=1000*)
Returns the physical area of each pixel in the map in steradians. Heavy for big maps.

pixell.enmap.**lmap**(*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.**modlmap**(*shape*, *wcs*, *oversample=1*)
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*)

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.**fft**(*emap*, *omap=None*, *nthread=0*, *normalize=True*)
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*)
Performs the 2d iFFT of the complex enmap given, and returns a pixel-space enmap.

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

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

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

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.

pixell.enmap.**calc_window**(*shape*)
Compute fourier-space window function. Like the other fourier-based functions in this module, equi-spaced pixels are assumed. Since the window function is separable, it is returned as an x and y part, such that window = wy[:,None]*wx[None,:].

pixell.enmap.**apply_window**(*emap*, *pow=1.0*)
Apply the pixel window function to the specified power to the map, returning a modified copy. Use pow=-1 to unapply the pixel window.

pixell.enmap.**samewcs**(*arr*, **args*)
Returns arr with the same wcs information as the first enmap among args. If no mathces are found, arr is returned as is.

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}] 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 {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 bounding box can differ by a

fraction of a pixel from the one requested. To force the geometry to exactly match the bounding box 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'*)
Build an enmap covering the full sky, with the outermost pixel centers at the poles and wrap-around points. Assumes a CAR (clenshaw curtis variant) projection for now.

pixell.enmap.**band_geometry**(*dec_cut*, *res=None*, *shape=None*, *dims=()*, *proj='car'*)
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.**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.**multi_pow**(*mat, exp, axes=[0, 1]*)
Raise each sub-matrix of mat (ncomp,ncomp,...) to the given exponent in eigen-space.

pixell.enmap.**downgrade**(*emap*, *factor*)
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*)
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*)
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*)
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='bubble'*, *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 "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.**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. If pix is a scalar, it specifies the number of pixels to add on all sides. If it is 1d, it specifies the number of pixels to add at each end for each axis. If it is 2d, the number of pixels to add at each end of an axis can be specified individually.

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.**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.**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*)

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] and 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*)

Like rbin, but for fourier space

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*, *extra={}*)
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.

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*)
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'.

pixell.enmap.**read_map_geometry**(*fname*, *fmt=None*, *hdu=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 supposed for now, though.

pixell.enmap.**write_fits**(*fname*, *emap*, *extra={}*)
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*)
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*)
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*, *extra={}*)
Write an enmap as an hdf file, preserving all the WCS metadata.

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*)
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.

pixell.enmap.**read_hdf_geometry**(*fname*)
Read an enmap wcs from the specified hdf file.

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*, *sel_threshold=10000000.0*)
Helper function for map reading. Handles the slicing, sky-wrapping and capping, etc.

class pixell.enmap.**fits_wrapper**(*hdu*, *wcs*, *threshold=10000000.0*)

shape

class pixell.enmap.**hdf_wrapper**(*dset*, *wcs*, *threshold=10000000.0*)

shape

ndim

dtype

pixell.enmap.**fix_endian**(*map*)
Make endianness of array map match the current machine. Returns the result.

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.**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.**spin_helper**(*spin*, *n*)

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.

pixell.fft.**numpy_n_byte_align_empty**(*shape*, *alignment*, *dtype*)
This dummy function just skips the alignment, since numpy doesn't provide an easy way to get it.

class pixell.fft.**NumpyEngine**

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

pixell.fft.**fft**(*tod, ft=None, nthread=0, axes=[-1], flags=None*)
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. 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. If ft is left out, a complex transform is assumed.

pixell.fft.**ifft**(*ft, tod=None, nthread=0, normalize=False, axes=[-1], flags=None*)
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 nrmalized, 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*)
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*)
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.**redft00**(*a*, *b=None*, *nthread=0*, *normalize=False*, *flags=None*)
pyFFTW does not support the DCT yet, so this is a work-around. 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*)
The chebyshev transform of a, along its last dimension.

pixell.fft.**ichebt**(*a*, *b=None*, *nthread=0*)
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.**shift**(*a*, *shift*, *axes=None*, *nofft=False*, *deriv=None*)
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.

3.3 2.3 curvedsky - Curved-sky harmonic transforms

3.4 2.4 utils - General utilities

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.**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.**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.**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.**unwind**(*a, period=6.283185307179586, axes=[-1], ref=0*)
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.**mjd2ctime**(*mjd*)
Converts from modified julian date to unix time

pixell.utils.**medmean**(*x*, *frac=0.5*)

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

pixell.utils.**moveaxes**(*a*, *old*, *new*)
Move the axes listed in old to the positions given by new. This is like repeated calls to numpy rollaxis while taking into account the effect of previous rolls.

This version is slow but simple and safe. It moves all axes to be moved to the end, and then moves them one by one to the target location.

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*)

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.**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*)
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.**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.**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.**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.**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.**to_Nd**(*a*, *n*, *return_inverse=False*)

pixell.utils.**between_angles**(*a*, *range*, *period=6.283185307179586*)

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.**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 gatherv([[1,2]],comm) on one task and gatherv([[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*)
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=0*)
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 + (3,3).

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,ndim], return groups[ngroup][{ind,ind,ind,...}] of indices into a for which all the values in the second index of a is the same. group_equal([[0,1],[1,2],[0,1]]) -> [[0,2],[1]].

pixell.utils.**minmax**(*a*, *axis=None*)
Shortcut for np.array([np.min(a),np.max(a)]), since I do this a lot.

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.**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.**triangle_wave**(*x*, *period=1*)
Return a triangle wave with amplitude 1 and the given period.

pixell.utils.**flux_factor**(*beam_area*, *freq*, *T0=2.725*)
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=2.725*)
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.**planck**(*f*, *T*)
Return the Planck spectrum at the frequency f and temperature T in Jy/sr

pixell.utils.**blackbody**(*f*, *T*)
Return the Planck spectrum at the frequency f and temperature T in Jy/sr

pixell.utils.**graybody**(*f*, *T*, *beta=1*)
Return a graybody spectrum at the frequency f and temperature T in Jy/sr

pixell.utils.**edges2bins**(*edges*)

pixell.utils.**bins2edges**(*bins*)

pixell.utils.**linbin**(*n*, *nbin=None*, *nmin=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.**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.**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.

pixell.utils.**fix_dtype_mpi4py**(*dtype*)
Work around mpi4py bug, where it refuses to accept dtypes with endian info

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

3.5 2.5 reproject - Map reprojection

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=<type 'numpy.float64'>, oversample=2.0, spin=[0, 2], output='l', geodesic=True, verbose=False, delta_theta=None*)

pixell.lensing.**rand_map**(*shape, wcs, ps_lensinput, lmax=None, maplmax=None, dtype=<type 'numpy.float64'>, seed=None, phi_seed=None, oversample=2.0, 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_srcs**(*shape*, *wcs*, *srcs*, *beam*, *omap=None*, *dtype=None*, *nsigma=5*, *rmax=None*, *smul=1*, *return_padded=False*, *pixwin=False*, *op=<ufunc 'add'>*, *wrap='auto'*, *verbose=False*, *cache=None*)
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=<type 'numpy.float64'>*, *op=<ufunc 'add'>*, *verbose=False*)

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.**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.

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.**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*)

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.**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.**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.**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*)

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.

3.12 2.12 powspec - CMB power spectra utilities

pixell.powspec.**sym_compress**(*mat, which=None, n=None, scheme=None, axes=[0, 1]*)
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*)
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*)

pixell.powspec.**scale_camb_scalar_phi**(*a*, *direction*)

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.

CHAPTER 4

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 2.7, 3.4, 3.5 and 3.6, and for PyPy. Check https://travis-ci.org/simonsobs/pixell/pull_requests 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). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

Travis 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 5

Credits

5.1 Development Lead

- Simons Observatory Collaboration Analysis Library Task Force

5.2 Contributors

- Sigurd Naess (@amaurea)
- Mathew Madhavacheril (@msyriac)
- Matthew Hasselfield (@mhasself)

CHAPTER 6

History

6.1 0.1.0 (2018-06-15)

- First release on PyPI.

6.2 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

6.3 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

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

Z