mhealpy

Release 0.3.1.dev0

Israel Martinez-Castellanos

Aug 23, 2023

CONTENTS:

HEALPix is a **H**ierarchical, **E**qual **A**rea, and iso-**L**atitude **Pix**elisation of the sphere. It has been implemented in multiple languages, including Python through the healpy library.

mhealpy is an object-oriented wrapper of *healpy*, in the fashion of Healpix C++, that extends its functionalities to handle multi-resolution maps.

Reference: Martinez-Castellanos, I. et al. *Multi-Resolution HEALPix Maps for Multi-Wavelength and Multi-Messenger Astronomy*. arXiv: 2111.11240 [astro-ph.IM] (2021).

CHAPTER

ONE

INSTALLATION

Run:

```
pip install mhealpy
```

Or alternatively, install it from source:

```
pip install --user git+https://gitlab.com/burstcube/mhealpy.git@master
```

1.1 For developers

First, install *healpy*, the only dependency:

```
pip install healpy
```

Then you can get a working version of *mhealpy* with:

```
git clone git@gitlab.com:burstcube/mhealpy.git
cd  mhealpy
python setup.py develop
```

CHAPTER

TWO

QUICK START

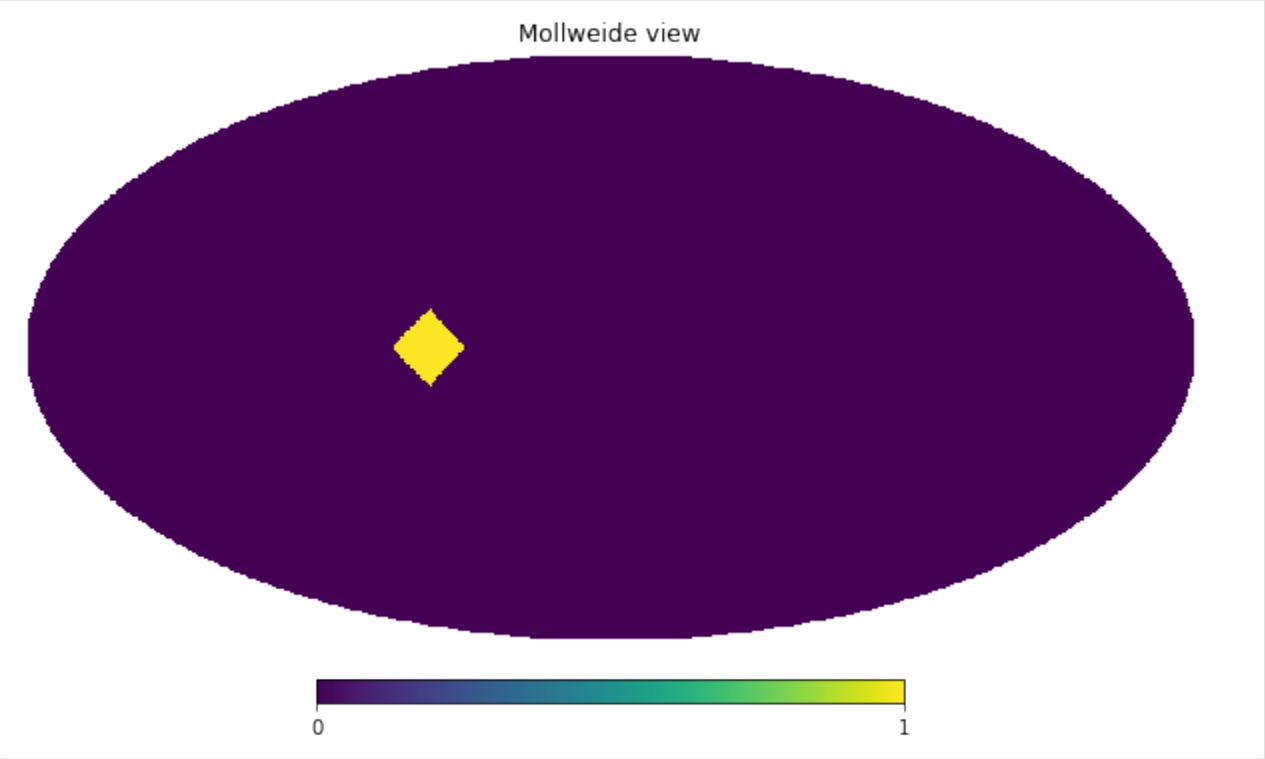
This tutorial shows you how to handle single and multi-resolution maps (a.k.a. multi-order coverage maps or MOC maps). It assumes previous knowledge of HEALPix. If you already are a healpy user, it should be straightforward to start using *mhealpy*.

See also the *API* documentation, as this is not meant to be exhaustuve.

2.1 *mhealpy* as an object-oriented *healpy* wrapper

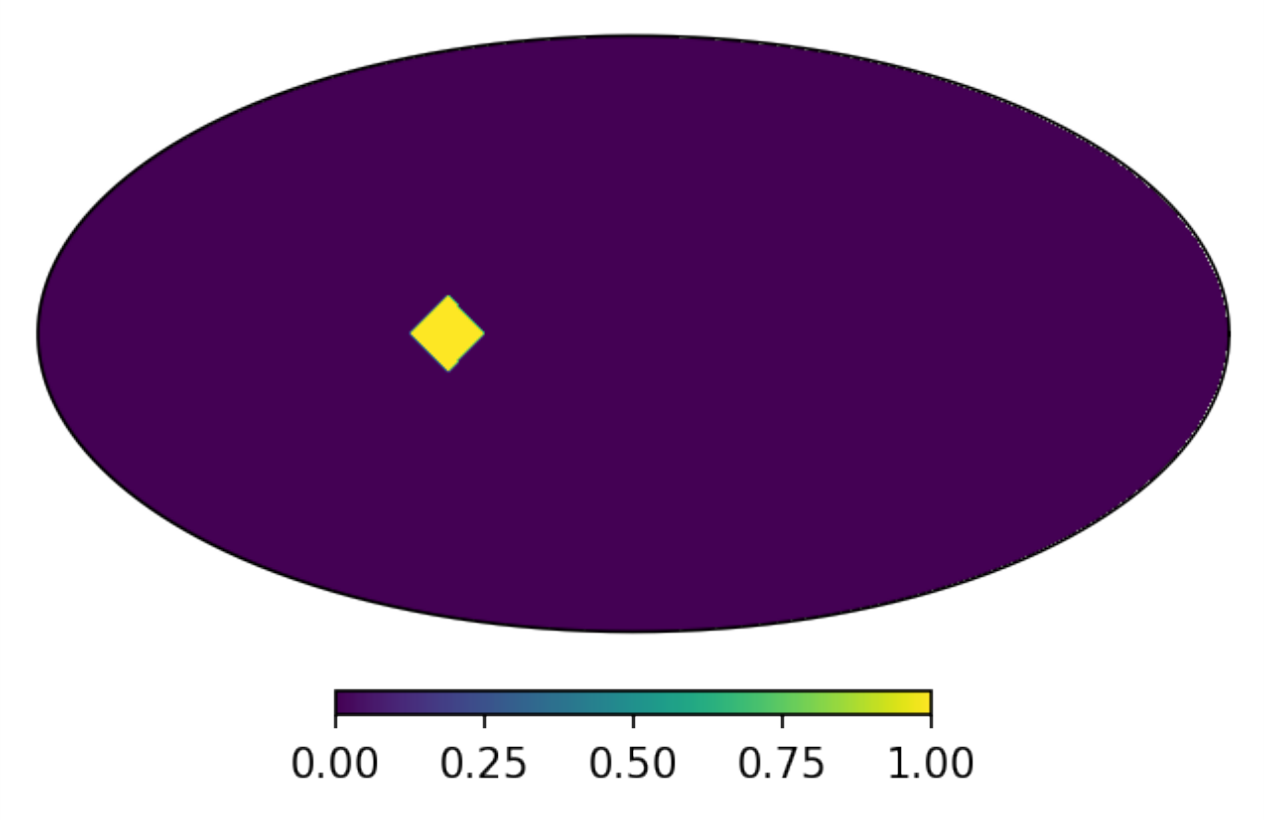
A single-resolution map is completely defined by an order ($npix = 12 * 4^{order} = 12 * nside^2$), a scheme (*RING* or *NESTED*) and an list of the maps contents. In *healpy* there is no class that contains this information, but rather the user needs to keep track of these and pass this information around to various functions. For example, to fill a map you can do:

[1]:
```
import numpy as np
import healpy as hp

# Define the grid
nside = 4
scheme = 'nested'
is_nested = (scheme == 'nested')

# Initialize the "map", which is a simple array
data = np.zeros(hp.nside2npix(nside))

# Get the pixel where a point lands in the current scheme
theta = np.deg2rad(90)
phi = np.deg2rad(50)

sample_pix = hp.ang2pix(nside, theta, phi, nest = is_nested)

# Add the count
data[sample_pix] += 1

# Save to disc
hp.write_map("my_map.fits", data, nest = is_nested, overwrite=True, dtype = int)

# Plot
hp.mollview(data, nest = is_nested)
```



At zeroth-order, *HealpixMap* is a container that keeps track of the information defining the grid. The equivalent code would look like:

[2]:
```
from mhealpy import HealpixMap

# Define the grid and initialize
m = HealpixMap(nside = nside, scheme = scheme, dtype = int)

# Get the pixel where a point lands in the current scheme
sample_pix = m.ang2pix(theta, phi)

# Add the count
m[sample_pix] += 1

# Save to disc
m.write_map("my_map.fits", overwrite=True)

# Plot
m.plot();
```



HealpixMap objects are array-like, which mean you can cast them into a *numpy array*, do slicing and indexing, iterate over them and use built-in functions. For example:

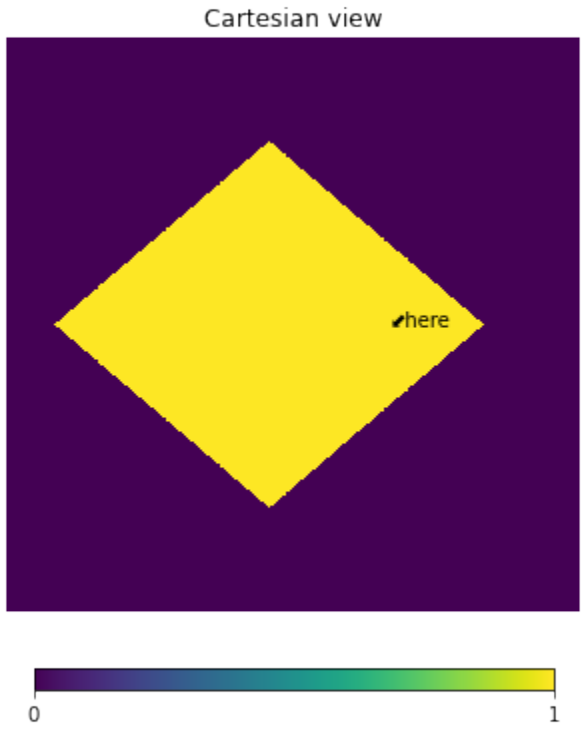
[3]:
```
data = np.array(m)

print("Data: {}".format(data))

print("Max: {}".format(max(m)))

for pix,content in enumerate(m):
    if content > 0:
        print("Max center: {} deg".format(np.rad2deg(m.pix2ang(pix))))
```

```
Data: [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 0 0 0 0 0 0]
Max: 1
Max center: [90.   56.25] deg
```

2.2 Plotting

healpy contains multiple routines to plot maps with various projections, and add text, points and graticules. For example, we can do a zoom in the previous map with:

[4]:
```
hp.cartview(data, nest = is_nested, latra = [-15,15], lonra = [40,70])

hp.projtext(theta, phi, "here");
```



HealpixMap has a single *plot()* method, but it is more versatile. It plots into an astropy WCSAxes, a type of *matplotlib* Axes, which allows full control over how and where to plot the map, and to use the native methods such as plot(), scatter() and text(). While you can use any WCSAxes you want, `mhealpy` includes various custom axes that take similar parameters to `healpy`'s visualization methods. For example:

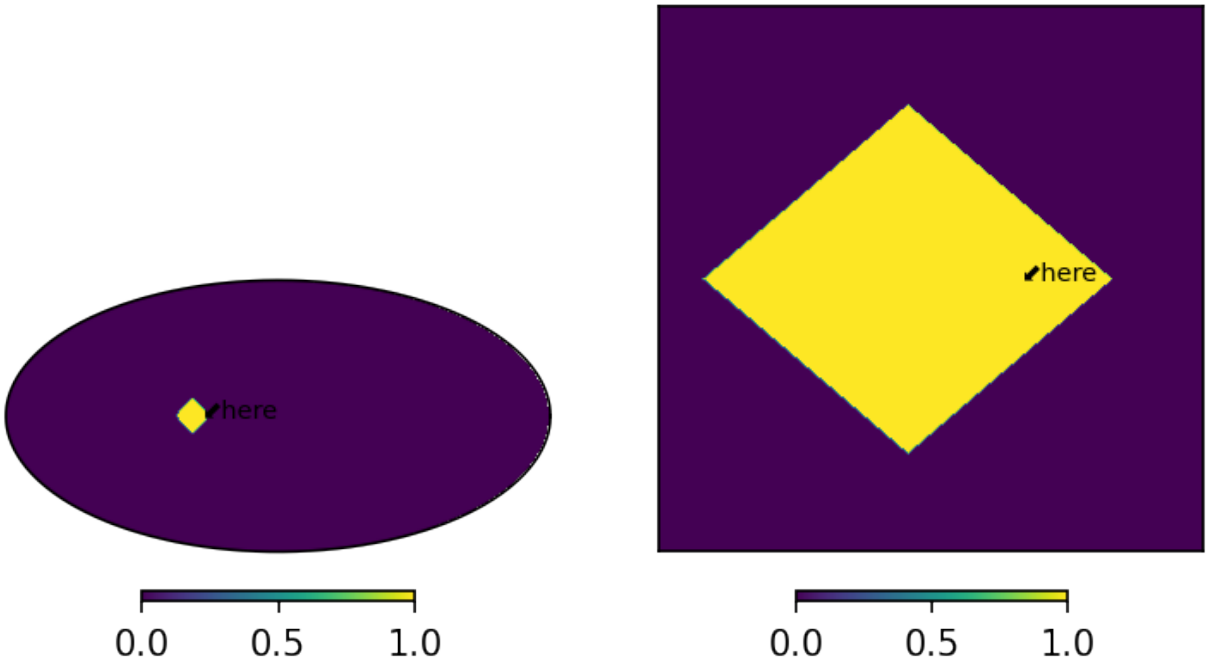
[5]:
```
import matplotlib.pyplot as plt

# Create custom axes
fig = plt.figure(dpi = 150)

axMoll = fig.add_subplot(1,2,1, projection = 'mollview')
axCart = fig.add_subplot(1,2,2, projection = 'cartview', latra = [-15,15], lonra = [40,
↪70])

# Plot in one of the axes
plotMoll, projMoll = m.plot(ax = axMoll)
plotCart, projCart = m.plot(ax = axCart)

# Use the projector to get the equivalent plot pixel for a given coordinate
# You can use it with any matplotlib method
```

(continues on next page)

(continued from previous page)

```
lon = np.rad2deg(phi)
lat = 90 - np.rad2deg(theta)
axMoll.text(lon, lat, "here", size = 7, transform = axMoll.get_transform('world'))
axCart.text(lon, lat, "here", size = 7, transform = axCart.get_transform('world'))
```

[5]:
```
Text(50.0, 0.0, 'here')
```



2.3 Order/scheme changes and the *density* parameter

In *healpy* you can change the underlaying grid of a map with a combination of ud_grade(), reorder(). The equivalent in a *HealpixMap* is to use *rasterize()*, e.g.:

[6]:
```
# Upgrade from nside = 4 to nside = 8, and change scheme from 'nested' to 'ring'
m_up = m.rasterize(8, scheme = 'ring')

print("New nside: {}".format(m_up.nside))
print("New scheme: {}".format(m_up.scheme))
print("New max: {}".format(max(m_up)))
print("New total: {}".format(sum(m_up)))
```

```
New nside: 8
New scheme: RING
New max: 0.25
New total: 1.0
```

In the original map we had one single count, so the maximum was 1. When we upgraded the resolution, the maximum of the new map is 0.25 but the total is still 1. This happened because, by default, the map are considered histograms and the value of new pixels is updated when they are split or combined. In this case, a pixel with a value of 1 was split into 4 child pixels, each with a value of 0.25.

This behaviour can be changed with the `density` parameter. If `True`, the value of each pixels is considered to be the evaluation of a function at the center of the pixel. Or equivalently, the map is considered a histogram whose contents have been divided by the area of the pixel, resulting in a density distribution.

[7]:
```
m.density(True)

# Change grid
m_up = m.rasterize(8, scheme = 'ring')

print("New nside: {}".format(m_up.nside))
print("New scheme: {}".format(m_up.scheme))
print("New max: {}".format(max(m_up)))
print("New total: {}".format(sum(m_up)))
```

```
New nside: 8
New scheme: RING
New max: 1.0
New total: 4.0
```

The maximum now stays a constant 1, but the total count is no longer conserved. We now have 4 pixels with the same value as the parent pixel.

2.4 Arithmethic operations

Regardless of the underlaying grid, you can operate on maps pixel-wise using *, /, +, -, **, == and `abs`. To illustrate this let's multiply two simple maps:

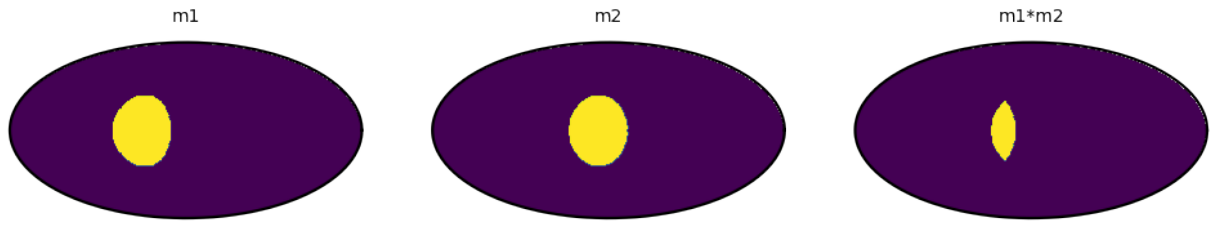
[8]:
```
import mhealpy as hmap

# Initialize map.
# Note this are density maps. This parameters comes into play when operatiing over two
# maps with different NSIDE
# If both maps are density-like, the result is also density-like,
# otherwise the result is histogram-like
m1 = HealpixMap(nside = 64, scheme = 'ring', density = True)
m2 = HealpixMap(nside = 128, scheme = 'nested', density = True)

# Fill first map with a simple disc
theta = np.deg2rad(90)
phi = np.deg2rad(45)
radius = np.deg2rad(30)
disc_pix = m1.query_disc(hmap.ang2vec(theta, phi), radius)

m1[disc_pix] = 1

# Fill second map with a similar disc, just shifted
phi = np.deg2rad(10)
disc_pix = m2.query_disc(hmap.ang2vec(theta, phi), radius)

m2[disc_pix] = 1

# Multiply
```

(continues on next page)

(continued from previous page)

```
mRes = m1*m2

print("Result nside: {}".format(mRes.nside))
print("Result scheme: {}".format(mRes.scheme))

# Plot side by side
fig,(ax1,ax2,axRes) = plt.subplots(1,3, dpi=200, subplot_kw = {'projection': 'mollview'})

ax1.set_title("m1", size= 5);
ax2.set_title("m2", size= 5);
axRes.set_title("m1*m2", size= 5);

m1.plot(ax1, cbar = None)
m2.plot(ax2, cbar = None)
mRes.plot(axRes, cbar = None);
```

```
Result nside: 128
Result scheme: NESTED
```

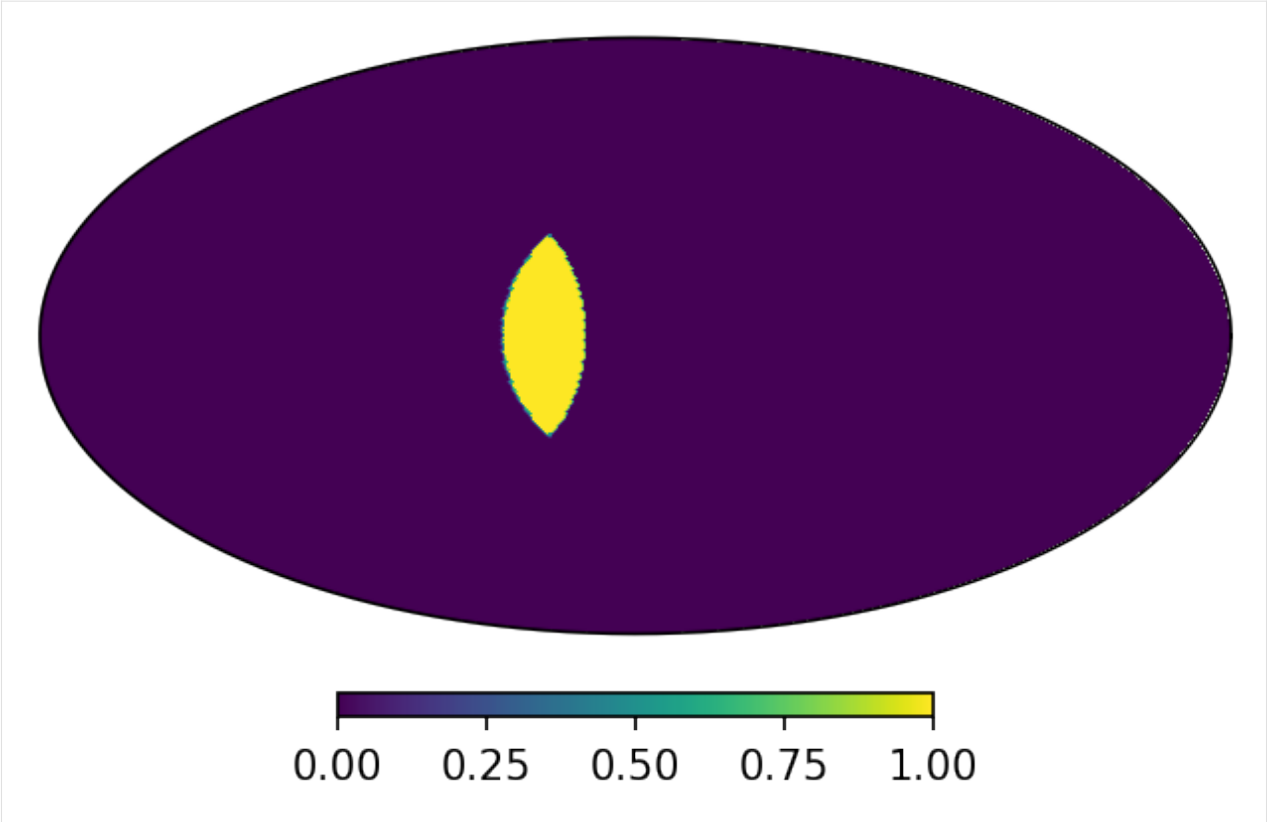


The resulting map of binary operation always has the finest grid of the two inputs. This ensures there is no loss of information. If both maps have the same *nside*, the output map has the scheme of the left operarand.

Sometimes though you want to keep the grid of a specific map. For that you can use in-place operations. If you really need a new map, you can use in-place operations in combination with `deepcopy`. For example:

[9]:
```
from copy import deepcopy

mRes = deepcopy(m1)

mRes *= m2

print("Result nside: {}".format(mRes.nside))
print("Result scheme: {}".format(mRes.scheme))

mRes.plot();
```

```
Result nside: 64
Result scheme: RING
```



2.5 Multi-resolution maps

Multi-order coverage (MOC) map –i.e. multi-resolution maps– are maps that tile the sky using pixels corresponding to different *nside*. Because a simple pixel number correspond to different locations depending on the map order, each pixel that composes the map needs to know the corresponding *nside*. Instead of storing two numbera though, we use the *NUNIQ* scheme, where each pixels is labeled by an *uniq* number defined as:

$$uniq = 4 * nside * nside + ipix,$$

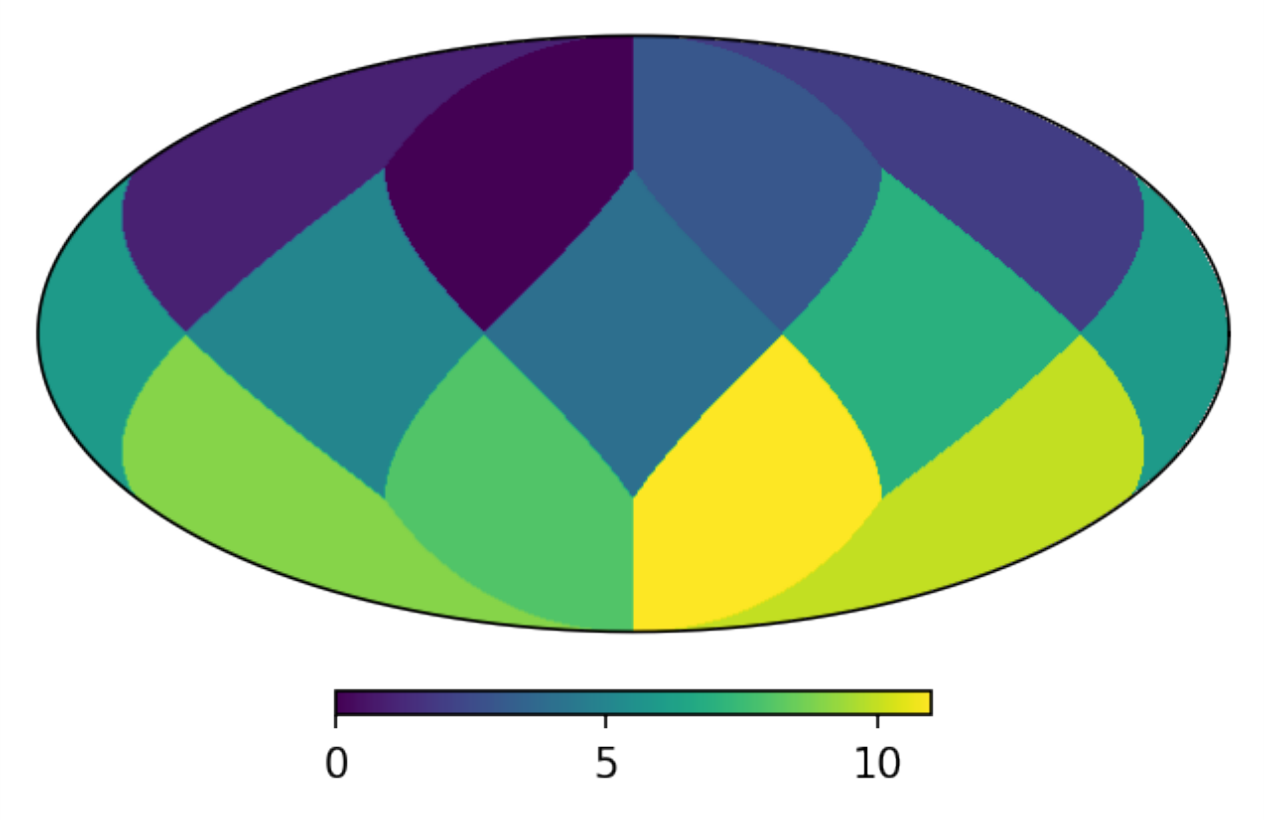
where $ipix$ corresponds to the pixel number in a *NESTED* scheme. This operation can be easily reversed to obtain both the *nside* and *ipix* values.

As a first example, let's assign to the 12 base pixels of a zero-order map the values of their own indices, but splitting the first pixel into the four child pixels of the nest order:

[10]:
```
#           child pixels of ipix=0 | rest of the base pixels
uniq     = [16, 17, 18, 19,          5, 6, 7, 8,  9, 10, 11, 12, 13, 14, 15]
contents = [0,   0,  0,  0,          1, 2, 3, 4,  5,  6,  7,  8,  9, 10, 11]

m = HealpixMap(contents, uniq, density = True)
```

If you plot it, it'll look as a single-resolution map:

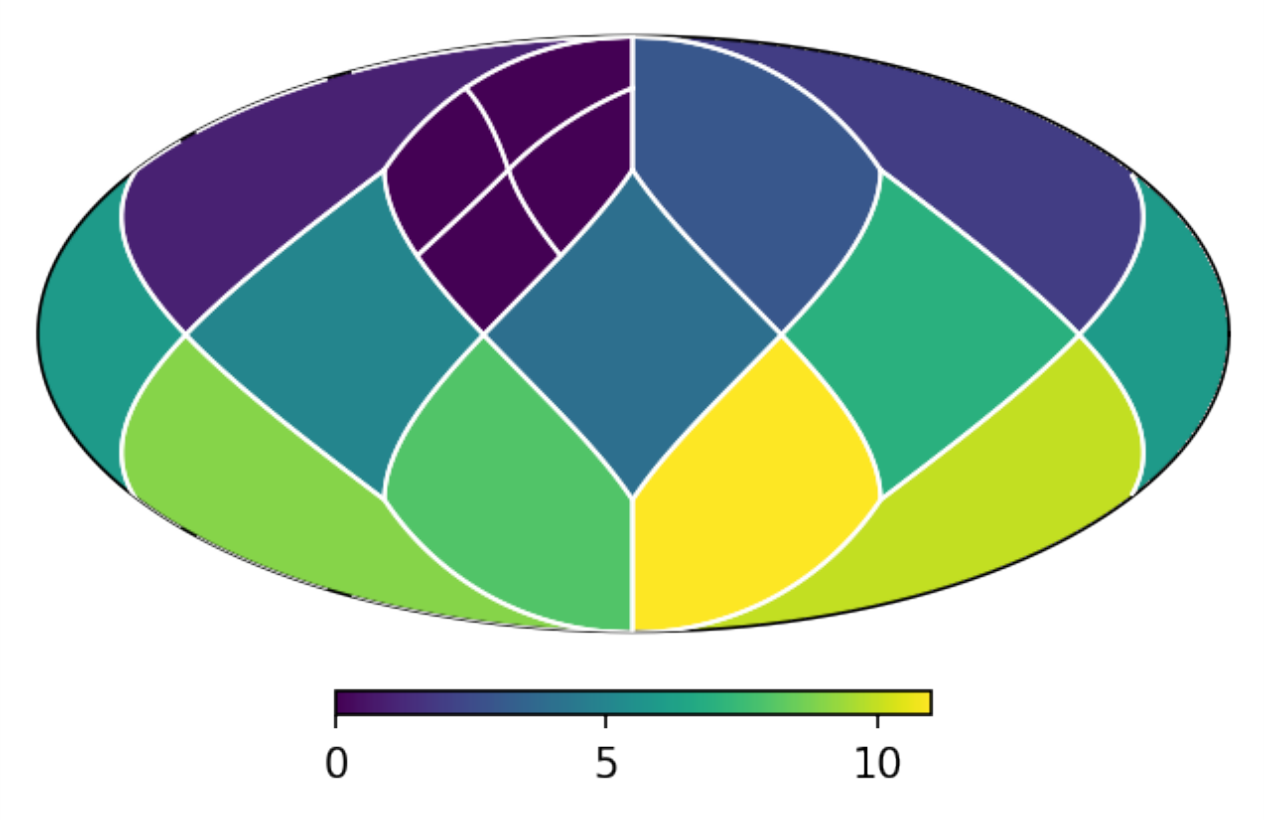
[11]:
```
m.plot();
```



But if we plot the pixel boundaries it'll be clear that his is a multi-resolution map:

[12]:
```
print("Is this a multi-resolution map? {}".format(m.is_moc))

m.plot()
m.plot_grid(ax = plt.gca(), color = 'white', linewidth = 1);
```

```
Is this a multi-resolution map? True
```



If you initialize the data of a MOC map by hand, it's generally a good idea to check that all locations of the sphere are covered by one and only one pixel:

[13]:
```
m.is_mesh_valid()
```

[13]:
```
True
```

2.5.1 Adaptive grids

Usually though, you don't need to create the grid of a MOC map by hand since *mhealpy* can choose an appropiate pixelation for you. The simplest case is when you know exactly which region needs a higher pixelation. For example, assume there is a source localized to a ~0.01deg reosolution. In order to achieve this resolution, you need a map with an *nside* of 16384 (the pixel size is ~0.003deg). It would be wasteful to hold in memory a full map for only this region of the sky.

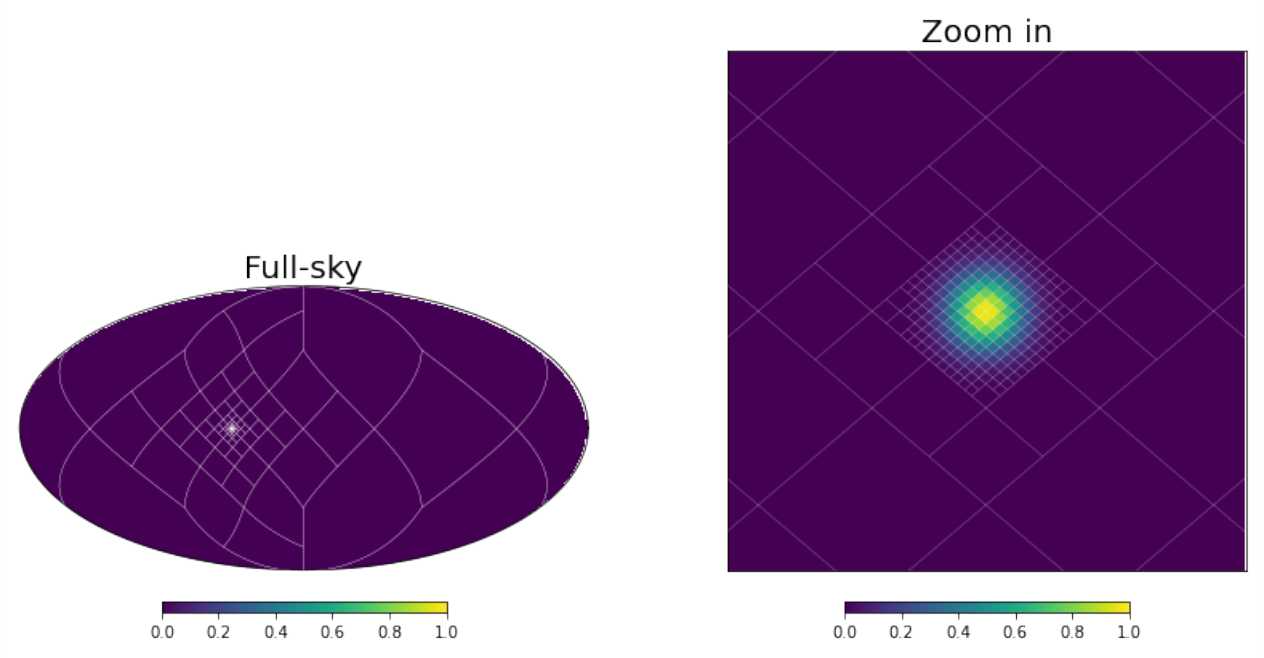
[14]:
```
from numpy import exp
from mhealpy import HealpixBase

# Location and uncertainty of the source
theta0 = np.deg2rad(90)
phi0 = np.deg2rad(45)
sigma = np.deg2rad(0.01)

# Chose an appropiate nside to represent it
# HealpixBase is a map without data, only the grid is defined
mEq = HealpixBase(order = 14)
```

(continues on next page)

(continued from previous page)

```
# Create a MOC map where the region around the source is
# finely pixelated at the highest order, and the rest of
# the map is left to mhealpy to fill appropiately
disc_pix = mEq.query_disc(hmap.ang2vec(theta0, phi0), 3*sigma)

m = HealpixMap.moc_from_pixels(mEq.nside, disc_pix, density=True)

print("NUNIQ pixels in MOC map: {}".format(m.npix))
print("Equivalent single-resolution pixels: {}".format(mEq.npix))

# Fill the map. This code would look exactly the same if this were a
# single-resolution map
for pix in range(m.npix):

    theta,phi = m.pix2ang(pix)

    m[pix] = exp(-((theta-theta0)**2 + (phi-phi0)**2) / 2 / sigma**2)

# Plot, zooming in around the source
fig = plt.figure(figsize = [14,7])

lonra = np.rad2deg(phi0) + [-.1, .1]
latra = 90 - np.rad2deg(theta0) + [-.1, .1]

axMoll = fig.add_subplot(1,2,1, projection = "mollview")
axCart = fig.add_subplot(1,2,2, projection = "cartview", lonra = lonra, latra = latra)

axMoll.set_title("Full-sky", size = 20)
axCart.set_title("Zoom in", size = 20)

m.plot(axMoll, vmin = 0, vmax = 1);
m.plot(axCart, vmin = 0, vmax = 1);

axCart.set_xlabel("Azimuth angle [deg]")
axCart.set_ylabel("Zenith angle [deg]");

# Show grid
m.plot_grid(axMoll, color='white', linewidth = .2)
m.plot_grid(axCart, color='white', linewidth = .1);
```

```
NUNIQ pixels in MOC map: 372
Equivalent single-resolution pixels: 3221225472
```



The function *moc_from_pixels()* is a convenience routine derived from *adaptive_moc_mesh()*. The same is true for *moc_histogram()* and *to_moc()*. In the more general *adaptive_moc_mesh()* the user provides an arbitrary function that decides, recusively, whether a pixel must be split into child pixels of higher order or remain as a single pixel.

2.5.2 Arithmethic operations

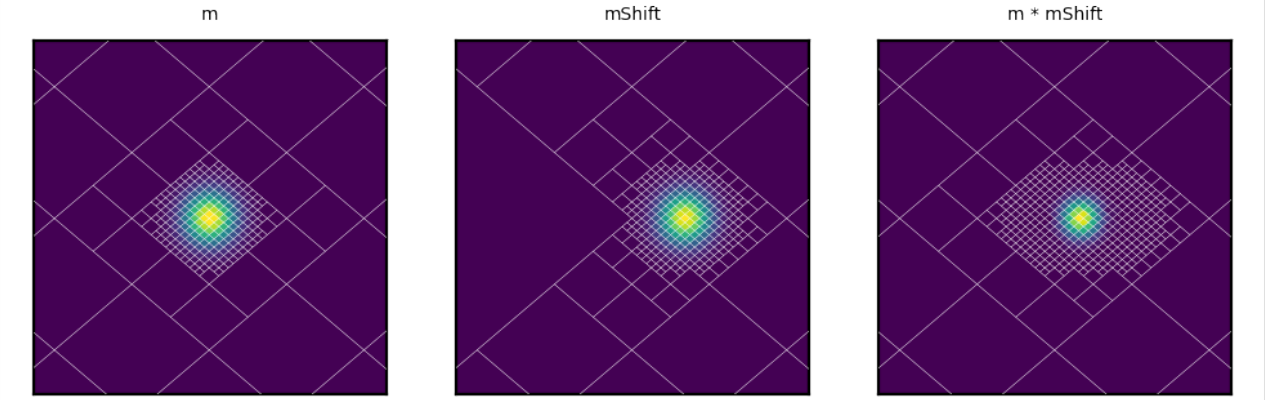
Operations between MOC maps, or a MOC map and a single-resolution map, is the sames as between two single-resolution maps. Something to keep in mind though is that, for binary operations, if any of the two maps is a MOC map, the results will be a MOC map as well. The grid will be a combination that of the two operands. This ensures that there is no loss of information. If you want to keep the same grid, you can use in-place operator (e.g. *=, /=), but the information might degrade in that case.

[15]:
```
# Inject another shifted source in a new map
phi0 = np.deg2rad(45-.03)
disc_pix = mEq.query_disc(hmap.ang2vec(theta0, phi0), 3*sigma)

mShift = HealpixMap.moc_from_pixels(mEq.nside, disc_pix, density=True)

for pix in range(mShift.npix):

    theta,phi = mShift.pix2ang(pix)

    mShift[pix] = exp(-((theta-theta0)**2 + (phi-phi0)**2) / 2 / sigma**2)

#Multiply
mRes = m * mShift

# Plot side by side
fig,axes = plt.subplots(1,3, dpi=200, subplot_kw = {'projection': 'cartview', 'lonra':␣
↪lonra, 'latra': latra})
```

(continues on next page)

(continued from previous page)

```
axes[0].set_title("m", size= 5);
axes[1].set_title("mShift", size= 5);
axes[2].set_title("m * mShift", size= 5);

m.plot(axes[0], cbar = False);
mShift.plot(axes[1], cbar = False);
mRes.plot(axes[2], cbar = False);

# Show grid
m.plot_grid(axes[0], color='white', linewidth = .1)
mShift.plot_grid(axes[1], color='white', linewidth = .1)
mRes.plot_grid(axes[2], color='white', linewidth = .1);
```



CHAPTER

THREE

EXAMPLES

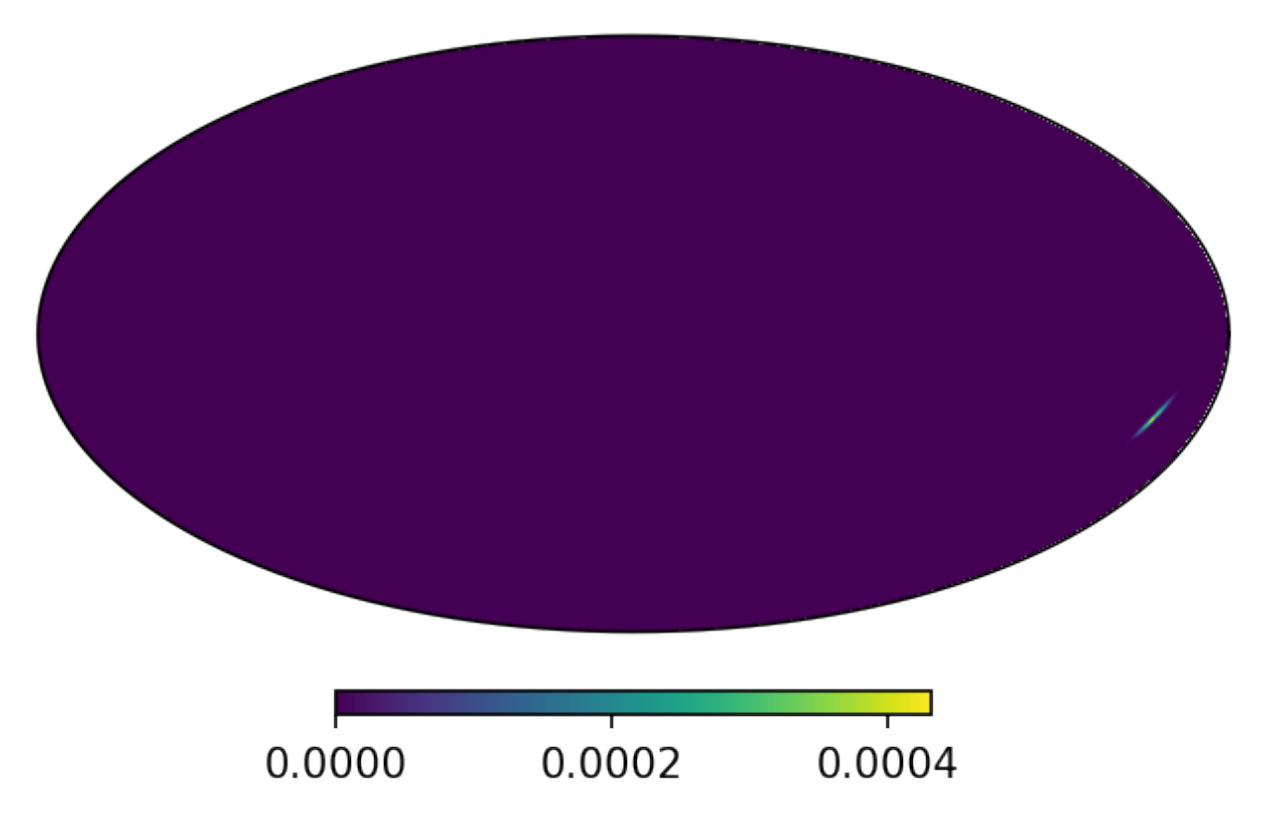
3.1 LIGO/Virgo maps, I/O and resampling

On this example we'll use LIGO/Virgo skymaps as an excuse to see how to open/write a map to/from disc, how create a multi-resolution maps out of a single-resolution map, and how to resample an already multi-resolution map.

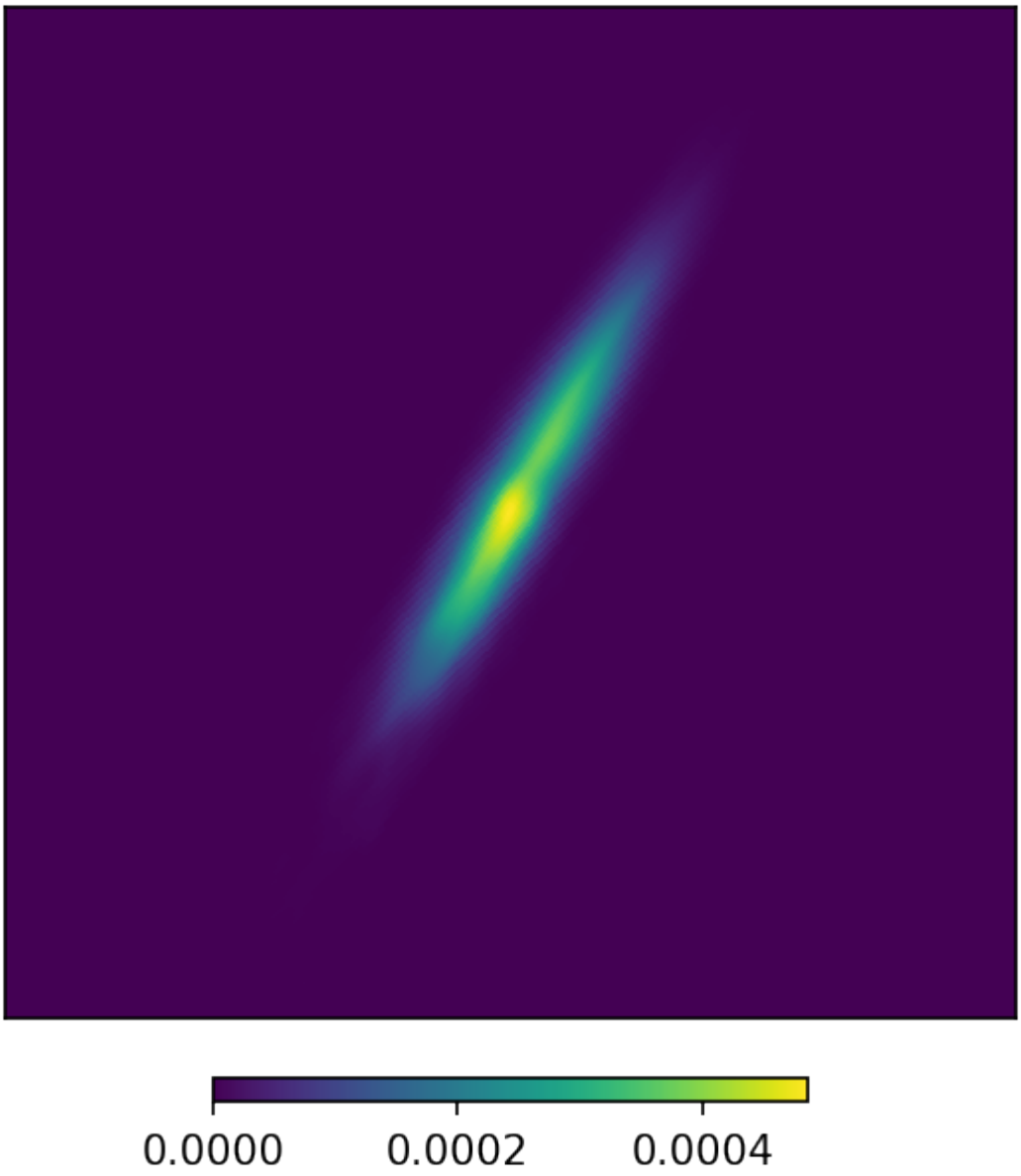
3.1.1 Reading a map

Both single and mult-resolution maps are stored in FITS files. While you can download a map and then provide the local path, you can also use the URL directly, e.g.

[1]:
```
from mhealpy import HealpixMap

# GW170817!
m = HealpixMap.read_map("https://dcc.ligo.org/public/0157/P1800381/007/GW170817_skymap.
↪fits.gz", density = False)

m.plot();
```



[2]:
```
# Zoom in around the maximum

import numpy as np

theta_max,phi_max = m.pix2ang(np.argmax(m))

lonra = np.rad2deg(phi_max) + np.array([-10,10])
latra = (90-np.rad2deg(theta_max)) + np.array([-10,10])

m.plot(ax = 'cartview', ax_kw = {'latra': latra, 'lonra': lonra});
```



3.1.2 Single to multi-resolution

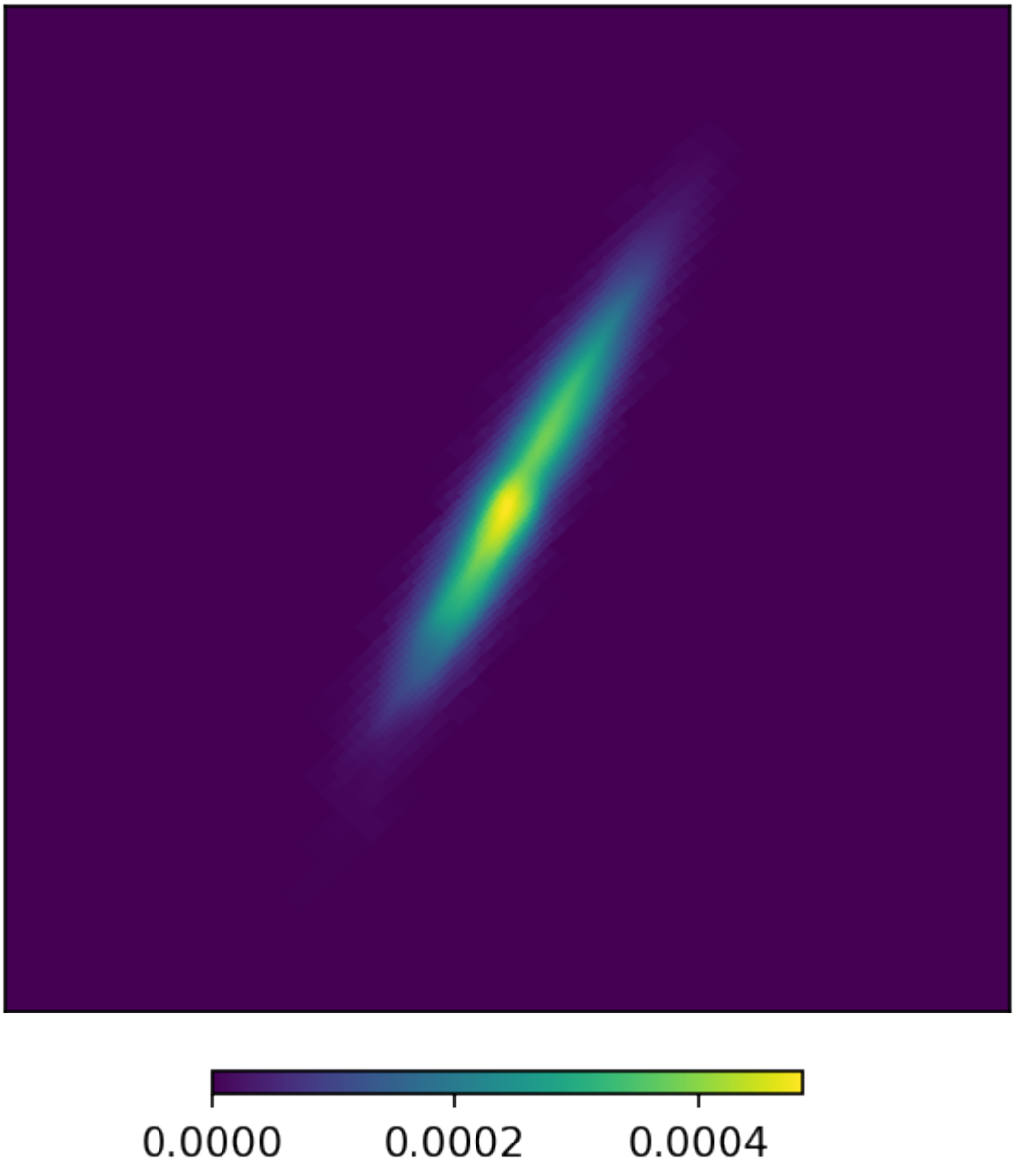
The map we just opened is a single resolution map. Since this is a well-localized event the full sky is sampled at a relatively high resolution.

[3]:
```
print("Is multi-resolution? {}".format(True if m.is_moc else False))
print("nside: {}".format(m.nside))
print("# of pixels: {}".format(m.npix))
```

```
Is multi-resolution? False
nside: 1024
# of pixels: 12582912
```

We can create a multi-resolution map by setting the condition that the probability assigned to any given pixel is at most the maximum value of the current single-resolution map. This mitigates the loss of information and results in a fair sampling for all locations.

[4]:
```
mm = m.to_moc(max_value = max(m))

mm.plot(ax = 'cartview', ax_kw = {'latra': latra, 'lonra': lonra});
```



While this plot looks pretty much the same as before, we reduced the number of pixels by 3 orders to magnitude.

[5]:
```
print("Is multi-resolution? {}".format(True if mm.is_moc else False))
print("nside: {}".format(mm.nside))
print("# of pixels: {}".format(mm.npix))
```

```
Is multi-resolution? True
```

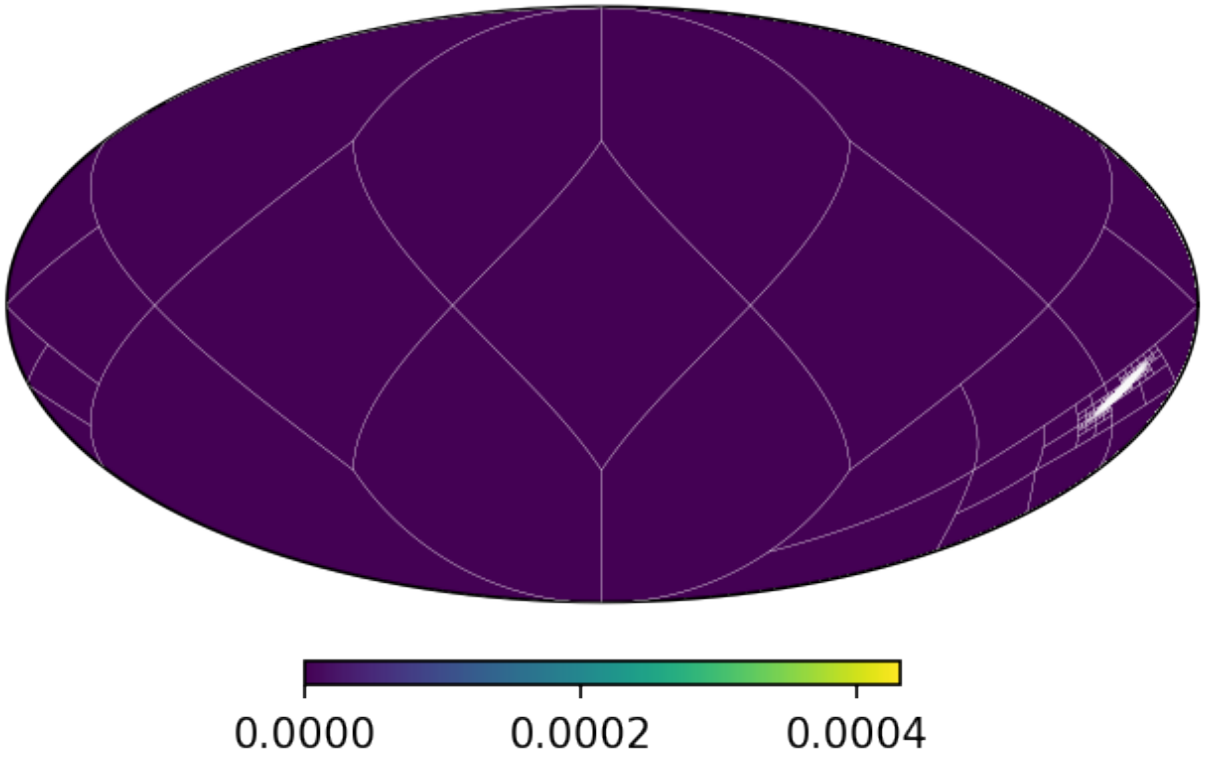
(continues on next page)

(continued from previous page)

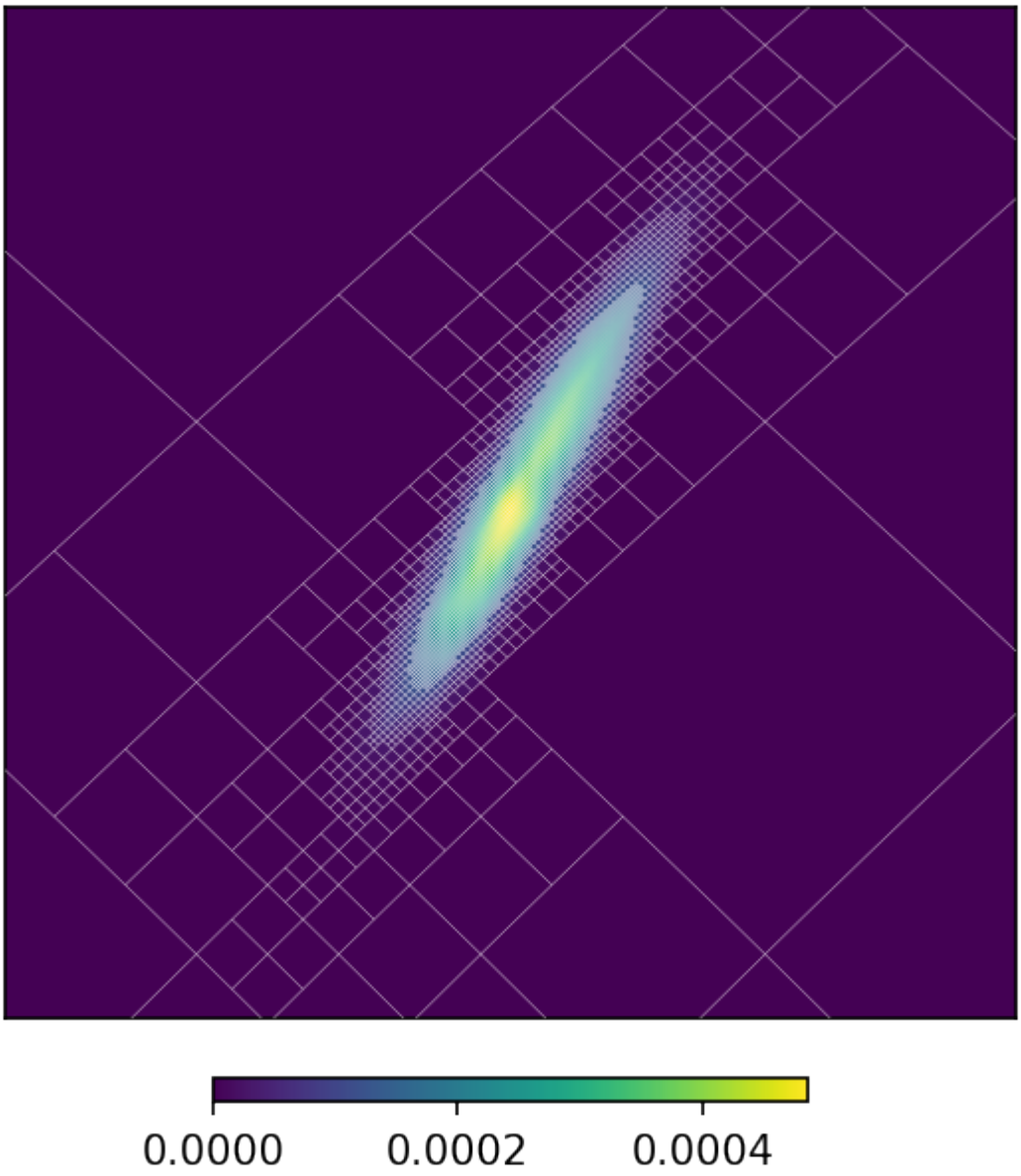
```
nside: 1024
# of pixels: 4026
```

We can see the way this works more clearly by superimposing the grid

[6]:
```
import matplotlib.pyplot as plt

mm.plot()
mm.plot_grid(plt.gca(), linewidth = .1, color = 'white');
```

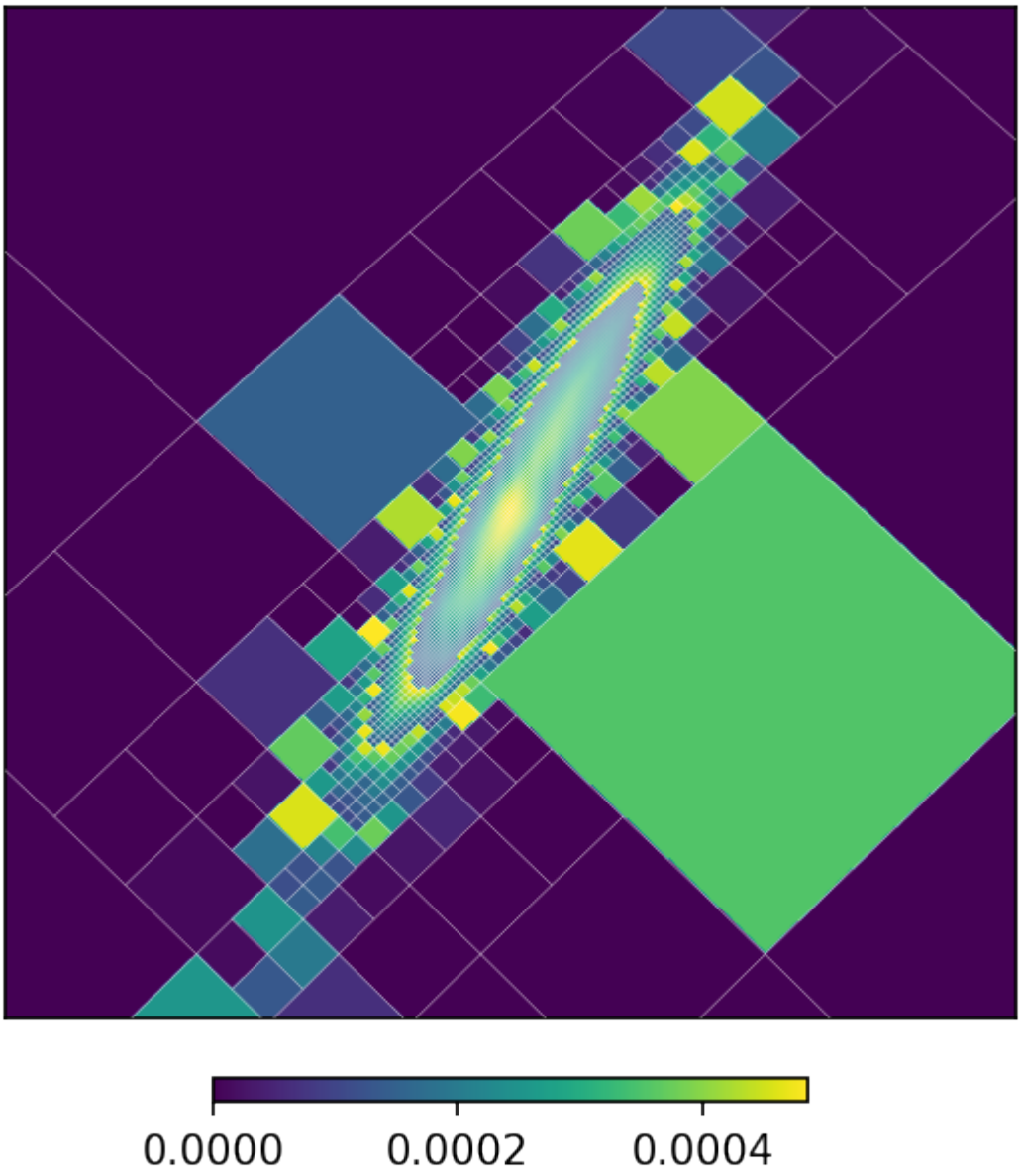


[7]:
```
# Zoom in
mm.plot(ax = 'cartview', ax_kw = {'latra': latra, 'lonra': lonra})
mm.plot_grid(plt.gca(), linewidth = .1, color = 'white');
```



By default the equivalent to the rasterized map is plotted. That is, the previous plot is proportional to the probability *density* distribution, rather than the integrated probability on each pixel. You can changed this behaviour using the `rasterize` option.

[8]:
```
mm.plot(ax = 'cartview', ax_kw = {'latra': latra, 'lonra': lonra}, rasterize = False)
mm.plot_grid(plt.gca(), linewidth = .1, color = 'white');
```



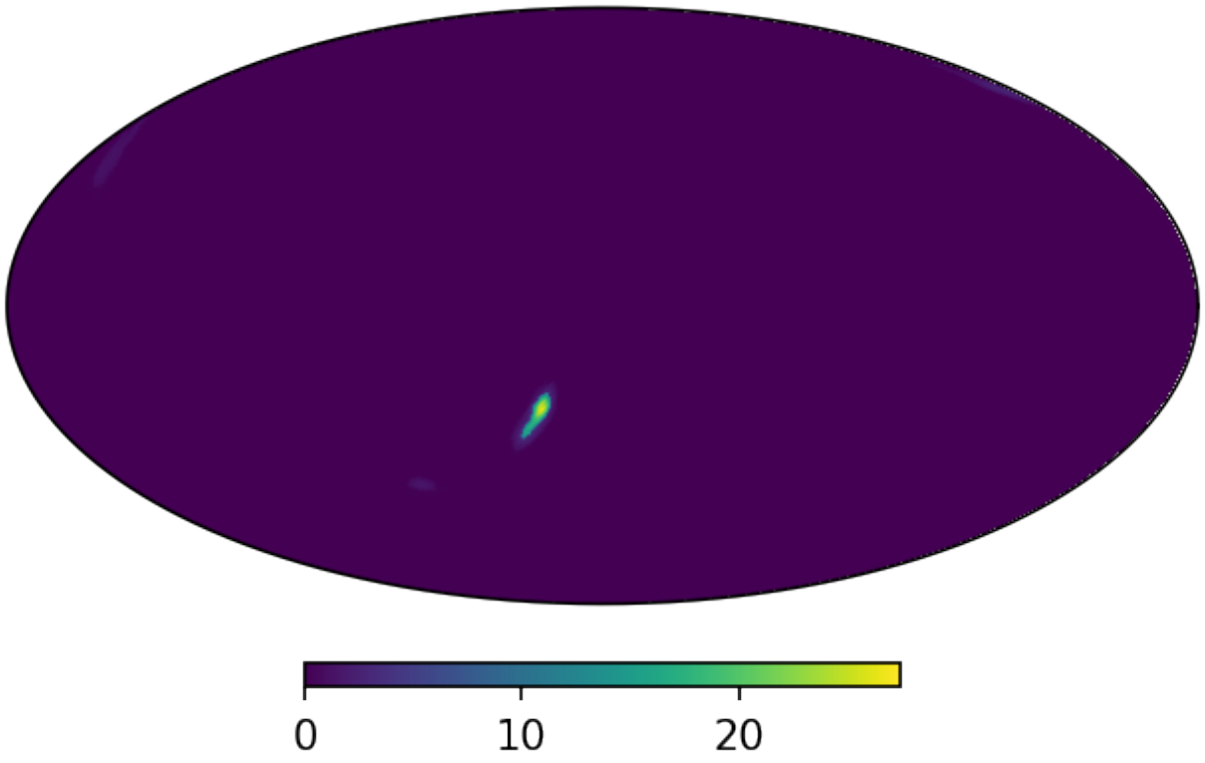
The function *to_moc()* is a convenience routine derived from *adaptive_moc_mesh()*. The same is true for *moc_histogram()* and *moc_from_pixels()*. In the more general *adaptive_moc_mesh()* the user provides an arbitrary function that decides, recusively, whether a pixel must be split into child pixels of higher order or remain as a single pixel.

3.1.3 Resampling multi-resolution maps

For new gravitational wave events LIGO/Virgo also provides multi-resolution maps straight out the box, e.g.

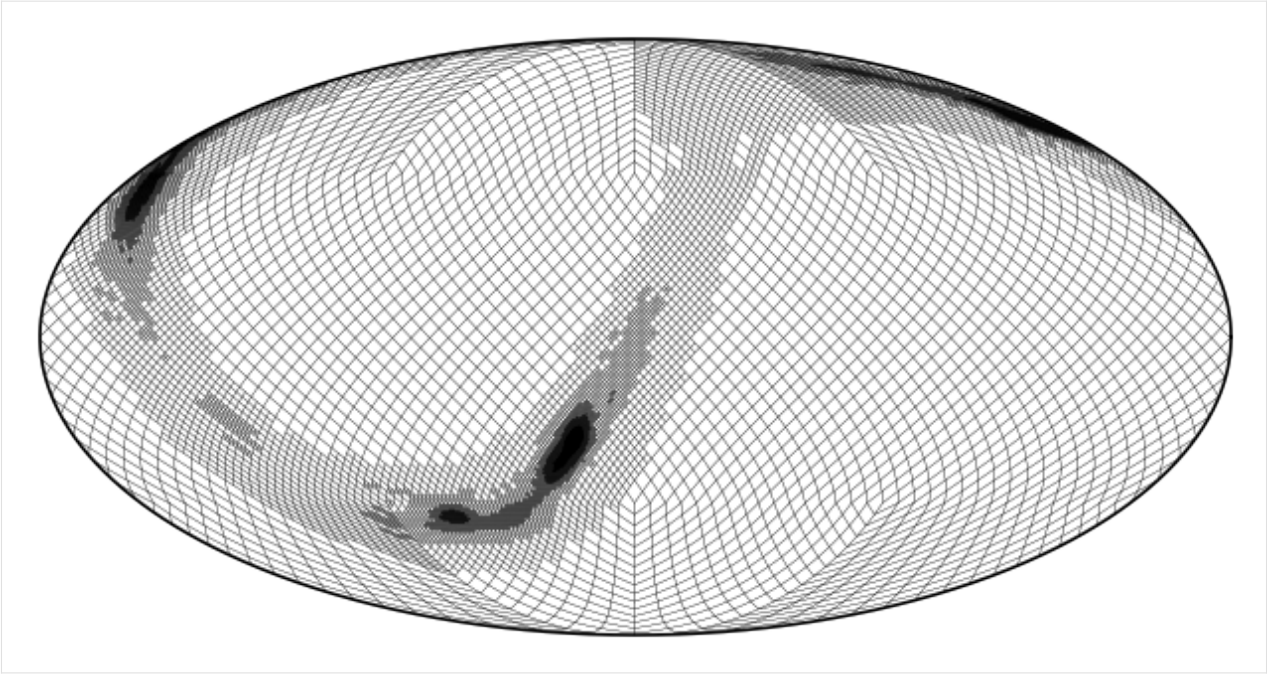
[9]:
```
from mhealpy import HealpixMap

m = HealpixMap.read_map("https://gracedb.ligo.org/api/superevents/S200219ac/files/
↪LALInference.multiorder.fits,0", density = False)

print("Is multi-resolution? {}".format(True if m.is_moc else False))
print("nside: {}".format(m.nside))
print("# of pixels: {}".format(m.npix))

m.plot();
```

```
Is multi-resolution? True
nside: 512
# of pixels: 16896
```



The grid though corresponds to the adaptive mesh used to generate the sky localization:

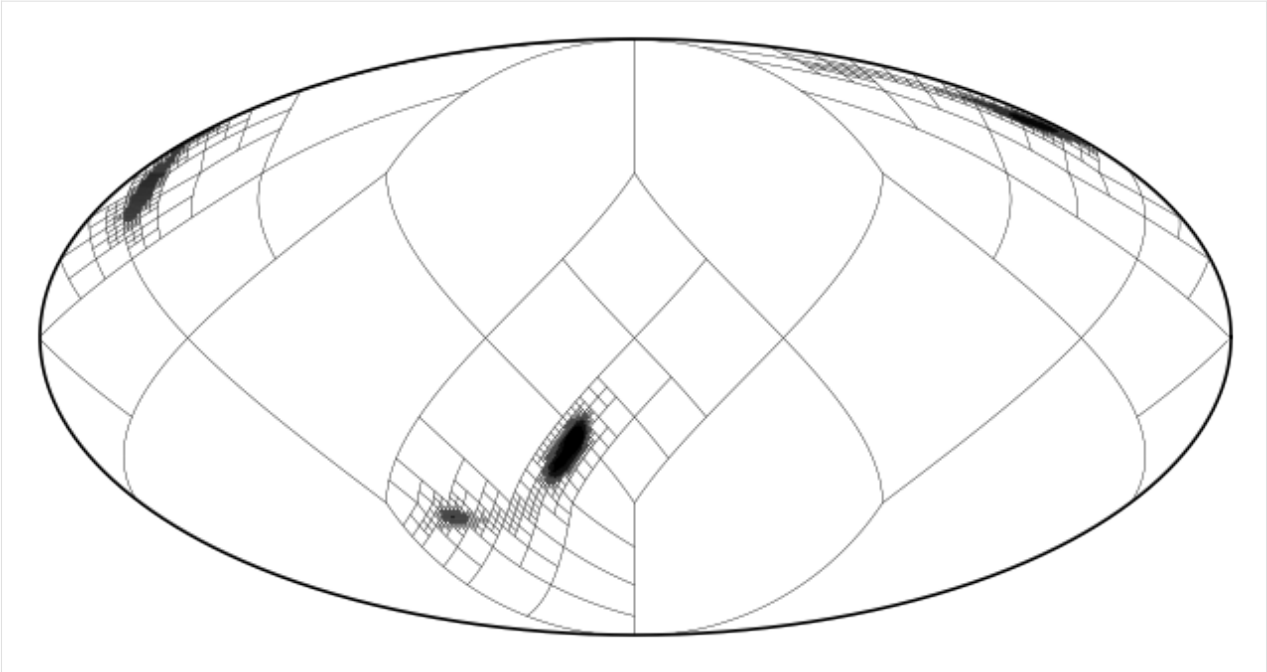
[10]:
```
m.plot_grid(linewidth = .1);
```



We can resample it the same way we did for the single-resolution map:

[11]:
```
mm = m.to_moc(max(m))

print("Is multi-resolution? {}".format(True if mm.is_moc else False))
print("nside: {}".format(mm.nside))
print("# of pixels: {}".format(mm.npix))

mm.plot_grid(linewidth = .1);
```

```
Is multi-resolution? True
nside: 512
# of pixels: 5616
```



While only a modest improvement in this case, resampling can be useful when the map is the product of maps from multiple sources. As seen in the quick start tutorial, in order to assure there is no information loss, the grid resulting from an operation is the union of the grid of its operands. This can result in high resolution in regions where it is no longer needed.

3.1.4 Writing a map to disc

We can now save this resampled map to disc.

[12]:
```
mm.write_map("S200219ac_LALInference_resampled.multiorder.fits")
```

The format is compliant with the the IVOA MOC recommendation. The map is saved into the second (0-th indexed) HDU as an extension table, each pixel specified explicitely by its UNIQ number:

[13]:
```
from astropy.io import fits

f = fits.open("S200219ac_LALInference_resampled.multiorder.fits")

f[1].header
```

[13]:
```
XTENSION= 'BINTABLE'           / binary table extension
BITPIX  =                    8 / array data type
NAXIS   =                    2 / number of array dimensions
NAXIS1  =                   16 / length of dimension 1
NAXIS2  =                 5616 / length of dimension 2
PCOUNT  =                    0 / number of group parameters
GCOUNT  =                    1 / number of groups
TFIELDS =                    2 / number of table fields
TTYPE1  = 'UNIQ    '
TFORM1  = 'K       '
TTYPE2  = 'CONTENTS'
```

(continues on next page)

(continued from previous page)

```
TFORM2  = 'D       '
PIXTYPE = 'HEALPIX '           / HEALPIX pixelisation
ORDERING= 'NUNIQ   '           / Pixel ordering scheme: RING, NESTED, or NUNIQ
COORDSYS= 'C       '           / Celestial (C), Galactic (G) or Ecliptic (E)
NSIDE   =                  512 / Resolution parameter of HEALPIX
INDXSCHM= 'EXPLICIT'           / Indexing: IMPLICIT or EXPLICIT
MOCORDER=                    9 / Best resolution order
```

3.2 IPN annulus map

The Interplanetary Network (IPN) is a group of spacecrafts that localize gamma-ray bursts (GRBs) based on the arrival time of the event at the location of each space mission. This, as other triangulation methods, results in a localization annulus when only two missions detect the GRB.

The following function returns a multi-resolution map that approximately describes this localization probability. The strategy is similar to generating a map for a well-localized source, as shown in the *Quick Start*) tutorial. We'll first generate an empty map with high resolution around the approximate region where we need it. Then we'll evaluate a normal distribution (in the radial coordinate) around the middle of the annulus.

[1]:
```
import mhealpy as mhp
from mhealpy import HealpixMap,HealpixBase
import numpy as np

def get_annulus_map(theta, phi, radius, sigma):
    """
    Obtain a probability distribution map representing the annulus resulting
    from triangulating data from two observers. The annulus is defined
    by the locations of the circle's center, radius and width.

    Args:
        theta: Colatitude of the circle center [rad]
        phi: Longitude if the circle center [rad]
        radius: Angular radius of the circle
        sigma: Circle's width, defined as the standard deviation of a
               radial distribution [rad]

    Return:
        HealpixMap
    """

    # First, get an equivalent single-resolution order, such that the pixel
    # size is smaller than the annulus width

    approx_nside = np.sqrt(4*np.pi/12)/sigma # Pixel size is approximately sqrt(4*pi/12)/
↪nside
    order = int(np.ceil(np.log2(approx_nside))+2)

    mEq = HealpixBase(order = order, scheme = 'nested') # "Empty" map

    # Now, get the pixels around the annulus' main circle, which is the only
```

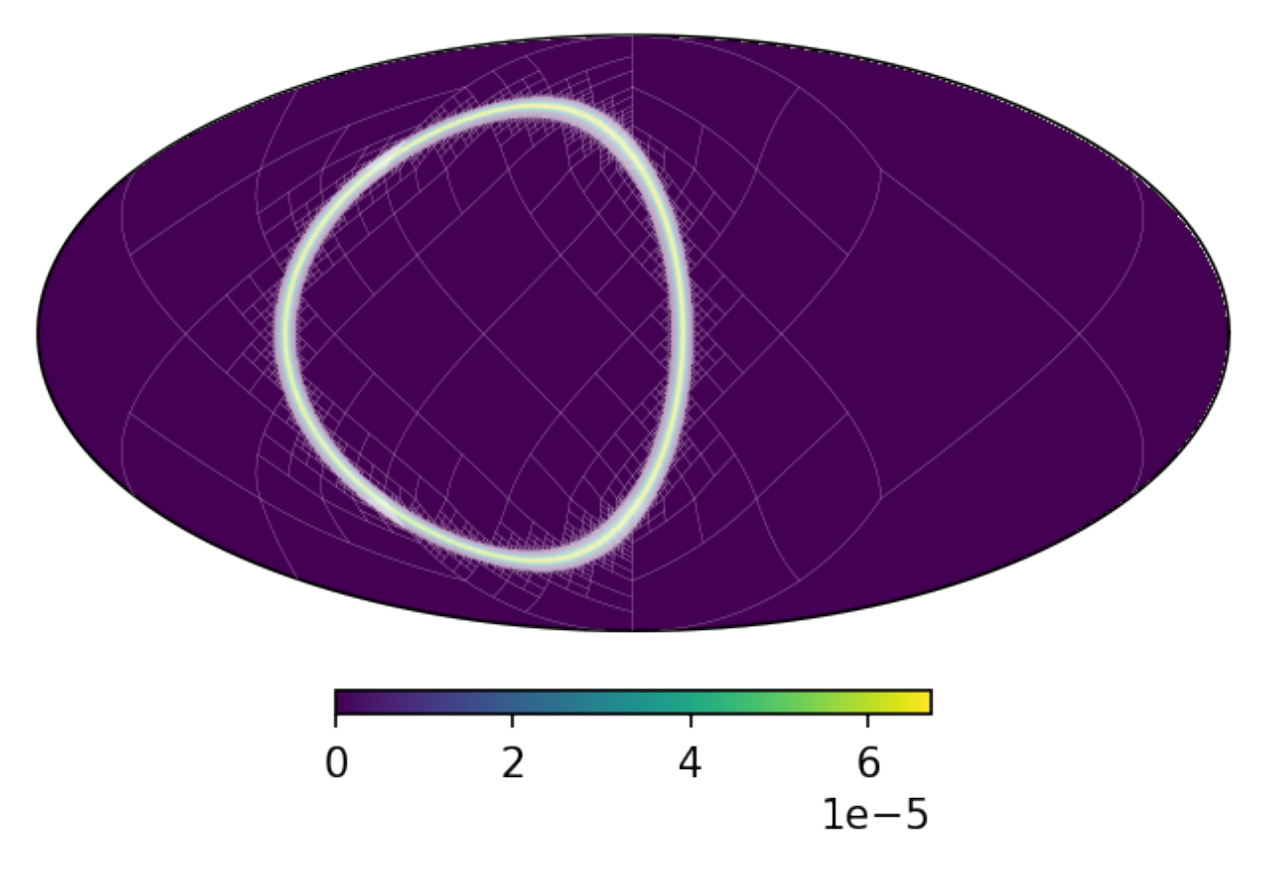
(continues on next page)

(continued from previous page)

```
    # region that needs high resolution. We query the equivalent
    # to 3 standard deviations around.
    # We use query_disc to get the pixels within the outer and inner bounds. The
    # pixels we want are the intersection between these.

    center_vec = mhp.ang2vec(theta, phi)

    outer_disc= mEq.query_disc(center_vec, radius = radius + 3*sigma)
    inner_disc = mEq.query_disc(center_vec, radius = radius - 3*sigma)

    hires_pix = np.setdiff1d(outer_disc, inner_disc)

    # Next, let mhealpy generate the apropiate mesh for a multi-resolution map
    # containing these pixels

    m = HealpixMap.moc_from_pixels(mEq.nside, hires_pix, nest = mEq.is_nested, density =␣
↪False)

    # We then initialize all pixels based on a radial normal distribution

    for pix in range(m.npix):
        pix_vec = m.pix2vec(pix)
        pix_radius = np.arccos(sum(pix_vec*center_vec))
        m[pix] = np.exp(-(pix_radius - radius)**2/2/sigma**2) * m.pixarea(pix).value

    # Finally, normalize probability distribution to 1 and return
    m /= sum(m)

    return m
```

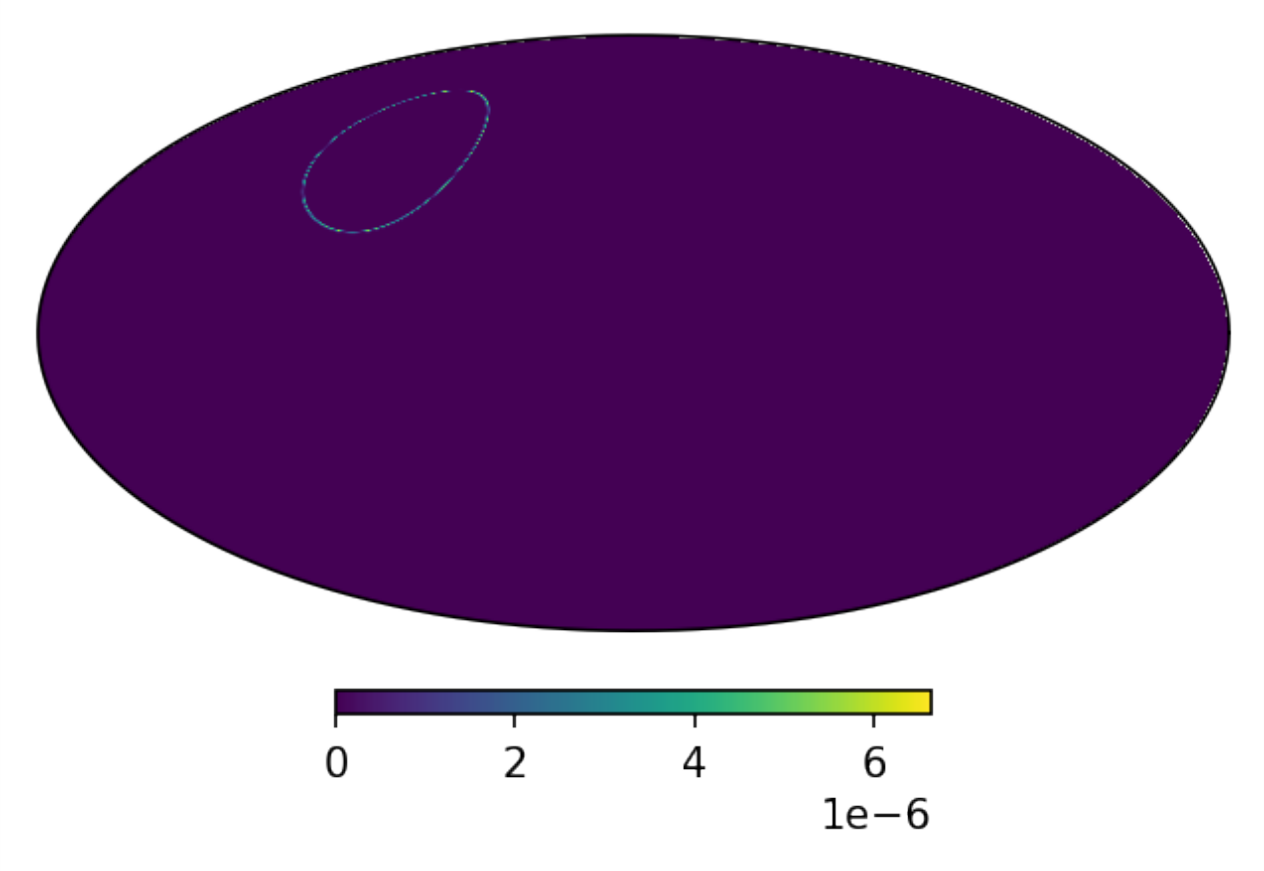
Let's use it to create a map and plot it:

[2]:
```
m0 = get_annulus_map(theta = np.deg2rad(90),
                     phi = np.deg2rad(45),
                     radius = np.deg2rad(60),
                     sigma = np.deg2rad(1))

#Plot
import matplotlib.pyplot as plt

m0.plot()
m0.plot_grid(ax = plt.gca(), linewidth = .1, color = 'white', alpha = .5);
```



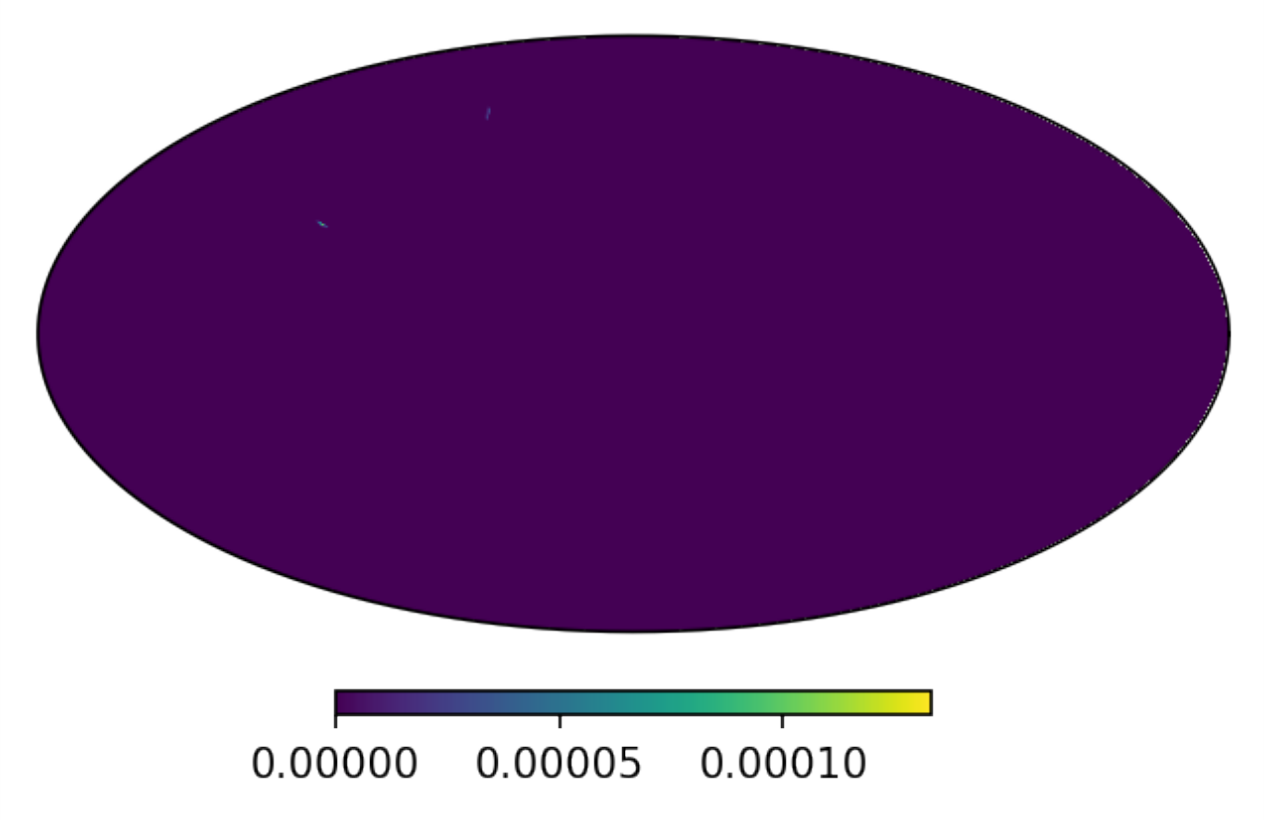
Now, let's assume a third spacecraft detected the event and we have an extra constrain that results in a second annulus

[3]:
```
m1 = get_annulus_map(theta = np.deg2rad(45),
                     phi = np.deg2rad(90),
                     radius = np.deg2rad(20),
                     sigma = np.deg2rad(0.1))

m1.plot();
```



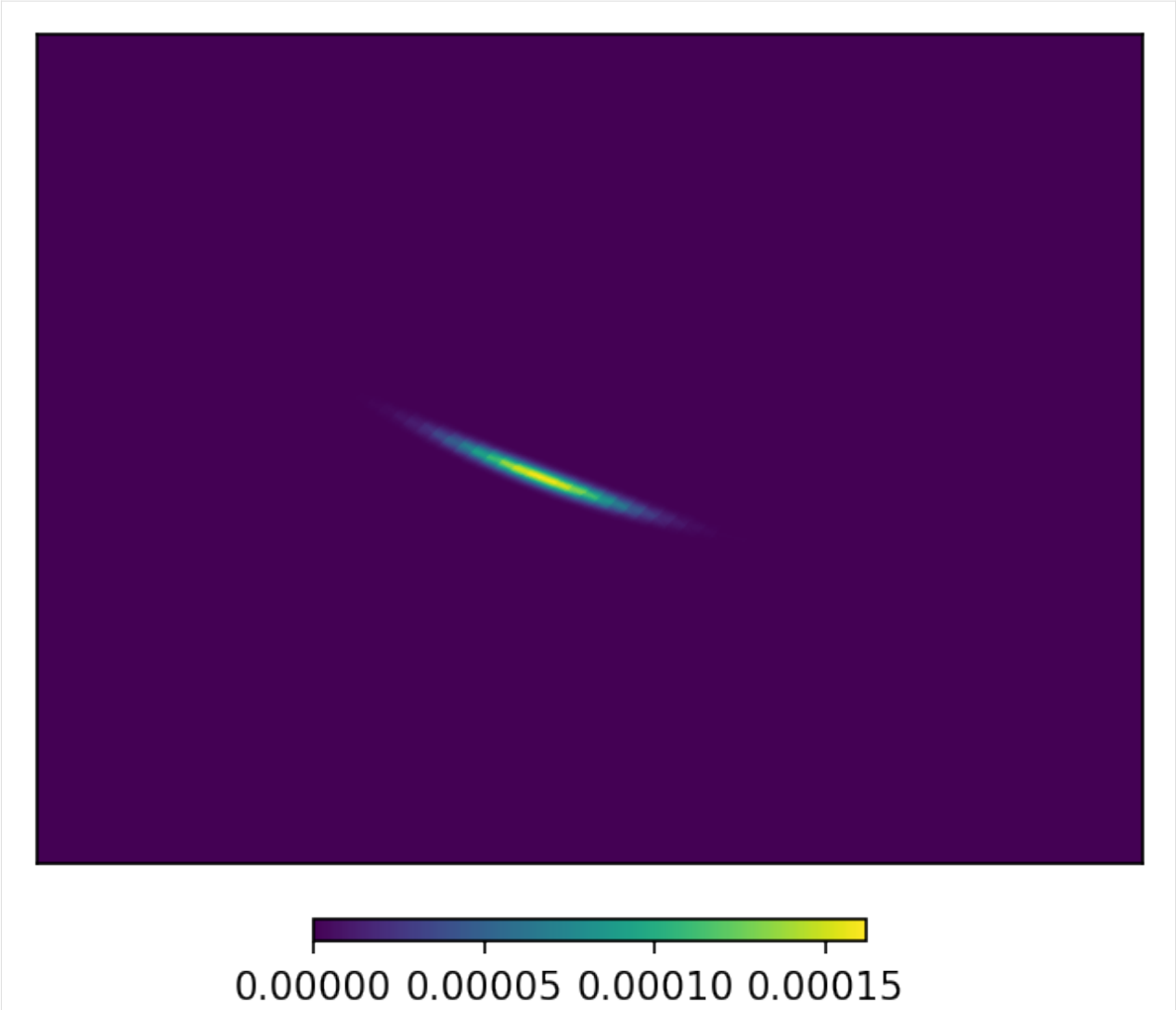
This constrains the source location to approximately two points in the sky

[4]:
```
# This divides the value of each pixel by its area, turning the map into a probability␣
↪density distribution
m0.density(True)
m1.density(True)

mProd = m0*m1

# Return to a probability distribution and normalize
mProd.density(False)
mProd /= sum(mProd)

mProd.plot();
```



We can see the details by zooming into one of them. It has an elongated shape since one of the annuli was much narrower than the other.

[5]:
```
mProd.plot(ax = 'cartview', ax_kw = {'latra':  [20, 35], 'lonra': [90, 110]});
```


0.00000 0.00005 0.00010 0.00015

CHAPTER

FOUR

API

4.1 Classes

4.1.1 HealpixBase

class mhealpy.**HealpixBase**(*uniq=None*, *order=None*, *nside=None*, *npix=None*, *scheme='ring'*, *coordsys=None*, *base=None*)

Bases: `object`

Basic operations related to HEALPix pixelization, for which the map contents information is not needed. This class is conceptually very similar the the Healpix_Base class of Healpix_cxx.

Single resolution maps are fully defined by specifying their order (or NSIDE) and ordering scheme ("RING" or "NESTED").

Multi-resolution maps follow an explicit "NUNIQ" scheme, with each pixel identfied by a _uniq_ number. No specific is needed nor guaranteed.

Warning: The initialization input is not validated by default. Consider calling *is_mesh_valid()* after initialization, otherwise results might be unexpected.

Parameters

- **uniq** (*array*) – Explicit numbering of each pixel in an "NUNIQ" scheme.
- **Order** (*int*) – Order of HEALPix map.
- **nside** (*int*) – Alternatively, you can specify the NSIDE parameter.
- **npix** (*int*) – Alternatively, you can specify the total number of pixels.
- **scheme** (*str*) – Healpix scheme. Either 'RING', 'NESTED' or 'NUNIQ'
- **coordsys** (*BaseFrameRepresentation or str*) – Instrinsic coordinates of the map. Either 'G' (Galactic), 'E' (Ecliptic) , 'C' (Celestial = Equatorial) or any other coordinate frame recognized by astropy.
- **base** (HealpixBase) – Alternatively, you can copy the properties of another HealpixBase object

classmethod adaptive_moc_mesh(*max_nside*, *split_fun*, *coordsys=None*)

Return a MOC mesh with an adaptive resolution determined by an arbitrary function.

Parameters

- **max_nside** (*int*) – Maximum HEALPix nside to consider
- **split_fun** (*function*) – This method should return `True` if a pixel
- **order** (*should be split into pixel of a higher*) –
- **otherwise.** (*and False*) –
- **integers** (*It takes two*) –
- **start** (inclusive) and `stop` (exclusive) –
- **a** (*which correspond to a single pixel in nested rangeset format for*) –
- **max_nside.** (*map of nside*) –
- **coordsys** (*BaseFrameRepresentation or str*) – Assigns a coordinate system to the map

Returns
HealpixBase

classmethod moc_from_pixels(*nside*, *pixels*, *nest=False*, *coordsys=None*)

Return a MOC mesh where a list of pixels are kept at a given nside, and every other pixel is appropiately downsampled.

Also see the more generic `adaptive_moc()` and `adaptive_moc_mesh()`.

Parameters

- **nside** (*int*) – Maximum healpix NSIDE (that is, the NSIDE for the pixel list)
- **pixels** (*array*) – Pixels that must be kept at the finest pixelation
- **nest** (*bool*) – Whether the pixels are a 'NESTED' or 'RING' scheme
- **coordsys** (*BaseFrameRepresentation or str*) – Assigns a coordinate system to the map

conformable(*other*)

For single-resolution maps, return `True` if both maps have the same nside, scheme and coordinate system.

For MOC maps, return *True* if both maps have the same list of UNIQ pixels (including the ordering)

property npix

Get number of pixels.

For multi-resolutions maps, this corresponds to the number of utilized UNIQ pixels.

Returns
int

property order

Get map order

Returns
int

property nside

Get map NSIDE

Returns
int

property scheme

Return HEALPix scheme

Returns
Either 'NESTED', 'RING' or 'NUNIQ'

Return type
str

property is_nested

Return true if scheme is NESTED or NUNIQ

Return
bool

property is_ring

Return true if scheme is RING

Return
bool

property is_moc

Return true if this is a Multi-Dimensional Coverage (MOC) map (multi-resolution)

Returns
bool

pix_rangesets(*nside=None*, *argsort=False*)

Get the equivalent range of *child pixels* in nested scheme for a map of equal or higher nside

Parameters

- **nside** (*int or None*) – Nside of output range sets. If None, the map nside will be used. nside = mhealpy.MAX_NSIDE returns the cached result
- **argsort** (*bool*) – Also return also the indices that would sort the array.

Returns

With columns named 'start' (inclusive) and
'stop' (exclusive)

Return type
recarray

pix_order_list()

Get a list of lists containing all pixels sorted by order

Returns

(pix_per_order, nest_pix_per_order)
Each list has a size equal to the map order. Each element is a list of all pixels whose order matches the index of the list position. The first output contains the index of the pixels, while the second contains their coresponding pixel number in a nested scheme.

Return type
(list, list)

pix2range(*nside*, *pix*)

Get the equivalent range of *child pixels* in nested scheme for a map of equal or higher nside

Parameters

- **nside** (*int*) – Nside of output range sets
- **pix** (*int or array*) – Pixel numbers

Returns

Start pixel (inclusive) and
stop pixel (exclusive)

Return type
(int or array, int or array)

pixarea(*pix=None*)

Return area of a pixel

Parameters
pix (*int or array*) – Pixel number. Only relevant for MOC maps. Default: All pixels for MOC, a single value for single resolution

Returns
Quantity

pix2ang(*pix*, *lonlat=False*)

Return the coordinates of the center of a pixel

Parameters
pix (*int or array*) –

Returns
(float or array, float or array)

pix2vec(*pix*)

Return a vector corresponding to the center of a pixel

Parameters
pix (*int or array*) –

Returns
Size (3,N)

Return type
array

pix2skycoord(*pix*)

Return the sky coordinate for the center of a given pixel

Parameters
pix (*int or array*) – Pixel number

Returns
SkyCoord

ang2pix(*theta*, *phi=None*, *lonlat=False*)

Get the pixel (as used in []) that contains a given coordinate

Parameters

- **theta** (*float, array or SkyCoord*) – Zenith angle
- **phi** (*float or arrray*) – Azimuth angle

Returns
int or array

vec2pix(*x*, *y*, *z*)

Get the pixel (as used in []) that contains a given coordinate

Parameters

- **x** (*float or array*) – x coordinate
- **y** (*float or array*) – y coordinate
- **z** (*float or array*) – z coordinate

Returns
int or array

pix2uniq(*pix*)

Get the UNIQ representation of a given pixel index.

Parameters
pix (*int*) – Pixel number in the current scheme (as used for [])

property uniq

Get an array with the NUNIQ numbers for all pixels

nest2pix(*pix*)

Get the corresponding pixel in the current grid for a pixel in NESTED scheme. For MOC map, return the pixel that contains it.

Parameters
pix (*int or array*) – Pixel number in NESTED scheme. Must correspond to a map of the same order as the current.

Returns
int or array

get_interp_weights(*theta*, *phi=None*, *lonlat=False*)

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

Parameters

- **theta** (*float or array*) – Zenith angle (rad)
- **phi** (*float or array*) – Azimuth angle (rad)

Returns

(pixels, weights), each with of (4,) if the input is scalar,
if (4,N) where N is size of theta and phi. For MOC maps, these pixel numbers might repeate.

Return type
tuple

get_all_neighbours(*theta*, *phi=None*, *lonlat=False*)

Return the 8 nearest pixels. For MOC maps, these might repeat, as this is equivalent to raterizing the maps to the highest order, getting the neighbohrs, and then finding the pixels tha contain them.

Parameters

- **theta** (*float or int or array*) – Zenith angle (rad). If phi is None, these are assumed to be pixels numbers.
- **phi** (*float or array or None*) – Azimuth angle (rad)

Returns

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

Return type
array

`is_mesh_valid()`

Return `True` if the map pixelization is valid. For single resolution this simply checks that the size is a valid NSIDE value. For MOC maps, it checks that every point in the sphere is covered by one and only one pixel.

Returns
True

`query_polygon`(*vertices*, *inclusive=False*, *fact=4*)

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

Parameters

- **vertices** (*float*) – Vertex array containing the vertices of the polygon, shape (N, 3).
- **inclusive** (*bool*) – f False, return the exact set of pixels whose pixels centers lie within the region; if True, return all pixels that overlap with the region.
- **fact** (*int*) – Only used when inclusive=True. The overlapping test will be done at the resolution fact*nside. For NESTED ordering, fact must be a power of 2, less than 2**30, else it can be any positive integer. Default: 4.

Returns
The pixels which lie within the given polygon.

Return type
int array

`query_disc`(*vec*, *radius*, *inclusive=False*, *fact=4*)

Parameters

- **vec** (*float, sequence of 3 elements, SkyCoord*) – The coordinates of unit vector defining the disk center.
- **radius** (*float*) – The radius (in radians) of the disk
- **inclusive** (*bool*) – f False, return the exact set of pixels whose pixels centers lie within the region; if True, return all pixels that overlap with the region.
- **fact** (*int*) – Only used when inclusive=True. The overlapping test will be done at the resolution fact*nside. For NESTED ordering, fact must be a power of 2, less than 2**30, else it can be any positive integer. Default: 4.

Returns
The pixels which lie within the given disc.

Return type
int array

`query_strip`(*theta1*, *theta2*, *inclusive=False*)

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

Parameters

- **theta** (*float*) – First colatitude (radians)
- **phi** (*float*) – Second colatitude (radians)
- **inclusive** (*bool*) – f False, return the exact set of pixels whose pixels centers lie within the region; if True, return all pixels that overlap with the region.

Returns

The pixels which lie within the given strip.

Return type

int array

`boundaries`(*pix*, *step=1*)

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

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

`plot_grid`(*ax='mollview'*, *ax_kw={}*, *step=32*, *coord=None*, ***kwargs*)

Plot the pixel boundaries of a Healpix grid

Parameters

- **ax** (*WCSAxes or str*) – Astropy's WCSAxes to plot the map. Either an existing instance or the name of a registered projection.
- **ax_kw** (*dict*) – Extra arguments if a new axes needs to be created.
- **step** (*int*) – How many points per pixel side
- **coord** (*str*) – Instrinsic coordinates of the map. Either 'G' (Galactic), 'E' (Ecliptic) , 'C' (Celestial = Equatorial) or any other coordinate frame recognized by astropy. The default is 'C' unless `coordsys` is defined. This option overrides `coordsys`
- ****kwargs** – Passed to matplotlib.pyplot.plot()

Returns

The first return value

corresponds to the output `pyplot.plot()` for one of the pixels. The second is the astropy WCSAxes object used.

Return type

matplotlib.lines.Line2D list, WCSAxes

`moc_sort`()

Sort the uniq pixels composing a MOC map based on its rangeset representation

4.1.2 HealpixMap

class mhealpy.**HealpixMap**(*data=None*, *uniq=None*, *order=None*, *nside=None*, *scheme='ring'*, *base=None*, *density=False*, *dtype=None*, *coordsys=None*, *unit=None*)

Bases: *HealpixBase*

Object-oriented healpy wrapper with support for multi-resolutions maps (known as multi-order coverage map, or MOC).

You can instantiate a map by providing either:

- Size (through `order` or `nside`), and a `scheme` ('RING' or 'NESTED'). This will initialize an empty map.
- A list of UNIQ pixels. This will initialize a MOC map. Providing the values for each pixel is optional, zero-initialized by default.
- An array (in `data`) and an a scheme ('RING' or 'NESTED'). This will initialize the contents of the single-resolution map.
- A HealpixBase object. The data will be zero-initialized.

Warning: The initialization input is not validated by default. Consider calling *is_mesh_valid()* after initialization, otherwise results might be unexpected.

Regardless of the underlaying grid, you can operate on maps using `*`, `/`, `+`, `-`, `**`, `==` and `abs`. For binary operations the result always corresponds to the finest grid, so there is no loss of information. If any of the operands is a MOC, the result is a MOC with an appropiate updated grid.. If both operands have the same NSIDE, the scheme of the result corresponds to the left operand. If you want to preserve the grid for a specific operand, use *=, /=, etc.

The result of most binary operations have the same density parameter as the left-most operand, except the following cases: 1) the product of a density-like maps with a histogram-like results in histogram-like map. 2) the ratio between two histogram-like maps is a density-like map. Operations with scalars leave the map type unchanged.

Warning: Information might degrade if you use in-place operators (e.g. *=, /=)

The maps are array-like, that is, the can be casted into a regular numpy array (as used by healpy), are iterable (over the pixel values) and can be used with built-in function such as `sum` and `max`.

You can also access the value of pixels using regular numpy indexing with `[]`. For MOC maps, no specific pixel ordering is guaranted. For a given pixel number `ipix` in the current grid, you can get the corresponding UNIQ pixel number using `m.pix2uniq(ipix)`.

Parameters

- **data** (*array*) – Values to initialize map. Zero-initialized it not provided. The map NSIDE is deduced from the array size, unless `uniq` is specified in which case this is considered a multi-resolution map.
- **uniq** (*array or* HealpixBase) – List of NUNIQ pixel number to initialize a MOC map.
- **order** (*int*) – Order of HEALPix map.
- **nside** (*int*) – Alternatively, you can specify the NSIDE parameter.
- **scheme** (*str*) – Healpix scheme. Either 'RING', 'NESTED' or 'NUNIQ'
- **base** (HealpixBase) – Specify the grid using a HealpixBase object

- **density** (*bool*) – Whether the value of each pixel should be treated as counts in a histogram (`False`) or as the value of a [density] function evaluated at the center of the pixel (`True`). This affect operations involving the splitting of a pixel.
- **dtype** (*array*) – Numpy data type. Will be ignored if data is provided.
- **coordsys** (*BaseFrameRepresentation or str*) – Instrinsic coordinates of the map. Either 'G' (Galactic), 'E' (Ecliptic) , 'C' (Celestial = Equatorial) or any other coordinate frame recognized by astropy.

to(*unit*, *equivalencies=[]*, *update=True*, *copy=True*)

Return a map with converted units

Parameters

- **unit** (*unit-like*) – Unit to convert to.
- **equivalencies** (*list or tuple*) – A list of equivalence pairs to try if the units are not directly convertible.
- **update** (*bool*) – If `update` is `False`, only the units will be changed without updating the contents accordingly
- **copy** (*bool*) – If True (default), then the value is copied. Otherwise, a copy will only be made if necessary.

classmethod read_map(*filename*, *field=None*, *uniq_field=0*, *hdu=1*, *density=False*)

Read a HEALPix map from a FITS file.

Parameters

- **filename** (*Path*) – Path to file
- **field** (*int*) – Column where the map contents are. Default: 0 for single-resolution maps, 1 for MOC maps.
- **uniq_field** (*int*) – Column where the UNIQ pixel numbers are. For MOC maps only.
- **hdu** (*int*) – The header number to look at. Starts at 0.
- **density** (*bool*) – Whether this is a histogram-like or a density-like map.

Returns

HealpixMap

get_fits_hdu(*extra_maps=None*, *column_names=None*)

Build HDU needed to store map in a FITS file

Parameters

- **extra_maps** (HealpixMap *or array*) – Save more maps in the same file as extra columns. Must be conformable.
- **column_names** (*str or array*) – Name of colums. Must have the same length as the number for maps. Defaults to 'CONTENTSn', where `n` is the map number (ommited for a single map). For MOC maps, the pixel information is always stored in the first column, called 'UNIQ'.

Returns

astropy.io.fits.BinTableHDU

write_map(*filename*, *extra_maps=None*, *column_names=None*, *extra_header=None*, *overwrite=False*)

Write map to disc.

Parameters

- **filename** (*Path*) – Path to output file
- **extra_maps** (*HealpixMap or array*) – Save more maps in the same file as extra columns. Must be conformable.
- **column_names** (*str or array*) – Name of colums. Must have the same length as the number for maps. Defaults to 'CONTENTSn', where n is the map number (ommited for a single map). For MOC maps, the pixel information is always stored in the first column, called 'UNIQ'.
- **extra_header** (*iterable*) – Iterable of (keyword, value, [comment]) tuples
- **overwrite** (*bool*) – If True, overwrite the output file if it exists. Raises an OSError if False and the output file exists.

classmethod adaptive_moc_mesh(*max_nside*, *split_fun*, *density=False*, *dtype=None*, *coordsys=None*, *unit=None*)

Return a zero-initialized MOC map, with an adaptive resolution determined by an arbitrary function.

Parameters

- **max_nside** (*int*) – Maximum HEALPix nside to consider
- **split_fun** (*function*) – This method should return True if a pixel
- **order** (*should be split into pixel of a higher*) –
- **otherwise.** (*and False*) –
- **integers** (*It takes two*) –
- **start** (inclusive) and stop (exclusive) –
- **a** (*which correspond to a single pixel in nested rangeset format for*) –
- **max_nside.** (*map of nside*) –
- **density** (*bool*) – Will be pass to HealpixMap initialization.
- **dtype** (*dtype*) – Data type
- **coordsys** (*BaseFrameRepresentation or str*) – Assigns a coordinate system to the map

Returns

HealpixMap

classmethod moc_from_pixels(*nside*, *pixels*, *nest=False*, *density=False*, *dtype=None*, *coordsys=None*, *unit=None*)

Return a zero-initialize MOC map where a list of pixels are kept at a given nside, and every other pixel is appropiately downsampled.

Also see the more generic adaptive_moc_mesh().

Parameters

- **nside** (*int*) – Maximum healpix NSIDE (that is, the NSIDE for the pixel order list)
- **pixels** (*array*) – Pixels that must be kept at the finest pixelation
- **nest** (*bool*) – Whether the pixels are a 'NESTED' or 'RING' scheme
- **density** (*bool*) – Wheather the map is density-like or histogram-like
- **dtype** – Daty type

- **coordsys** (*BaseFrameRepresentation or str*) – Assigns a coordinate system to the map

classmethod moc_histogram(*nside*, *samples*, *max_value*, *nest=False*, *weights=None*, *coordsys=None*, *unit=None*)

Generate an adaptive MOC map by histogramming samples.

If the number of samples is greater than the number of pixels in a map of the input `nside`, consider generating a single-resolution map and then use *to_moc()*.

Also see the more generic `adaptive_moc_mesh()`.

Parameters

- **nside** (*int*) – Healpix NSIDE of the samples and maximum NSIDE of the output map
- **samples** (*int array*) – List of pixels representing the samples. e.g. the output of *healpy.ang2pix()*.
- **max_value** – maximum number of samples (or sum of weights) per pixel. Note that due to limitations of the input `nside`, the output could contain pixels with a value largen than this
- **nest** (*bool*) – Whether the samples are in NESTED or RING scheme
- **weights** (*array*) – Optionally weight the samples. Both must have the same size.
- **coordsys** (*BaseFrameRepresentation or str*) – Assigns a coordinate system to the map

Returns
HealpixMap

to_moc(*max_value*)

Convert a single-resolution map into a MOC based on the maximum value a given pixel the latter should have.

Note: The maximum nside of the MOC map is the same as the nside of the single-resolution map, so the output map could contain pixels with a value greater than this.

If the map is already a MOC map, it will recompute the grid accordingly by combining uniq pixels. Uniq pixels are never split.

Also see the more generic `adaptive_moc_mesh()`.

Parameters
max_value – Maximum value per pixel of the MOC. Whether the map is histogram-like or density-like is taken into account.

Returns
HealpixMap

density(*density=None*, *update=True*)

Switch between a density-like map and a histogram-like map.

Parameters

- **density** (*bool or None*) – Whether the value of each pixel should be treated as counts in a histogram (`False`) or as the value of a [density] function evaluated at the center of the pixel (`True`). This affect operations involving the splitting of a pixel. `None` will leave this paramter unchanged.

- **update** (*bool*) – If True, the values of the map will be updated accordingly. Otherwise only the density parameter is changed.

Note: The `updat=True` the pixels values are divided/multiplied by their effective number of pixels rather than their solid angle area. In order to achieve this scale the map by *1/m.pixarea()*

Returns
The current density

Return type
bool

property data

Get the raw data in the form of an array.

rasterize(*nside=None*, *scheme='ring'*, *uniq=None*, *order=None*, *npix=None*, *base=None*)

Convert to map of a given NSIDE and scheme, or any arbitrary MOC mesh.

Parameters

- **uniq** (*array*) – Explicit numbering of each pixel in an "NUNIQ" scheme.
- **Order** (*int*) – Order of HEALPix map.
- **nside** (*int*) – Alternatively, you can specify the NSIDE parameter.
- **npix** (*int*) – Alternatively, you can specify the total number of pixels.
- **scheme** (*str*) – Healpix scheme. Either 'RING', 'NESTED' or 'NUNIQ'
- **base** (`HealpixBase`) – Alternatively, you can copy the properties of another HealpixBase object

Returns
HealpixMap

get_wcs_img(*wcs*, *coord=None*, *rasterize=True*)

Rasterize map into a set of WCS axes.

Parameters

- **wcs** (*WCS or WCSAxes*) – Astropy's WCSAxes to plot the map. Either an existing instance or the name of a registered projection.
- **coord** (*str*) – Instrinsic coordinates of the map. Either 'G' (Galactic), 'E' (Ecliptic) , 'C' (Celestial = Equatorial) or any other coordinate frame recognized by astropy. The default is 'C' unless `coordsys` is defined. This option overrides `coordsys`
- **rasterize** (*bool*) – If True, the resulting image is equivalent to having called `rasterize()` before plotting. This only affects multi-resolution histogram-like maps.

Returns
array

plot(*ax='mollview'*, *ax_kw={}*, *rasterize=True*, *coord=None*, *cbar=True*, ***kwargs*)

Plot map. This is a wrapper for matplotlib.pyplot.imshow

Parameters

- **ax** (*WCSAxes or str*) – Astropy's WCSAxes to plot the map. Either an existing instance or the name of a registered projection.

- **ax_kw** (*dict*) – Extra arguments if a new axes needs to be created.
- **rasterize** (*bool*) – If True, the resulting image is equivalent to having called `rasterize()` before plotting. This only affects multi-resolution histogram-like maps.
- **coord** (*str*) – Instrinsic coordinates of the map. Either 'G' (Galactic), 'E' (Ecliptic) , 'C' (Celestial = Equatorial) or any other coordinate frame recognized by astropy. The default is 'C' unless `coordsys` is defined. This option overrides `coordsys`
- **cbar** (*bool*) – Whether to plot the colorbar.
- ****kwargs** – Passed to matplotlib.pyplot.imshow

Returns

The first return value
corresponds to the output `imgshow`. The second is the astropy WCSAxes object used.

Return type
AxesImage, WCSAxes

ang2pix(*theta*, *phi=None*, *lonlat=False*)

Get the pixel (as used in []) that contains a given coordinate

Parameters

- **theta** (*float, array or SkyCoord*) – Zenith angle
- **phi** (*float or arrray*) – Azimuth angle

Returns
int or array

boundaries(*pix*, *step=1*)

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

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

conformable(*other*)

For single-resolution maps, return `True` if both maps have the same nside, scheme and coordinate system.

For MOC maps, return *True* if both maps have the same list of UNIQ pixels (including the ordering)

get_all_neighbours(*theta*, *phi=None*, *lonlat=False*)

Return the 8 nearest pixels. For MOC maps, these might repeat, as this is equivalent to raterizing the maps to the highest order, getting the neighbohrs, and then finding the pixels tha contain them.

Parameters

- **theta** (*float or int or array*) – Zenith angle (rad). If phi is `None`, these are assumed to be pixels numbers.
- **phi** (*float or array or None*) – Azimuth angle (rad)

Returns

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

Return type
array

get_interp_val(*theta*, *phi=None*, *lonlat=False*)

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

For MOC maps, this is equivalent to raterizing the map first to the highest order.

Parameters

- **theta** (*float, array or SkyCoord*) – Zenith angle (rad)
- **phi** (*float or array*) – Azimuth angle (rad)

Returns
scalar or array

get_interp_weights(*theta*, *phi=None*, *lonlat=False*)

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

Parameters

- **theta** (*float or array*) – Zenith angle (rad)
- **phi** (*float or array*) – Azimuth angle (rad)

Returns

(pixels, weights), each with of (4,) if the input is scalar,
if (4,N) where N is size of theta and phi. For MOC maps, these pixel numbers might repeate.

Return type
tuple

is_mesh_valid()

Return `True` if the map pixelization is valid. For single resolution this simply checks that the size is a valid NSIDE value. For MOC maps, it checks that every point in the sphere is covered by one and only one pixel.

Returns
True

property is_moc

Return true if this is a Multi-Dimensional Coverage (MOC) map (multi-resolution)

Returns
bool

property is_nested

Return true if scheme is NESTED or NUNIQ

Return
bool

property is_ring

Return true if scheme is RING

Return
bool

nest2pix(*pix*)

Get the corresponding pixel in the current grid for a pixel in NESTED scheme. For MOC map, return the pixel that contains it.

Parameters
pix (*int or array*) – Pixel number in NESTED scheme. Must correspond to a map of the same order as the current.

Returns
int or array

property npix

Get number of pixels.

For multi-resolutions maps, this corresponds to the number of utilized UNIQ pixels.

Returns
int

property nside

Get map NSIDE

Returns
int

property order

Get map order

Returns
int

pix2ang(*pix*, *lonlat=False*)

Return the coordinates of the center of a pixel

Parameters
pix (*int or array*) –

Returns
(float or array, float or array)

pix2range(*nside*, *pix*)

Get the equivalent range of *child pixels* in nested scheme for a map of equal or higher nside

Parameters

- **nside** (*int*) – Nside of output range sets
- **pix** (*int or array*) – Pixel numbers

Returns

Start pixel (inclusive) and
stop pixel (exclusive)

Return type
(int or array, int or array)

pix2skycoord(*pix*)

Return the sky coordinate for the center of a given pixel

Parameters
pix (*int or array*) – Pixel number

Returns
SkyCoord

pix2uniq(*pix*)

Get the UNIQ representation of a given pixel index.

Parameters
pix (*int*) – Pixel number in the current scheme (as used for [])

pix2vec(*pix*)

Return a vector corresponding to the center of a pixel

Parameters
pix (*int or array*) –

Returns
Size (3,N)

Return type
array

pix_order_list()

Get a list of lists containing all pixels sorted by order

Returns

(pix_per_order, nest_pix_per_order)
Each list has a size equal to the map order. Each element is a list of all pixels whose order matches the index of the list position. The first output contains the index of the pixels, while the second contains their coresponding pixel number in a nested scheme.

Return type
(list, list)

pix_rangesets(*nside=None*, *argsort=False*)

Get the equivalent range of *child pixels* in nested scheme for a map of equal or higher nside

Parameters

- **nside** (*int or None*) – Nside of output range sets. If None, the map nside will be used. nside = mhealpy.MAX_NSIDE returns the cached result
- **argsort** (*bool*) – Also return also the indices that would sort the array.

Returns

With columns named 'start' (inclusive) and
'stop' (exclusive)

Return type
recarray

pixarea(*pix=None*)

Return area of a pixel

Parameters
pix (*int or array*) – Pixel number. Only relevant for MOC maps. Default: All pixels for MOC, a single value for single resolution

Returns
Quantity

plot_grid(*ax='mollview'*, *ax_kw={}*, *step=32*, *coord=None*, ***kwargs*)

Plot the pixel boundaries of a Healpix grid

Parameters

- **ax** (*WCSAxes or str*) – Astropy's WCSAxes to plot the map. Either an existing instance or the name of a registered projection.
- **ax_kw** (*dict*) – Extra arguments if a new axes needs to be created.
- **step** (*int*) – How many points per pixel side
- **coord** (*str*) – Instrinsic coordinates of the map. Either 'G' (Galactic), 'E' (Ecliptic) , 'C' (Celestial = Equatorial) or any other coordinate frame recognized by astropy. The default is 'C' unless `coordsys` is defined. This option overrides `coordsys`
- ****kwargs** – Passed to matplotlib.pyplot.plot()

Returns

The first return value
corresponds to the output `pyplot.plot()` for one of the pixels. The second is the astropy WCSAxes object used.

Return type
matplotlib.lines.Line2D list, WCSAxes

query_disc(*vec*, *radius*, *inclusive=False*, *fact=4*)

Parameters

- **vec** (*float, sequence of 3 elements, SkyCoord*) – The coordinates of unit vector defining the disk center.
- **radius** (*float*) – The radius (in radians) of the disk
- **inclusive** (*bool*) – f False, return the exact set of pixels whose pixels centers lie within the region; if True, return all pixels that overlap with the region.
- **fact** (*int*) – Only used when inclusive=True. The overlapping test will be done at the resolution fact*nside. For NESTED ordering, fact must be a power of 2, less than 2**30, else it can be any positive integer. Default: 4.

Returns
The pixels which lie within the given disc.

Return type
int array

query_polygon(*vertices*, *inclusive=False*, *fact=4*)

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

Parameters

- **vertices** (*float*) – Vertex array containing the vertices of the polygon, shape (N, 3).
- **inclusive** (*bool*) – f False, return the exact set of pixels whose pixels centers lie within the region; if True, return all pixels that overlap with the region.
- **fact** (*int*) – Only used when inclusive=True. The overlapping test will be done at the resolution fact*nside. For NESTED ordering, fact must be a power of 2, less than 2**30, else it can be any positive integer. Default: 4.

Returns
The pixels which lie within the given polygon.

Return type
int array

query_strip(*theta1*, *theta2*, *inclusive=False*)

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

Parameters

- **theta** (*float*) – First colatitude (radians)
- **phi** (*float*) – Second colatitude (radians)
- **inclusive** (*bool*) – f False, return the exact set of pixels whose pixels centers lie within the region; if True, return all pixels that overlap with the region.

Returns
The pixels which lie within the given strip.

Return type
int array

property scheme

Return HEALPix scheme

Returns
Either 'NESTED', 'RING' or 'NUNIQ'

Return type
str

property uniq

Get an array with the NUNIQ numbers for all pixels

vec2pix(*x*, *y*, *z*)

Get the pixel (as used in []) that contains a given coordinate

Parameters

- **x** (*float or array*) – x coordinate
- **y** (*float or array*) – y coordinate
- **z** (*float or array*) – z coordinate

Returns
int or array

moc_sort()

Sort the uniq pixels composing a MOC map based on its rangeset representation

4.2 Pixelization functions

These functions can be call without referencing any class. e.g.:

```
>>> import mhealpy as mhp
>>> mhp.nest2uniq(nside = 128, ipix = 3)
65539
```

4.2.1 Single-resolution maps

mhealpy.pixelfunc.single.**order2npix**(*order*)

Get the number of pixel for a map of a given order

Parameters
order (*int or array*) –

Returns
int or array

mhealpy.pixelfunc.single.**nside2order**(*nside*)

mhealpy.pixelfunc.single.**order2nside**(*order*)

mhealpy.pixelfunc.single.**nside2npix**(*nside*)

mhealpy.pixelfunc.single.**npix2nside**(*npix*)

mhealpy.pixelfunc.single.**nside2pixarea**(*nside*)

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

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

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

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

mhealpy.pixelfunc.single.**vec2ang**(*vectors*)

mhealpy.pixelfunc.single.**ang2vec**(*theta*, *phi*, *lonlat=False*)

mhealpy.pixelfunc.single.**nest2ring**(*nside*, *ipix*)

mhealpy.pixelfunc.single.**ring2nest**(*nside*, *ipix*)

mhealpy.pixelfunc.single.**isnpixok**(*npix*)

mhealpy.pixelfunc.single.**get_all_neighbours**(*nside*, *theta*, *phi=None*, *nest=False*, *lonlat=False*)

mhealpy.pixelfunc.single.**query_disc**(*nside*, *vec*, *radius*, *inclusive=False*, *fact=4*, *nest=False*)

mhealpy.pixelfunc.single.**query_polygon**(*nside*, *vertices*, *inclusive=False*, *fact=4*, *nest=False*)

mhealpy.pixelfunc.single.**query_strip**(*nside*, *theta1*, *theta2*, *inclusive=False*, *nest=False*)

mhealpy.pixelfunc.single.**boundaries**(*nside*, *pix*, *step=1*, *nest=False*)

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

4.2.2 Multi-resolution maps

mhealpy.pixelfunc.moc.**uniq2nside**(*uniq*)

Extract the corresponding nside from a UNIQ numbered pixel

Parameters
uniq (*int or array*) – Pixel number

Returns
int or array

mhealpy.pixelfunc.moc.**uniq2nest**(*uniq*)

Convert from UNIQ ordering scheme to NESTED

Parameters
uniq (*int or array*) – Pixel number

Return
(int or array, int or array): nside, npix

mhealpy.pixelfunc.moc.**nest2uniq**(*nside*, *ipix*)

Convert from from NESTED to UNIQ scheme

Parameters

- **nside** (*int*) – HEALPix NSIDE parameter
- **ipix** (*int or array*) – Pixel number in NESTED scheme

Returns
int or array

mhealpy.pixelfunc.moc.**nest2range**(*nside_input*, *pix*, *nside_output*)

Get the equivalent range of pixel that correspond to all *child pixels* of a map of a greater order.

Parameters

- **nside_input** (*int or array*) – Nside of input pixel
- **pix** (*int or array*) – Input pixel.
- **nside_output** (*int*) – Nside of map with *child pixels*

Returns

Start pixel (inclusive) and
stop pixel (exclusive)

Return type
(int or array, int or array)

mhealpy.pixelfunc.moc.**uniq2range**(*nside*, *uniq*)

Convert from a pixel number in NUNIQ scheme to the range of children pixels that it would correspond to in a NESTED map of a given order

Parameters

- **order** (*int*) – Nside of equivalent single resolution map
- **uniq** (*int or array*) – Pixel number in NUNIQ scheme

Returns

Start pixel (inclusive) and
stop pixel (exclusive)

Return type
(int or array, int or array)

mhealpy.pixelfunc.moc.**range2uniq**(*nside*, *pix_range*)

Convert from range of children pixels in a NESTED map of a given order to the corresponding uniq pixel number.

Parameters

- **nside** (*int*) – Nside of equivalent single resolution map
- **pix_range** (*int or array, int or array*) – Star pixel (inclusive) and stop pixel (exclusive)

Returns
int

PYTHON MODULE INDEX

m

INDEX