
healsparse

Eli Rykoff and Javier Sanchez

Oct 08, 2021

CONTENTS:

1	Install	3
2	Quickstart	5
3	Basic HealSparse Interface	7
3.1	Getting Started	7
3.2	Integer Maps	8
3.3	Recarray Maps	9
3.4	Wide Masks	9
3.5	Writing Maps	10
3.6	Metadata	10
3.7	Coverage Masks	10
3.8	Fractional Detection Maps	11
3.9	Basic Visualization	11
4	HealSparse Map Operations	13
4.1	Introduction	13
4.2	Operations With a Constant	13
4.3	Operations With Multiple Maps	14
5	HealSparse Geometry	15
5.1	HealSparse Geometry Shapes	15
5.2	Making a Map	15
5.3	Using realize_geom()	16
6	HealSparse Randoms	17
6.1	Fast Random Generation	17
6.2	Regular Random Generation	17
7	Concatenation of HealSparse Files	19
7.1	Using cat_healsparse_files()	19
8	HealSparseMap File Specification v1.4.0	21
9	HealSparseMap Memory Layout	23
9.1	Terminology	23
9.2	Pixel Lookups	23
9.3	Coverage Map	24
9.4	Sparse Map	24
10	HealSparseMap FITS Serialization	27

10.1 Coverage Map	27
11 HealSparseMap Parquet Serialization	29
11.1 Parquet File Structure	29
11.2 Parquet Metadata	29
11.3 Parquet Coverage Map	29
11.4 Parquet Map Files	30
12 Modules API Reference	31
12.1 healsparse modules	31
13 Search	49
Python Module Index	51
Index	53

HealSparse is a sparse implementation of [HEALPix](#) in Python, written for the Rubin Observatory Legacy Survey of Space and Time Dark Energy Science Collaboration ([DESC](#)). *HealSparse* is a pure Python library that sits on top of [numpy](#) and [healpy](#) and is designed to avoid storing full sky maps in case of partial coverage, including easy reading of sub-maps. This reduces the overall memory footprint allowing maps to be rendered at arcsecond resolution while keeping the familiarity and power of [healpy](#).

HealSparse expands on [healpy](#) and straight [HEALPix](#) maps by allowing maps of different data types, including 32- and 64-bit floats; 8-, 16-, 32-, and 64-bit integers; “wide bit masks” of arbitrary width (allowing hundreds of bits to be efficiently and conveniently stored); and [numpy](#) record arrays. Arithmetic operations between maps are supported, including sum, product, min/max, and and/or/xor bitwise operations for integer maps. In addition, there is general support for any [numpy](#) universal function.

HealSparse also includes a simple geometric primitive library, to render circles and convex polygons.

The code is hosted in [GitHub](#). Please use the [issue tracker](#) to let us know about any problems or questions with the code. The list of released versions of this package can be found [here](#), with the master branch including the most recent (non-released) development.

The *HealSparse* code was written by Eli Rykoff and Javier Sanchez based on an idea from Anže Slosar. This software was developed under the Rubin Observatory Legacy Survey of Space and Time (LSST) Dark Energy Science Collaboration (DESC) using LSST DESC resources. The DESC acknowledges ongoing support from the Institut National de Physique Nucléaire et de Physique des Particules in France; the Science & Technology Facilities Council in the United Kingdom; and the Department of Energy, the National Science Foundation, and the LSST Corporation in the United States. DESC uses resources of the IN2P3 Computing Center (CC-IN2P3–Lyon/Villeurbanne - France) funded by the Centre National de la Recherche Scientifique; the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231; STFC DiRAC HPC Facilities, funded by UK BIS National E-infrastructure capital grants; and the UK particle physics grid, supported by the GridPP Collaboration. This work was performed in part under DOE Contract DE-AC02-76SF00515.

INSTALL

HealSparse requires *healpy*, *numpy*, and *astropy*. If you have *fitsio* installed then additional features including memory-efficient concatenation of *HealSparse* maps are made available.

HealSparse is available at [pypi](#), and the most convenient way of installing the latest released version is simply:

```
pip install healsparse
```

To install from source, you can run from the root directory:

```
python setup.py install
```

or use *pip* from the root directory:

```
pip install .
```


QUICKSTART

A [jupyter notebook](#) is available for tutorial purposes [here](#)

BASIC HEALSPARSE INTERFACE

3.1 Getting Started

To conserve memory, *HealSparse* uses a dual-map approach, where a low-resolution full-sky “coverage map” is combined with a high resolution map containing the pixel data where it is available. The resolution of the coverage map is controlled by the `nside_coverage` parameter, and the resolution of the high-resolution map is controlled by the `nside_sparse` parameter. Behind the scenes, *HealSparse* uses clever indexing to allow the user to treat these as contiguous maps with minimal overhead. **All HealSparse maps use HEALPix nest indexing behind the scenes, should be treated as nest-indexed maps.**

There are 3 basic ways to make a `HealSparseMap`. First, one can read in an existing *HEALPix* map; second, one can read in an existing `HealSparseMap`; and third, one can create a new map.

```
import numpy as np
import healsparse

# To read a HEALPix map, the nside_coverage must be specified
map1 = healsparse.HealSparseMap.read('healpix_map.fits', nside_coverage=32)

# To read a healsparse map, no additional keywords are necessary
map2 = healsparse.HealSparseMap.read('healsparse_map.hs')

# To read part of a healsparse map, you can specify the coverage pixels to read
map2_partial = healsparse.HealSparseMap.read('healsparse_map.hs', pixels=[100, 101])

# To create a new map, the resolutions and datatype must be specified
nside_coverage = 32
nside_sparse = 4096
map3 = healsparse.HealSparseMap.make_empty(nside_coverage, nside_sparse, np.float64)
```

To set values in the map, you can use simple indexing or the explicit API:

```
map3[0: 1000] = np.arange(1000, dtype=np.float64)
map3.update_values_pix(np.arange(1000, 2000), np.arange(1000, dtype=np.float64),
↳ nest=True)
```

To retrieve values from the map, you can use simple indexing or the explicit API via pixels or positions:

```
print(map3[0: 1000])
>>> [0. ... 999.]
print(map3.get_values_pix(np.arange(1000, 2000), nest=True))
```

(continues on next page)

(continued from previous page)

```
>>> [0. ... 999.]
print(map3.get_values_pos(45.0, 0.1, lonlat=True))
>>> 51.0
```

A HealSparseMap has the concept of “valid pixels”, the pixels over which the map is defined (as opposed to `healpy.UNSEEN` in the case of floating point maps). You can retrieve the array of valid pixels or the associated positions of the valid pixels easily:

```
print(map3.valid_pixels)
>>> [ 0  1  2 ... 1997 1998 1999]

ra, dec = map3.valid_pixels_pos(lonlat=True)
print(ra)
>>> [45.         ... 45.3515625 ]
print(dec)
>>> [0.00932548 ... 0.81134431]
```

You can convert a HealSparseMap to a `healpy` map (numpy array) either by using a full slice (`[:]`) or with the `generate_healpix_map()` method. Do watch out, at high resolution this can blow away your memory! In these cases, `generate_healpix_map()` can degrade the map before conversion, using a reduction function (over valid pixels) of your choosing, including mean, median, std, max, min, `:code: and`, `:code: or`, `:code: sum`, `:code: prod` (product), and `:code: wmean` (weighted mean).

```
hmap4096 = map3[:]
hmap128 = map3.generate_healpix_map(nside=128, reduction='mean')
```

3.2 Integer Maps

In addition to floating-point maps, which are natively supported by `healpy`, `HealSparseMap` supports integer maps. The “sentinel” value of these maps (equivalent to `healpy.UNSEEN`) is either `-MAXINT` or `0`, depending on the desired use of the map (e.g., integer values or positive bitmasks). Note that these maps cannot be trivially converted to `healpy` maps because `HEALPix` has no concept of sentinel values that are not `healpy.UNSEEN`, which is a very large negative floating-point value.

```
import numpy as np
import healsparse

map_int = healsparse.HealSparseMap.make_empty(32, 4096, np.int32)
print(map_int)
>>> HealSparseMap: nside_coverage = 32, nside_sparse = 4096, int32

map_int[0: 1000] = np.arange(1000, dtype=np.int32)

print(map_int[500])
>>> 500
```

3.3 Recarray Maps

HealSparseMap also supports maps made up of `numpy` record arrays. These recarray maps will have one field that is the “primary” field which is used to test if a pixel has a valid value or not. Therefore, these recarray maps should be used to describe associated values that share the exact same valid footprint. Each field in the recarray can be treated as its own `HealSparseMap`. For example,

```
import numpy as np
import healsparse

dtype = [('a', np.float32), ('b', np.int32)]

map_rec = healsparse.HealSparseMap.make_empty(32, 4096, dtype, primary='a')

map_rec[0: 10000] = np.zeros(10000, dtype=dtype)
print(map_rec.valid_pixels)
>>> [ 0  1  2 ... 9997 9998 9999]

map_rec['a'][0: 5000] = np.arange(5000, dtype=np.float32)
map_rec['b'][5000: 10000] = np.arange(5000, dtype=np.int32)

print(map_rec[map_rec.valid_pixels])
>>> [(0.,  0) (1.,  0) (2.,  0) ... (0., 4997) (0., 4998) (0., 4999)]
```

Note that the call `map_rec['a'][0: 5000] = values` will work, but `map_rec[0: 5000]['a'] = values` will not. Also note that using the fields of the recarray *cannot* be used to set new pixels, this construction can only be used to change pixel values.

3.4 Wide Masks

HealSparse has support for “wide” bit masks with an arbitrary number of bits that are referred to by bit position rather than value. This is useful, for example, when constructing a coadd coverage map where every pixel can uniquely identify the set of input exposures that contributed at the location of that pixel. In the case of >64 input exposures you can no longer use a simple 64-bit integer bit mask. Wide mask bits are always specified by giving a list of integer positions rather than values (e.g., use `10` to set the 10th bit instead of `1024 = 2**10`).

```
import numpy as np
import healsparse

map_wide = healsparse.HealSparseMap.make_empty(32, 4096, healsparse.WIDE_MASK, wide_mask_
↳maxbits=128)

pixels = np.arange(10000)
map_wide.set_bits_pix(pixels, [4, 100])

print(map_wide.check_bits_pix(pixels, [2]))
>>> [False False False ... False False False]
print(map_wide.check_bits_pix(pixels, [4]))
>>> [ True True True ... True True True]
print(map_wide.check_bits_pix(pixels, [100]))
>>> [ True True True ... True True True]
```

(continues on next page)

(continued from previous page)

```
print(map_wide.check_bits_pix(pixels, [101]))
>>> [False False False ... False False False]

# Check if any of the bits are set
print(map_wide.check_bits_pos([45.2], [0.2], [100, 101], lonlat=True))
>>> [ True]
```

3.5 Writing Maps

Writing a HealSparseMap is easy. To write a map in the default FITS format:

```
map3.write('output_file.hs', clobber=False)
```

And to write a map in the Parquet format with pyarrow:

```
map3.write('output_file.hsparquet', clobber=False, format='parquet')
```

3.6 Metadata

You can also set key/value metadata to a map that will be stored in the fits header of the file and read back in. The keys must conform to FITS header key standards (strings, upper case). The metadata will be stored as a Python dictionary, and can be accessed with the metadata property.

```
metadata = {'KEY1': 5, 'KEY2': 10.0}
map3.metadata = metadata
print(map3.metadata['KEY2'])
>>> 10.0
```

3.7 Coverage Masks

A HealSparseMap contains a coverage map that defines the coarse coverage over the sky. You can retrieve a boolean array describing which pixels are covered in the map with the coverage_mask property:

```
import healpy as hp
import matplotlib.pyplot as plt

cov_mask = map3.coverage_mask
cov_pixels, = np.where(cov_mask)
ra, dec = hp.pix2ang(map3.nside_coverage, cov_pixels, lonlat=True, nest=True)
plt.plot(ra, dec, 'r.')
plt.show()
```

It is also possible to read the coverage map of a HealSparseMap on its own:

```
cov_map = healsparse.HealSparseCoverage.read('output_file.hs')
cov_mask = cov_map.coverage_mask
```

In some cases, you may be building a map and you already know the coverage when it will be finished. In this case, it can be faster to initialize the memory at the beginning. In this case, you can add `cov_pixels` to the `make_empty` call. Be aware this may make the map larger than your actual coverage.

```
import healsparse

nside_coverage = 32
nside_sparse = 4096
map4 = healsparse.HealSparseMap.make_empty(nside_coverage, nside_sparse, np.float32,
                                           cov_pixels=[5, 10, 20, 21])
```

3.8 Fractional Detection Maps

One can compute the fractional detection map of a `HealSparseMap` with the `fracdet_map()` method. This method will compute the fractional area covered by the sparse map at an arbitrary resolution (not higher than the native resolution, and not lower than the coverage map `nside_coverage`). This is a count of the fraction of “valid” sub-pixels (those that are not equal to the sentinel value) in the original map. These maps can be useful in conjunction with a degraded map to easily determine the coverage fraction of each degraded pixel.

In order to translate a `fracdet_map` to lower resolution, the `degrade()` method should be used with the default “mean” reduction operation. If one tries to compute the `fracdet_map` of an existing `fracdet_map` then you will not get the expected output, because this is the fractional coverage of the `fracdet_map` itself, not of the original sparse map.

3.9 Basic Visualization

`healsparse` does not provide any built-in visualization tools. However, it is possible to perform quick visualizations of a `HealSparseMap` using the `matplotlib` package. For example, we can take render our map as a collection of hexagonal cells using `matplotlib.pyplot.hexbin`:

```
import healsparse
import matplotlib.pyplot as plt

nside_coverage = 32
nside_sparse = 4096

# Generation of the map
hsp_map = healsparse.HealSparseMap.make_empty(nside_coverage, nside_sparse, np.float32)
idx = np.arange(2000, 6000)
hsp_map[idx] = np.random.uniform(size=idx.size).astype(np.float32)

# Visualization of the map
vpix, ra, dec = hsp_map.valid_pixels_pos(return_pixels=True)
plt.hexbin(ra, dec, C=hsp_map[vpix])
plt.colorbar()
plt.show()
```


HEALSPARSE MAP OPERATIONS

4.1 Introduction

HealSparse has support for basic arithmetic operations between maps, including sum, product, min/max, and and/or/xor bitwise operations for integer maps. In addition, there is general support for any `numpy` universal function. It is important to note that map operations are complicated by the fact that any given maps may not have the same pixel coverage. Therefore, map operations can be done either with “union” (where the final map has a valid pixel list that is the union of the input maps) or “intersection” (where the final map has a valid pixel list that is the intersection of the input maps). In the case of “union” operations, the maps that do not have coverage at a given point will be filled with an appropriate value (for example, 0 for summation and 1 for products).

4.2 Operations With a Constant

Arithmetic operations with a constant are very simple, and are handled with the default Python operations, supporting copy or in-place operations.

```
import numpy as np
import healsparse

map_float = healsparse.HealSparseMap.make_empty(32, 4096, np.float64)
map_float[0: 10000] = np.zeros(1, dtype=np.float64)

print(map_float[0: 10000])
>>> [0. 0. 0. ... 0. 0. 0.]

map_float += 10.0
print(map_float[0: 10000])
>>> [10. 10. 10. ... 10. 10. 10.]

map_float /= 10.0
print(map_float[0: 10000])
>>> [1. 1. 1. ... 1. 1. 1.]

map_float2 = map_float*100.0
print(map_float[0: 10000])
>>> [1. 1. 1. ... 1. 1. 1.]
print(map_float2[0: 10000])
>>> [100. 100. 100. ... 100. 100. 100.]
```

4.3 Operations With Multiple Maps

Operations between maps can be done with either “union” or “intersection” mode. The basic operations supported are `sum`, `product`, `min`, `max`, `or`, `and`, `xor`, and `ufunc`. Note that `or`, `and`, and `xor` operations are only supported for integer maps. Note that operations between maps are only supported if they have the same `nside_sparse` resolution and data type. In all cases, the function name is `operation_union()` or `operation_intersection()`. For example,

```
import numpy as np
import healsparse

map1 = healsparse.HealSparseMap.make_empty(32, 4096, np.float64)
map1[0: 10000] = np.ones(10000)
map2 = healsparse.HealSparseMap.make_empty_like(map1)
map2[5000: 15000] = np.ones(10000)*5.0

# The union sum will have coverage that was from map1 OR map2
sum_union = healsparse.operations.sum_union([map1, map2])
print(sum_union[sum_union.valid_pixels].min())
>>> 1.0
print(sum_union[sum_union.valid_pixels].max())
>>> 6.0
print(sum_union.valid_pixels.min(), sum_union.valid_pixels.max())
>>> 0 14999

# The intersection sum will have coverage that was from map1 AND map2
sum_intersection = healsparse.operations.sum_intersection([map1, map2])
print(sum_intersection[sum_intersection.valid_pixels].min())
>>> 6.0
print(sum_intersection[sum_intersection.valid_pixels].max())
>>> 6.0
print(sum_intersection.valid_pixels.min(), sum_intersection.valid_pixels.max())
>>> 5000 9999
```

HEALSPARSE GEOMETRY

HealSparse has a basic geometry library that allows you to generate maps from circles and convex polygons, as supported by `healpy`. Each geometric object is associated with a single value. On construction, geometry objects only contain information about the shape, and they are only rendered onto a *HEALPix* grid when requested.

There are two methods to realize geometry objects. The first is that each object can be used to generate a `HealSparseMap` map, and the second, for integer-valued objects is the `realize_geom()` method which can be used to combine multiple objects by *or-ing* the integer values together.

5.1 HealSparse Geometry Shapes

The two shapes supported are `Circle` and `Polygon`. They share a base class, and while the instantiation is different, the operations are the same.

Circle

```
import healsparse

# All units are decimal degrees
circ = healsparse.Circle(ra=200.0, dec=0.0, radius=1.0, value=1)
```

Convext Polygon

```
# All units are decimal degrees
poly = healsparse.Polygon(ra=[200.0, 200.2, 200.3, 200.2, 200.1],
                          dec=[0.0, 0.1, 0.2, 0.25, 0.13],
                          value=8)
```

5.2 Making a Map

To make a map from a geometry object, use the `get_map()` method as such. The higher resolution you choose, the better the aliasing at the edges (given that these are pixelized approximations of the true shapes). You can also combine two maps using the general operations. Note that if the polygon is an integer value, the default sentinel when using `get_map()` is 0.

```
smap_poly = poly.get_map(nside_coverage=32, nside_sparse=32768, dtype=np.int16)
smap_circ = circ.get_map(nside_coverage=32, nside_sparse=32768, dtype=np.int16)

combo = healsparse.or_union([smap_poly, smap_circ])
```

5.3 Using `realize_geom()`

You can only use `realize_geom()` to create maps from combinations of polygons if you are using integer maps, and want to or them together. This method is more memory efficient than generating each individual individual map and combining them, as above.

```
realized_combo = healsparse.HealSparseMap.make_empty(32, 32768, np.int16, sentinel=0)
healsparse.realize_geom([poly, circ], realized_combo)
```

HEALSPARSE RANDOMS

HealSparse can generate uniform randoms based on the valid pixels in a HealSparseMap map. It is up to the user of these random points to weight them appropriately.

There are two methods to generate randoms. The “fast” method requires more memory, and produces randoms that are quantized (at a very high level). The regular method is not quantized and does not require any extra memory, but is significantly slower.

6.1 Fast Random Generation

The fast random generation is run with `healsparse.make_uniform_randoms_fast()`. This code path requires more memory and may not be suitable for very large, high resolution masks. The output randoms are quantized by the `nside_randoms` quantity. The default is 2^{23} , which is approximately $7e-6$ arcsecond quantization, which should be adequate for most purposes.

```
import healsparse

circ = healsparse.Circle(ra=200.0, dec=0.0, radius=1.0, value=1)
smap = circ.get_map(nside_coverage=32, nside_sparse=32768, dtype=np.int16)

# Make 1000000 randoms
ra_rand, dec_rand = healsparse.make_uniform_randoms_fast(smap, 1000000)
```

6.2 Regular Random Generation

The regular random generation is run with `healsparse.make_uniform_randoms()`. This code requires less memory than the fast generation, and there is no quantization in the random points. However, it is slower than the fast generation. The API is very similar to `healsparse.make_uniform_randoms_fast()`.

```
import healsparse

circ = healsparse.Circle(ra=200.0, dec=0.0, radius=1.0, value=1)
smap = circ.get_map(nside_coverage=32, nside_sparse=32768, dtype=np.int16)

# Make 1000000 randoms
ra_rand, dec_rand = healsparse.make_uniform_randoms(smap, 1000000)
```


CONCATENATION OF HEALSPARSE FILES

HealSparse contains a routine for concatenating (combining) multiple HealSparseMap files. If `fitsio` is available, this will be done in a memory-efficient way. In this way, multiple non-overlapping maps can be combined. This makes possible a simple parallelized scatter-gather approach to creating complex survey maps, where individual tiles are run independently, and then all combined at the end.

7.1 Using `cat_healsparse_files()`

The `cat_healsparse_files()` routine takes in a list of filename, and an output filename. The individual files must have the same `nside_sparse`, but may have different `nside_coverage`. The output file will have the same `nside_coverage` as the first input file unless otherwise specified.

By default, for speed, the code will not check that the input HealSparseMap files are non-overlapping (that is, that they do not share `valid_pixels`; they may share coverage in the coverage map). This can be checked.

If `fitsio` is available (recommended), the combination is not done in-memory. This behavior can be modified by the user by setting `in_memory` to `True`. However, if only `astropy.io.fits` is available for FITS interfacing, the concatenation can only be done in-memory (and the `in_memory` value should be overridden).

```
import healsparse

healsparse.cat_healsparse_files(file_list, outfile, check_overlap=False, clobber=False,
                               in_memory=False, nside_coverage_out=None)
```


HEALSPARSEMAP FILE SPECIFICATION V1.4.0

A `HealSparseMap` file can be either a standard FITS file or a Parquet dataset. Each provides a way to record the data from a `HealSparseMap` object for efficient retrieval of individual coverage pixels. This document describes the memory layout of the key components of the `HealSparseMap` objects, as well as the specific file formats in FITS (*HealSparseMap FITS Serialization*) and Parquet (*HealSparseMap Parquet Serialization*).

HEALSPARSEMAP MEMORY LAYOUT

A HealSparseMap object has two primary components, the coverage map and the sparse map.

9.1 Terminology

This is a list of terminology used in a HealSparseMap object:

- `nside_sparse`: The *HEALPix* `nside` for the fine-grained (sparse) map
- `nside_coverage`: The *HEALPix* `nside` for the coverage map
- `bit_shift`: The number of bits to shift to convert from `nside_sparse` to `nside_coverage` in the *HEALPix* NEST scheme. $\text{bit_shift} = 2 * \log_2(\text{nside_sparse} / \text{nside_coverage})$.
- `valid_pixels`: The list of pixels with defined values ($>$ `sentinel`) in the sparse map.
- `sentinel`: The sentinel value that notes if a pixel is not a valid pixel. Default is `healpy.UNSEEN` for floating-point maps, `-MAXINT` for integer maps, and `0` for wide mask maps.
- `nfine_per_cov`: The number of fine (sparse) pixels per coverage pixel. $\text{nfine_per_cov} = 2 * \text{bit_shift}$.
- `wide_mask_width`: The width of a wide mask, in bytes.

9.2 Pixel Lookups

The HealSparseMap file format is derived from the method of encoding fast look-ups of arbitrary pixel values using the *HEALPix* nest pixel scheme.

Given a nest-encoded sparse (high resolution) pixel value, `pix_nest`, we can compute the coverage (low resolution) pixel value with a simple bit shift operation: `ipnest_cov = right_shift(pix_nest, bit_shift)`, where `bit_shift` is defined in *Terminology*.

The sparse map itself is stored in blocks of data, each of which includes `nfine_per_cov` contiguous pixels for each coverage pixel that contains valid data. To find the location *within* a given a coverage block, we need to subtract the first fine (sparse) nest pixel value for the given coverage pixel. Therefore, `first_pixel = nfine_per_cov * ipnest_cov`.

Next, if we have a map of offsets which points to the location of the proper sparse map block, `cov_map_raw_offset`, we find the look-up index is `index = pix_nest - nfine_per_cov * ipnest_cov + cov_map_raw_offset[ipnest_cov]`. In practice, we can combine the final terms here. The look-up index is then `index = pix_nest + cov_map[ipnest_cov]` where `cov_map = cov_map_raw_offset[ipnest_cov] - nfine_per_cov * ipnest_cov`.

As described in *Sparse Map*, the first block in the sparse map is special, and is always filled with `sentinel` values. All `cov_map` indices for coverage pixels outside the coverage map point to this sentinel block.

9.3 Coverage Map

The coverage map encodes the mapping from raw pixel number to location within the sparse map. It is an integer (`numpy.int64`) map with $12 * \text{nside_coverage} * \text{nside_coverage}$ values, all of which are filled.

As described in *Pixel Lookups*, the coverage map contains indices that are offset pointers to the block in the sparse map with associated values for that coverage pixel. An empty `HealSparseMap` is initialized with the following coverage pixel values, which all point to the first `sentinel` block in the sparse map.

```
import numpy as np
import healpy as hp

cov_map[:] = -1*np.arange(hp.nside2npix(nside_coverage), dtype=np.int64)*nfine_per_cov
```

9.4 Sparse Map

The sparse map contains the map data, split into blocks, each of which is `nfine_per_cov` elements long. The first block is special, and is always filled with `sentinel` values.

The following datatypes are allowed:

- 1-byte unsigned integer (`numpy.uint8`)
- 1-byte signed integer (`numpy.int8`)
- 2-byte unsigned integer (`numpy.uint16`)
- 2-byte signed integer (`numpy.int16`)
- 4-byte unsigned integer (`numpy.uint32`)
- 4-byte signed integer (`numpy.int32`)
- 8-byte signed integer (`numpy.int64`)
- 4-byte floating point (`numpy.float32`)
- 8-byte floating point (`numpy.float64`)
- Numpy record array of numeric types that can be serialized with FITS
- The `WIDE_MASK` special encoding

Sparse Map Image

If the sparse map is a single datatype, it is held in memory as a one-dimensional image array. The first block of `nfine_per_cov` values are set to `sentinel`. Each additional block of `nfine_per_cov` is associated with a single element in the coverage map. These blocks may be in any arbitrary order, allowing for easy appending of new coverage pixels. All invalid pixels must be set to `sentinel`.

Sparse Map Wide Mask

If the sparse map is a wide mask map, the sparse map is held in memory as a `wide_mask_width * npix` array. The sentinel value for wide masks must be `0`, and all invalid pixels must be set to `0`.

Sparse Map Table

If the sparse map is a numpy record array type, it is held in memory as a one dimensional table array. The first block of `nfine_per_cov` values are set such that the `primary` field must be set to `sentinel`. As with the sparse map image, each additional block of `nfine_per_cov` is associated with a single element in the coverage map. These blocks may

be in any arbitrary order, allowing for easy appending of new coverage pixels. All invalid pixels must have the primary field set to `sentinel`.

HEALSPARSEMAP FITS SERIALIZATION

A `HealSparseMap` FITS file is a standard FITS file with two extensions. The primary (zeroth) extension is an integer image that describes the coverage map, and the first extension is an image or binary table that describes the sparse map. This section describes the file format specification of these two extensions in the FITS file.

10.1 Coverage Map

Coverage Map Header

The coverage map header must contain the following keywords:

- **EXTNAME** must be "COV"
- **PIXTYPE** must be "HEALSPARSE"
- **NSIDE** is equal to `nside_coverage`

Coverage Map Image

The FITS coverage map is a direct serialization of the coverage map image in-memory layout described in [Coverage Map](#).

Sparse Map Header

The sparse map header must contain:

- **EXTNAME** must be "SPARSE"
- **PIXTYPE** must be "HEALSPARSE"
- **SENTINEL** is equal to `sentinel`
- **NSIDE** is equal to `nside_sparse`

If the sparse map is a numpy record array, it must contain:

- **PRIMARY** is equal to the name of the "primary" field which defines the valid pixels.

If the sparse map is a wide mask, it must contain:

- **WIDEMASK** must be True
- **WWIDTH** must be the width (in bytes) of the wide mask.

Sparse Map Image

If the sparse map is not of a numpy record array type, it is stored as a one dimensional image array. If the image is an integer type with 32 bits or fewer, it may be stored with FITS tile compression, with the tile size set to

the block size (`nfine_per_cov`). If the image is a floating-point image, it may be stored with FITS tile compression, with `quantization_level=0` and `GZIP_2` (lossless gzip compression), with the tile size set to the block size (`nfine_per_cov`).

Sparse Map Wide Mask

If the sparse map is a wide mask map, the sparse map is stored as a flattened version of the in-memory `wide_mask_width * npix` array. This should be flattened on storage, and reshaped on read, using the default numpy memory ordering. The wide mask image may be stored with FITS tile compression, with the tile size set to the block size times with width (`wide_mask_width * nfine_per_cov`).

Sparse Map Table

If the sparse map is a numpy record array type, it is stored as a one dimensional table array.

HEALSPARSEMAP PARQUET SERIALIZATION

A `HealSparseMap` serialized as Parquet is sharded as a Parquet dataset for efficient access to sub-regions of very large area, high resolution maps. The `HealSparseMap` Parquet dataset consists of a directory with two metadata files; the coverage map; and a list of low resolution “i/o pixel” directories (default `nside_io=4`). In each i/o pixel directory is a Parquet file with all of the sparse map data from that i/o pixel, divided into Parquet row groups for each coverage pixel. The `HealSparseMap` Parquet format uses the default `snappy` per-column compression.

11.1 Parquet File Structure

11.2 Parquet Metadata

The metadata is stored separately in the `_metadata` and `_common_metadata` files in the dataset directory, as per the Parquet dataset specification. The metadata is stored as a set of key-value pairs, each of which is a binary string. For simplicity we describe the strings here as regular strings, but beware that behind the scenes they are stored in the python `b'string'` format.

The following metadata strings are required: * `'healsparse::version': '1'` * `'healsparse::nside_sparse': str(nside_sparse)` * `'healsparse::nside_coverage': str(nside_coverage)` * `'healsparse::nside_io': str(nside_io)` * `'healsparse::filetype': 'healsparse'` * `'healsparse::primary': 'primary' or ''` * `'healsparse::sentinel': str(sentinel) or 'UNSEEN'` * `'healsparse::widemask': 'True' or 'False'` * `'healsparse::wwidth': str(wide_mask_width) or '1'`

Note that the string `'UNSEEN'` will use the special value `healpy.UNSEEN` to fill empty/overflow pixels.

Additional metadata from the map is stored as a FITS header string (for compatibility with the FITS serialization) such that: * `'healsparse::header': header_string`

11.3 Parquet Coverage Map

The coverage map is a Parquet file with the name `_coverage.parquet`, stored in the dataset directory. The coverage map has two columns: * `cov_pix`: Valid coverage pixels (`nside = nside_coverage`) for the sparse map. * `row_group`: The row group index within the appropriate i/o pixel file to find the sparse data for the given coverage map.

11.4 Parquet Map Files

Each sparse map Parquet file covers one i/o pixel. The name of each file is `iopix=###/###.parquet`, where `###` is a zero-padded three digit number for the given i/o pixel. By putting each pixel in its own directory with this naming scheme we allow pyarrow to use the hive partitioning and only touch the files as necessary.

Each file is written as a series of Parquet row groups. Each row group contains all the data for a single coverage pixel, with `nfine_per_cov` rows per row group. The row group number within the given i/o pixel is recorded in the `_coverage.parquet` coverage map file for quick access to individual and groups of coverage pixels.

The exact format of the data depends on whether the map is a simple image, a wide mask, or a record array.

Sparse Map Image

If the sparse map is not of a numpy record array type, it is stored in a two-column Parquet table. The schema is given by: `* cov_pix: int32 * sparse: Datatype of the sparse image data.`

The `cov_pix` gives the coverage pixel of the data, and is redundant with the data in `_coverage.parquet`. It compresses very efficiently, and can be used to reconstruct the `_coverage.parquet` from the data files if necessary.

The `sparse` column has the sparse map data (with sparse map image datatype).

Unlike the FITS serialization, the initial “overflow” coverage pixel is not serialized. Instead, on read this is filled in with the `sentinel` value from the Parquet metadata.

Sparse Map Wide Mask

If the sparse map is a wide mask map, the schema is the same as for a regular sparse map image. In this case, as with the FITS serialization, the sparse map is stored as a flattened version of the in-memory `wide_mask_width * npix` array. This means that there will be `wide_mask_width * nfine_per_cov` rows per row group in each wide mask Parquet file.

Sparse Map Table

If the sparse map is a numpy record array type, it is stored as a multi-column Parquet table with the following schema: `* cov_pix: int32 * column_1: Datatype of column 1. * column_2: Datatype of column 2. * Etc.`

Unlike the FITS serialization, the initial “overflow” coverage pixel is not serialized. Instead, on read this is filled in with the `sentinel` value from the Parquet metadata for the `primary` column. The other columns in the overflow coverage pixel are filled with the default sentinel for that datatype (e.g., `healpy.UNSEEN` for floating-point columns and `-MAXINT` for integer columns).

MODULES API REFERENCE

12.1 healsparse modules

12.1.1 HealSparseMap

class `healsparse.healSparseMap.HealSparseMap`(*cov_map=None, cov_index_map=None, sparse_map=None, nside_sparse=None, healpix_map=None, nside_coverage=None, primary=None, sentinel=None, nest=True, metadata=None, _is_view=False*)

Bases: `object`

Class to define a HealSparseMap

__add__(*other*)

Add a constant.

Cannot be used with recarray maps.

__and__(*other*)

Perform a bitwise and with a constant.

Cannot be used with recarray maps.

__getitem__(*key*)

Get part of a healpix map.

__iadd__(*other*)

Add a constant, in place.

Cannot be used with recarray maps.

__iand__(*other*)

Perform a bitwise and with a constant, in place.

Cannot be used with recarray maps.

__imul__(*other*)

Multiply a constant, in place.

Cannot be used with recarray maps.

__init__(*cov_map=None, cov_index_map=None, sparse_map=None, nside_sparse=None, healpix_map=None, nside_coverage=None, primary=None, sentinel=None, nest=True, metadata=None, _is_view=False*)

Instantiate a HealSparseMap.

Can be created with `cov_index_map`, `sparse_map`, and `nside_sparse`; or with `healpix_map`, `nside_coverage`. Also see `HealSparseMap.read()`, `HealSparseMap.make_empty()`, `HealSparseMap.make_empty_like()`.

Parameters

- **cov_map** (*HealSparseCoverage*, optional) – Coverage map object
- **cov_index_map** (*np.ndarray*, optional) – Coverage index map, will be deprecated
- **sparse_map** (*np.ndarray*, optional) – Sparse map
- **nside_sparse** (*int*, optional) – Healpix nside for sparse map
- **healpix_map** (*np.ndarray*, optional) – Input healpix map to convert to a sparse map
- **nside_coverage** (*int*, optional) – Healpix nside for coverage map
- **primary** (*str*, optional) – Primary key for recarray, required if dtype has fields.
- **sentinel** (*int* or *float*, optional) – Sentinel value. Default is *hp.UNSEEN* for floating-point types, and minimum int for int types.
- **nest** (*bool*, optional) – If input healpix map is in nest format. Default is True.
- **metadata** (*dict*-like, optional) – Map metadata that can be stored in FITS header format.
- **_is_view** (*bool*, optional) – This healSparse map is a view into another healsparse map. Not all features will be available. (Internal usage)

Returns healSparseMap

Return type *HealSparseMap*

__ior__ (*other*)

Perform a bitwise or with a constant, in place.

Cannot be used with recarray maps.

__ipow__ (*other*)

Divide a constant, in place.

Cannot be used with recarray maps.

__isub__ (*other*)

Subtract a constant, in place.

Cannot be used with recarray maps.

__itruediv__ (*other*)

Divide a constant, in place.

Cannot be used with recarray maps.

__ixor__ (*other*)

Perform a bitwise xor with a constant, in place.

Cannot be used with recarray maps.

__mul__ (*other*)

Multiply a constant.

Cannot be used with recarray maps.

__or__ (*other*)

Perform a bitwise or with a constant.

Cannot be used with recarray maps.

__pow__(*other*)

Raise the map to a power.

Cannot be used with recarray maps.

__repr__()

Return repr(self).

__setitem__(*key, value*)

Set part of a healpix map

__str__()

Return str(self).

__sub__(*other*)

Subtract a constant.

Cannot be used with recarray maps.

__truediv__(*other*)

Divide a constant.

Cannot be used with recarray maps.

__weakref__

list of weak references to the object (if defined)

__xor__(*other*)

Perform a bitwise xor with a constant.

Cannot be used with recarray maps.

apply_mask(*mask_map, mask_bits=None, mask_bit_arr=None, in_place=True*)

Apply an integer mask to the map. All pixels in the integer mask that have any bits in mask_bits set will be zeroed in the output map. The default is that this operation will be done in place, but it may be set to return a copy with a masked map.

Parameters

- **mask_map** (*HealSparseMap*) – Integer mask to apply to the map.
- **mask_bits** (*int*, optional) – Bits to be treated as bad in the mask_map. Default is None (all non-zero pixels are masked)
- **mask_bit_arr** (*list* or *np.ndarray*, optional) – Array of bit values, used if mask_map is a wide_mask_map.
- **in_place** (*bool*, optional) – Apply operation in place. Default is True

Returns **masked_map** – self if in_place is True, a new copy otherwise

Return type *HealSparseMap*

astype(*dtype, sentinel=None*)

Convert sparse map to a different numpy datatype, including sentinel values. If sentinel is not specified the default for the converted datatype is used (*healpy.UNSEEN* for float, and *-MAXINT* for ints).

Parameters

- **dtype** (*numpy.dtype*) – Valid numpy dtype for a single array.
- **sentinel** (*int* or *float*, optional) – Converted map sentinel value.

Returns **sparse_map** – New map with new data type.

Return type *HealSparseMap*

check_bits_pix(*pixels, bits, nest=True*)

Check the bits at the map for a set of pixels.

Parameters

- **pixel** (*np.ndarray*) – Integer array of healpix pixels.
- **nest** (*bool*, optional) – Are the pixels in nest scheme? Default is True.
- **bits** (*list*) – List of bits to check

Returns **bit_flags** – Array of *np.bool_* flags on whether any of the input bits were set

Return type *np.ndarray*

check_bits_pos(*ra_or_theta, dec_or_phi, bits, lonlat=True*)

Check the bits at the map for an array of positions. Positions may be theta/phi co-latitude and longitude in radians, or longitude and latitude in degrees.

Parameters

- **ra_or_theta** (*float*, array-like) – Angular coordinates of points on a sphere.
- **dec_or_phi** (*float*, array-like) – Angular coordinates of points on a sphere.
- **lonlat** (*bool*, optional) – If True, input angles are longitude and latitude in degrees. Otherwise, they are co-latitude and longitude in radians.
- **bits** (*list*) – List of bits to check

Returns **bit_flags** – Array of *np.bool_* flags on whether any of the input bits were set

Return type *np.ndarray*

clear_bits_pix(*pixels, bits, nest=True*)

Clear bits of a *wide_mask* map.

Parameters

- **pixels** (*np.ndarray*) – Integer array of *sparse_map* pixel values
- **bits** (*list*) – List of bits to clear

static convert_healpix_map(*healpix_map, nside_coverage, nest=True, sentinel=-1.6375e+30*)

Convert a healpix map to a *healsparsemap*.

Parameters

- **healpix_map** (*np.ndarray*) – Numpy array that describes a healpix map.
- **nside_coverage** (*int*) – Nside for the coverage map to construct
- **nest** (*bool*, optional) – Is the input map in nest format? Default is True.
- **sentinel** (*float*, optional) – Sentinel value for null values in the *sparse_map*. Default is *hp.UNSEEN*

Returns

- **cov_map** (*HealSparseCoverage*) – Coverage map with pixel indices
- **sparse_map** (*np.ndarray*) – Sparse map of input values.

property coverage_map

Get the fractional area covered by the *sparse_map* in the resolution of the coverage map

Returns **cov_map** – Float array of fractional coverage of each pixel

Return type *np.ndarray*

property coverage_mask

Get the boolean mask of the coverage map.

Returns `cov_mask` – Boolean array of coverage mask.

Return type `np.ndarray`

degrade(*nside_out*, *reduction*='mean', *weights*=None)

Method to reduce the resolution, i.e., increase the pixel size of a given sparse map.

Parameters

- **nside_out** (*int*) – Output Nside resolution parameter.
- **reduction** (*str*) – Reduction method (mean, median, std, max, min, and, or, sum, prod, wmean).
- **weights** (*HealSparseMap*) – If the reduction is *wmean* this is the map with the weights to use. It should have the same characteristics as the original map.

Returns `healSparseMap` – New map, at the desired resolution.

Return type `HealSparseMap`

property dtype

get the dtype of the map

fracdet_map(*nside*)

Get the fractional area covered by the sparse map at an arbitrary resolution. This output `fracdet_map` counts the fraction of “valid” sub-pixels (those that are not equal to the sentinel value) at the desired *nside* resolution.

Note: You should not compute the `fracdet_map` of an existing `fracdet_map`. To get a `fracdet_map` at a lower resolution, use the `degrade` method with the default “mean” reduction.

Parameters **nside** (*int*) – Healpix *nside* for `fracdet` map. Must not be greater than sparse resolution or less than coverage resolution.

Returns `fracdet_map` – Fractional coverage map.

Return type `HealSparseMap`

generate_healpix_map(*nside*=None, *reduction*='mean', *key*=None, *nest*=True)

Generate the associated healpix map

if *nside* is specified, then reduce to that *nside*

Parameters

- **nside** (*int*) – Output *nside* resolution parameter (should be a multiple of 2). If not specified the output resolution will be equal to the parent’s `sparsemap` `nside_sparse`
- **reduction** (*str*) – If a change in resolution is requested, this controls the method to reduce the map computing the “mean”, “median”, “std”, “max”, “min”, “sum” or “prod” (product) of the neighboring pixels to compute the “degraded” map.
- **key** (*str*) – If the parent `HealSparseMap` contains recarrays, `key` selects the field that will be transformed into a HEALPix map.
- **nest** (*bool*, optional) – Output healpix map should be in nest format?

Returns `hp_map` – Output HEALPix map with the requested resolution.

Return type `np.ndarray`

get_single(*key*, *sentinel=None*, *copy=False*)

Get a single healpix map out of a recarray map, with the ability to override a sentinel value.

Parameters

- **key** (*str*) – Field for the recarray
- **sentinel** (*int* or *float* or *None*, optional) – Override the default sentinel value. Default is *None* (use default)

get_valid_area(*degrees=True*)

Get the area covered by valid pixels

Parameters

- **degrees** (*bool* If *True* (default) returns the area in square degrees,) –
- **steradians** (*if False it returns the area in*) –

Returns valid_area

Return type *float*

get_values_pix(*pixels*, *nest=True*, *valid_mask=False*)

Get the map value for a set of pixels.

Parameters

- **pixel** (*np.ndarray*) – Integer array of healpix pixels.
- **nest** (*bool*, optional) – Are the pixels in nest scheme? Default is *True*.
- **valid_mask** (*bool*, optional) – Return mask of *True/False* instead of values

Returns values – Array of values/validity from the map.

Return type *np.ndarray*

get_values_pos(*ra_or_theta*, *dec_or_phi*, *lonlat=True*, *valid_mask=False*)

Get the map value for the position. Positions may be theta/phi co-latitude and longitude in radians, or longitude and latitude in degrees.

Parameters

- **ra_or_theta** (*float*, array-like) – Angular coordinates of points on a sphere.
- **dec_or_phi** (*float*, array-like) – Angular coordinates of points on a sphere.
- **lonlat** (*bool*, optional) – If *True*, input angles are longitude and latitude in degrees. Otherwise, they are co-latitude and longitude in radians.
- **valid_mask** (*bool*, optional) – Return mask of *True/False* instead of values

Returns values – Array of values/validity from the map.

Return type *np.ndarray*

property is_integer_map

Check that the map is an integer map

Returns is_integer_map

Return type *bool*

property is_rec_array

Check that the map is a recArray map.

Returns is_rec_array

Return type *bool*

property `is_unsigned_map`

Check that the map is an unsigned integer map

Returns `is_unsigned_map`

Return type *bool*

property `is_wide_mask_map`

Check that the map is a wide mask

Returns `is_wide_mask_map`

Return type *bool*

classmethod `make_empty`(*nside_coverage*, *nside_sparse*, *dtype*, *primary=None*, *sentinel=None*, *wide_mask_maxbits=None*, *metadata=None*, *cov_pixels=None*)

Make an empty map with nothing in it.

Parameters

- **nside_coverage** (*int*) – Nside for the coverage map
- **nside_sparse** (*int*) – Nside for the sparse map
- **dtype** (*str* or *list* or *np.dtype*) – Datatype, any format accepted by numpy.
- **primary** (*str*, optional) – Primary key for recarray, required if dtype has fields.
- **sentinel** (*int* or *float*, optional) – Sentinel value. Default is *hp.UNSEEN* for floating-point types, and minimum int for int types.
- **wide_mask_maxbits** (*int*, optional) – Create a “wide bit mask” map, with this many bits.
- **metadata** (*dict*-like, optional) – Map metadata that can be stored in FITS header format.
- **cov_pixels** (*np.ndarray* or *list*) – List of integer coverage pixels to pre-allocate

Returns `healSparseMap` – `HealSparseMap` filled with sentinel values.

Return type *HealSparseMap*

classmethod `make_empty_like`(*sparsemap*, *nside_coverage=None*, *nside_sparse=None*, *dtype=None*, *primary=None*, *sentinel=None*, *wide_mask_maxbits=None*, *metadata=None*, *cov_pixels=None*)

Make an empty map with the same parameters as an existing map.

Parameters

- **sparsemap** (*HealSparseMap*) – Sparse map to use as basis for new empty map.
- **nside_coverage** (*int*, optional) – Coverage nside, default to `sparsemap.nside_coverage`
- **nside_sparse** (*int*, optional) – Sparse map nside, default to `sparsemap.nside_sparse`
- **dtype** (*str* or *list* or *np.dtype*, optional) – Datatype, any format accepted by numpy. Default is `sparsemap.dtype`
- **primary** (*str*, optional) – Primary key for recarray. Default is `sparsemap.primary`
- **sentinel** (*int* or *float*, optional) – Sentinel value. Default is `sparsemap._sentinel`
- **wide_mask_maxbits** (*int*, optional) – Create a “wide bit mask” map, with this many bits.
- **metadata** (*dict*-like, optional) – Map metadata that can be stored in FITS header format.
- **cov_pixels** (*np.ndarray* or *list*) – List of integer coverage pixels to pre-allocate

Returns healSparseMap – HealSparseMap filled with sentinel values.

Return type *HealSparseMap*

property metadata

Return the metadata dict.

Returns metadata

Return type *dict*

property nside_coverage

Get the nside of the coverage map

Returns nside_coverage

Return type *int*

property nside_sparse

Get the nside of the sparse map

Returns nside_sparse

Return type *int*

property primary

Get the primary field

Returns primary

Return type *str*

classmethod read(*filename, nside_coverage=None, pixels=None, header=False, degrade_nside=None, weightfile=None, reduction='mean', use_threads=False*)

Read in a HealSparseMap.

Parameters

- **filename** (*str*) – Name of the file to read. May be either a regular HEALPIX map or a HealSparseMap
- **nside_coverage** (*int*, optional) – Nside of coverage map to generate if input file is healpix map.
- **pixels** (*list*, optional) – List of coverage map pixels to read. Only used if input file is a HealSparseMap
- **header** (*bool*, optional) – Return the fits header metadata as well as map? Default is False.
- **degrade_nside** (*int*, optional) – Degrade map to this nside on read. None means leave as-is. Not yet implemented for parquet files.
- **weightfile** (*str*, optional) – Floating-point map to supply weights for degrade wmean. Must be a HealSparseMap (weighted degrade not supported for healpix degrade-on-read). Not yet implemented for parquet files.
- **reduction** (*str*, optional) – Reduction method with degrade-on-read. (mean, median, std, max, min, and, or, sum, prod, wmean). Not yet implemented for parquet files.
- **use_threads** (*bool*, optional) – Use multithreaded reading for parquet files.

Returns

- **healSparseMap** (*HealSparseMap*) – HealSparseMap from file, covered by pixels
- **header** (*fitsio.FITSHDR* or *astropy.io.fits* (if header=True)) – Fits header for the map file.

set_bits_pix(*pixels*, *bits*, *nest=True*)

Set bits of a wide_mask map.

Parameters

- **pixels** (*np.ndarray*) – Integer array of sparse_map pixel values
- **bits** (*list*) – List of bits to set

update_values_pix(*pixels*, *values*, *nest=True*, *operation='replace'*)

Update the values in the sparsemap for a list of pixels. The list of pixels must be unique if the operation is 'replace'.

Parameters

- **pixels** (*np.ndarray*) – Integer array of sparse_map pixel values
- **values** (*np.ndarray*) – Value or Array of values. Must be same type as sparse_map.
- **operation** (*str*, optional) – Operation to use to update values. May be 'replace' (default), 'or', or 'and' (for bit masks)

:raises ValueError : Raised if positions do not resolve to unique: positions and operation is 'replace'.

update_values_pos(*ra_or_theta*, *dec_or_phi*, *values*, *lonlat=True*, *operation='replace'*)

Update the values in the sparsemap for a list of positions.

Parameters

- **ra_or_theta** (*float*, array-like) – Angular coordinates of points on a sphere.
- **dec_or_phi** (*float*, array-like) – Angular coordinates of points on a sphere.
- **values** (*np.ndarray*) – Value or Array of values. Must be same type as sparse_map.
- **lonlat** (*bool*, optional) – If True, input angles are longitude and latitude in degrees. Otherwise, they are co-latitude and longitude in radians.
- **operation** (*str*, optional) – Operation to use to update values. May be 'replace' (default), 'or', or 'and' (for bit masks)

:raises ValueError : Raised if positions do not resolve to unique: positions and operation is 'replace'.

property valid_pixels

Get an array of valid pixels in the sparse map.

Returns valid_pixels

Return type *np.ndarray*

valid_pixels_pos(*lonlat=True*, *return_pixels=False*)

Get an array with the position of valid pixels in the sparse map.

Parameters

- **lonlat** (*bool*, optional) – If True, input angles are longitude and latitude in degrees. Otherwise, they are co-latitude and longitude in radians.
- **return_pixels** (*bool*, optional) – If true, return valid_pixels / co-lat / co-lon or valid_pixels / lat / lon instead of lat / lon

Returns positions – By default it will return a tuple of the form (*theta*, *phi*) in radians unless *lonlat = True*, for which it will return (*ra*, *dec*) in degrees. If *return_pixels = True*, valid_pixels will be returned as first element in tuple.

Return type *tuple*

property wide_mask_maxbits

Get the maximum number of bits stored in the wide mask.

Returns `wide_mask_maxbits` – Maximum number of bits. 0 if not wide mask.

Return type `int`

property wide_mask_width

Get the width of the wide mask

Returns `wide_mask_width` – Width of wide mask array. 0 if not wide mask.

Return type `int`

write(*filename*, *clobber=False*, *nocompress=False*, *format='fits'*, *nside_io=4*)

Write a HealSparseMap to a file. Use the *metadata* property from the map to persist additional information in the fits header.

Parameters

- **filename** (*str*) – Name of file to save
- **clobber** (*bool*, optional) – Clobber existing file? Default is False.
- **nocompress** (*bool*, optional) – If this is False, then integer maps will be compressed losslessly. Note that *np.int64* maps cannot be compressed in the FITS standard. This option only applies if *format='fits'*.
- **nside_io** (*int*, optional) – The healpix nside to partition the output map files in parquet. Must be less than or equal to *nside_coverage*, and not greater than 16. This option only applies if *format='parquet'*.
- **format** (*str*, optional) – File format. Currently only 'fits' is supported.

Raises

- **NotImplementedError** if file format is not supported. –
- **ValueError** if *nside_io* is out of range. –

12.1.2 HealSparseRandoms

healsparse.healSparseRandoms.make_uniform_randoms(*sparse_map*, *n_random*, *rng=None*)

Make an array of uniform randoms.

Parameters

- **sparse_map** (*healsparse.HealSparseMap*) – Sparse map object
- **n_random** (*int*) – Number of randoms to generate
- **rng** (*np.random.RandomState*, optional) – Pre-set Random number generator. Default is None.

Returns

- **ra_array** (*np.ndarray*) – Float array of RAs (degrees)
- **dec_array** (*np.ndarray*) – Float array of declinations (degrees)

healsparse.healSparseRandoms.make_uniform_randoms_fast(*sparse_map*, *n_random*,

nside_randoms=8388608, *rng=None*)

Make an array of uniform randoms.

Parameters

- **sparse_map** (*healsparse.HealSparseMap*) – Sparse map object
- **n_random** (*int*) – Number of randoms to generate
- **nside_randoms** (*int*, optional) – Nside for pixel centers to select random points
- **rng** (*np.random.RandomState*, optional) – Pre-set Random number generator. Default is `None`.

Returns

- **ra_array** (*np.ndarray*) – Float array of RAs (degrees)
- **dec_array** (*np.ndarray*) – Float array of declinations (degrees)

12.1.3 Operations

`healsparse.operations.and_intersection(map_list)`

Bitwise or a list of `HealSparseMaps` as an intersection. Only pixels that are valid in all the input maps will have valid values in the output. Only works on integer maps.

Parameters **map_list** (*list of HealSparseMap*) – Input list of maps to bitwise and

Returns **result** – Bitwise and of maps

Return type *HealSparseMap*

`healsparse.operations.and_union(map_list)`

Bitwise and a list of `HealSparseMaps` as a union. Empty values will be treated as 0s in the bitwise and, and the output map will have a union of all the input map pixels. Only works in integer maps.

Parameters **map_list** (*list of HealSparseMap*) – Input list of maps to bitwise and

Returns **result** – Bitwise and of maps

Return type *HealSparseMap*

`healsparse.operations.max_intersection(map_list)`

Element-wise maximum of the intersection of a list of the `HealSparseMaps`.

Parameters **map_list** (*list of HealSparseMap*) – Input list of maps to compute the maximum of

Returns **result** – Element-wise maximum of maps

Return type *HealSparseMap*

`healsparse.operations.max_union(map_list)`

Element-wise maximum of the union of a list of `HealSparseMaps`.

Parameters **map_list** (*list of HealSparseMap*) – Input list of maps to compute the maximum of

Returns **result** – Element-wise maximum of maps

Return type *HealSparseMap*

`healsparse.operations.min_intersection(map_list)`

Element-wise minimum of the intersection of a list of `HealSparseMaps`.

Parameters **map_list** (*list of HealSparseMap*) – Input list of maps to compute the minimum of

Returns **result** – Element-wise minimum of maps

Return type *HealSparseMap*

`healsparse.operations.min_union(map_list)`

Element-wise minimum of the union of a list of `HealSparseMaps`.

Parameters `map_list` (*list of HealSparseMap*) – Input list of maps to compute the minimum of

Returns `result` – Element-wise minimum of maps

Return type *HealSparseMap*

`healsparse.operations.or_intersection(map_list)`

Bitwise or a list of HealSparseMaps as an intersection. Only pixels that are valid in all the input maps will have valid values in the output. Only works on integer maps.

Parameters `map_list` (*list of HealSparseMap*) – Input list of maps to bitwise or

Returns `result` – Bitwise or of maps

Return type *HealSparseMap*

`healsparse.operations.or_union(map_list)`

Bitwise or a list of HealSparseMaps as a union. Empty values will be treated as 0s in the bitwise or, and the output map will have a union of all the input map pixels. Only works in integer maps.

Parameters `map_list` (*list of HealSparseMap*) – Input list of maps to bitwise or

Returns `result` – Bitwise or of maps

Return type *HealSparseMap*

`healsparse.operations.product_intersection(map_list)`

Compute the product of a list of HealSparseMaps as an intersection. Only pixels that are valid in all the input maps will have valid values in the output.

Parameters `map_list` (*list of HealSparseMap*) – Input list of maps to take the product

Returns `result` – Product of maps

Return type *HealSparseMap*

`healsparse.operations.product_union(map_list)`

Compute the product of a list of HealSparseMaps as a union. Empty values will be treated as 1s in the product, and the output map will have a union of all the input map pixels.

Parameters `map_list` (*list of HealSparseMap*) – Input list of maps to take the product

Returns `result` – Product of maps

Return type *HealSparseMap*

`healsparse.operations.sum_intersection(map_list)`

Sum a list of HealSparseMaps as an intersection. Only pixels that are valid in all the input maps will have valid values in the output.

Parameters `map_list` (*list of HealSparseMap*) – Input list of maps to sum

Returns `result` – Summation of maps

Return type *HealSparseMap*

`healsparse.operations.sum_union(map_list)`

Sum a list of HealSparseMaps as a union. Empty values will be treated as 0s in the summation, and the output map will have a union of all the input map pixels.

Parameters `map_list` (*list of HealSparseMap*) – Input list of maps to sum

Returns `result` – Summation of maps

Return type *HealSparseMap*

`healsparse.operations.ufunc_intersection(map_list, func, filler_value=0)`

Apply numpy ufunc to the intersection of a list of HealSparseMaps.

Parameters

- **map_list** (*list of HealSparseMap*) – Input list of maps where the operation is applied
- **func** (*np.ufunc*) – Numpy universal function to apply
- **filler_value** (*int or float*) – Starting value

Returns result – Resulting map

Return type *HealSparseMap*

`healsparse.operations.ufunc_union(map_list, func, filler_value=0)`

Apply numpy ufunc to the union of a list of HealSparseMaps.

Parameters

- **map_list** (*list of HealSparseMaps*) – Input list of maps where the operation is applied
- **func** (*np.ufunc*) – Numpy universal function to apply
- **filler_value** (*int or float*) – Starting value and filler for the union

Returns result – Resulting map

Return type *HealSparseMap*

`healsparse.operations.xor_intersection(map_list)`

Bitwise xor a list of HealSparseMaps as an intersection. Only pixels that are valid in all the input maps will have valid values in the output. Only works on integer maps.

Parameters map_list (*list of HealSparseMap*) – Input list of maps to bitwise xor

Returns result – Bitwise xor of maps

Return type *HealSparseMap*

`healsparse.operations.xor_union(map_list)`

Bitwise xor a list of HealSparseMaps as a union. Empty values will be treated as 0s in the bitwise or, and the output map will have a union of all the input map pixels. Only works in integer maps.

Parameters map_list (*list of HealSparseMap*) – Input list of maps to bitwise xor

Returns result – Bitwise xor of maps

Return type *HealSparseMap*

12.1.4 Geometry Library

`class healsparse.geom.Circle(*, ra, dec, radius, value)`

Bases: [healsparse.geom.GeomBase](#)

`__init__(*, ra, dec, radius, value)`

Parameters

- **ra** (*float*) – ra in degrees (scalar-only)
- **dec** (*float*) – dec in degrees (scalar-only)
- **radius** (*float*) – radius in degrees (scalar-only)

- **value** (*number*) – Value for pixels in the map (scalar or list of bits for *wide_mask*)

__repr__()

Return repr(self).

property dec

get the dec value

get_pixels(**, nside*)

get the pixels associated with this circle

Parameters *nside* (*int*) – Nside for the pixels

property ra

get the ra value

property radius

get the radius value

class `healsparse.geom.GeomBase`

Bases: `object`

base class for geometric objects that can convert themselves to maps

__weakref__

list of weak references to the object (if defined)

get_map(**, nside_coverage, nside_sparse, dtype, wide_mask_maxbits=None*)

get a healsparse map corresponding to this geometric primitive

Parameters

- **nside_coverage** (*int*) – nside of coverage map
- **nside_sparse** (*int*) – nside of sparse map
- **dtype** (*np.dtype*) – dtype of the output array
- **wide_mask_maxbits** (*int*, optional) – Create a “wide bit mask” map, with this many bits.

Returns

Return type *HealSparseMap*

get_map_like(*sparseMap*)

Get a healsparse map corresponding to this geometric primitive, with the same parameters as an input *sparseMap*.

Parameters *sparseMap* (*healsparse.HealSparseMap*) – Input map to match parameters

Returns

Return type *HealSparseMap*

get_pixels(**, nside*)

get pixels for this map

Parameters *nside* (*int*) – Nside for the pixels

property is_integer_value

Check if the value is an integer type

property value

get the value to be used for all pixels in the map

class `healsparse.geom.Polygon`(**, ra, dec, value*)

Bases: *healsparse.geom.GeomBase*

`__init__(*, ra, dec, value)`

represent a polygon

both counter clockwise and clockwise order for polygon vertices works

Parameters

- **ra** (*array*) – ra of vertices in degrees, size [nvert]
- **dec** (*array*) – dec of vertices in degrees, size [nvert]
- **value** (*number*) – Value for pixels in the map

`__repr__()`

Return repr(self).

property dec

get the dec value

`get_pixels(*, nside)`

get the pixels associated with this polygon

Parameters **nside** (*int*) – Nside for the pixels

property ra

get the ra value

property vertices

get the dec value

`healsparse.geom.realize_geom(geom, smap, type='or')`

Realize geometry objects in a map

Parameters

- **geom** (*geometric primitive or list thereof*) – List of Geom objects, e.g. Circle, Polygon
- **smap** (*HealSparseMaps*) – The map in which to realize the objects
- **type** (*string*) – Way to combine the list of geometric objects. Default is to “or” them

12.1.5 HealSparseCoverage

`class healsparse.healSparseCoverage.HealSparseCoverage(cov_index_map, nside_sparse)`

Bases: object

Class to define a HealSparseCoverage map.

Parameters

- **cov_index_map** (*np.ndarray*) – Coverage map with pixel indices.
- **nside_sparse** (*int*) – Healpix nside of the sparse map.

Returns **cov_map** – HealSparseCoverage map.

Return type *HealSparseCoverage*

`__init__(cov_index_map, nside_sparse)`

`__weakref__`

list of weak references to the object (if defined)

append_pixels(*sparse_map_size*, *new_cov_pix*, *check=True*, *copy=True*)

Append new pixels to the coverage map

Parameters

- **sparse_map_size** (*int*) – Size of current sparse map
- **new_cov_pix** (*np.ndarray*) – Array of new coverage pixels

property bit_shift

Get the `bit_shift` for the coverage map

Returns **bit_shift** – Number of bits to shift from coarse to fine maps

Return type *int*

cov_pixels(*sparse_pixels*)

Get coverage pixel numbers (nest) from a set of sparse pixels.

Parameters **sparse_pixels** (*np.ndarray*) – Array of sparse pixels

Returns **cov_pixels** – Coverage pixel numbers (nest format)

Return type *np.ndarray*

cov_pixels_from_index(*index*)

Get the coverage pixels from the sparse map index.

Parameters **index** (*np.ndarray*) – Array of indices in sparse map

Returns **cov_pixels** – Coverage pixel numbers (nest format)

Return type *np.ndarray*

property coverage_mask

Get the boolean mask of the coverage map.

Returns **cov_mask** – Boolean array of coverage mask.

Return type *np.ndarray*

initialize_pixels(*cov_pix*)

Initialize pixels in the index map

Parameters **cov_pix** (*np.ndarray*) – Array of coverage pixels

classmethod make_empty(*nside_coverage*, *nside_sparse*)

Make an empty coverage map.

Parameters

- **nside_coverage** (*int*) – Healpix nside for the coverage map
- **nside_sparse** (*int*) – Healpix nside for the sparse map

Returns **healSparseCoverage** – HealSparseCoverage from file

Return type *HealSparseCoverage*

classmethod make_from_pixels(*nside_coverage*, *nside_sparse*, *cov_pixels*)

Make an empty coverage map.

Parameters

- **nside_coverage** (*int*) – Healpix nside for the coverage map
- **nside_sparse** (*int*) – Healpix nside for the sparse map
- **cov_pixels** (*np.ndarray*) – Array of coverage pixels

Returns `healSparseCoverage` – HealSparseCoverage from file

Return type *HealSparseCoverage*

property `nfine_per_cov`

Get the number of fine (sparse) pixels per coarse (coverage) pixel

Returns `nfine_per_cov` – Number of fine (sparse) pixels per coverage pixel

Return type *int*

property `nside_coverage`

Get the nside of the coverage map

Returns `nside_coverage`

Return type *int*

property `nside_sparse`

Get the nside of the associated sparse map

Returns `nside_sparse`

Return type *int*

classmethod `read(filename_or_fits, use_threads=False)`

Read in a HealSparseCoverage map from a file.

Parameters

- **coverage_class** (*type*) – Type value of the HealSparseCoverage class.
- **filename_or_fits** (*str* or *HealSparseFits*) – Name of filename or already open *HealSparseFits* object.
- **use_threads** (*bool*, optional) – Use multithreaded reading for parquet files. Should not be necessary for coverage maps.

Returns `cov_map` – HealSparseCoverage map from file.

Return type *HealSparseCoverage*

12.1.6 Concatenation

`healsparse.cat_healsparse_files.cat_healsparse_files(file_list, outfile, check_overlap=False, clobber=False, in_memory=False, nside_coverage_out=None, or_overlap=False)`

Concatenate healsparse files together in a memory-efficient way.

Parameters

- **file_list** (*list of str*) – List of filenames to concatenate
- **outfile** (*str*) – Output filename
- **check_overlap** (*bool*, optional) – Check that each file has a unique sparse map. This may be slower.
- **clobber** (*bool*, optional) – Clobber existing outfile
- **in_memory** (*bool*, optional) – Do operations in-memory (required unless fitsio is available).
- **nside_coverage_out** (*int*, optional) – Output map with specific nside_coverage. Default is nside_coverage of first map in file_list.

- **or_overlap** (*bool*, optional) – If True compute the *or* overlap of two integer maps when concatenating.

CHAPTER
THIRTEEN

SEARCH

- search

PYTHON MODULE INDEX

h

`healsparse.cat_healsparse_files`, 47
`healsparse.geom`, 43
`healsparse.healSparseCoverage`, 45
`healsparse.healSparseMap`, 31
`healsparse.healSparseRandoms`, 40
`healsparse.operations`, 41

Symbols

- `__add__` () (*healsparse.healSparseMap.HealSparseMap* method), 31
 - `__and__` () (*healsparse.healSparseMap.HealSparseMap* method), 31
 - `__getitem__` () (*healsparse.healSparseMap.HealSparseMap* method), 31
 - `__iadd__` () (*healsparse.healSparseMap.HealSparseMap* method), 31
 - `__iand__` () (*healsparse.healSparseMap.HealSparseMap* method), 31
 - `__imul__` () (*healsparse.healSparseMap.HealSparseMap* method), 31
 - `__init__` () (*healsparse.geom.Circle* method), 43
 - `__init__` () (*healsparse.geom.Polygon* method), 44
 - `__init__` () (*healsparse.healSparseCoverage.HealSparseCoverage* method), 45
 - `__init__` () (*healsparse.healSparseMap.HealSparseMap* method), 31
 - `__ior__` () (*healsparse.healSparseMap.HealSparseMap* method), 32
 - `__ipow__` () (*healsparse.healSparseMap.HealSparseMap* method), 32
 - `__isub__` () (*healsparse.healSparseMap.HealSparseMap* method), 32
 - `__itruediv__` () (*healsparse.healSparseMap.HealSparseMap* method), 32
 - `__ixor__` () (*healsparse.healSparseMap.HealSparseMap* method), 32
 - `__mul__` () (*healsparse.healSparseMap.HealSparseMap* method), 32
 - `__or__` () (*healsparse.healSparseMap.HealSparseMap* method), 32
 - `__pow__` () (*healsparse.healSparseMap.HealSparseMap* method), 32
 - `__repr__` () (*healsparse.geom.Circle* method), 44
 - `__repr__` () (*healsparse.geom.Polygon* method), 45
 - `__repr__` () (*healsparse.healSparseMap.HealSparseMap* method), 33
 - `__setitem__` () (*healsparse.healSparseMap.HealSparseMap* method), 33
 - `__str__` () (*healsparse.healSparseMap.HealSparseMap* method), 33
 - `__sub__` () (*healsparse.healSparseMap.HealSparseMap* method), 33
 - `__truediv__` () (*healsparse.healSparseMap.HealSparseMap* method), 33
 - `__weakref__` (*healsparse.geom.GeomBase* attribute), 44
 - `__weakref__` (*healsparse.healSparseCoverage.HealSparseCoverage* attribute), 45
 - `__weakref__` (*healsparse.healSparseMap.HealSparseMap* attribute), 33
 - `__xor__` () (*healsparse.healSparseMap.HealSparseMap* method), 33
- ## A
- `and_intersection()` (in module *healsparse.operations*), 41
 - `and_union()` (in module *healsparse.operations*), 41
 - `append_pixels()` (*healsparse.healSparseCoverage.HealSparseCoverage* method), 45
 - `apply_mask()` (*healsparse.healSparseMap.HealSparseMap* method), 33
 - `astype()` (*healsparse.healSparseMap.HealSparseMap* method), 33
- ## B
- `Map_shift` (*healsparse.healSparseCoverage.HealSparseCoverage* property), 46
- ## C
- `cat_healsparse_files()` (in module *healsparse.cat_healsparse_files*), 47
 - `check_bits_pix()` (*healsparse.healSparseMap.HealSparseMap* method), 33
 - `check_bits_pos()` (*healsparse.healSparseMap.HealSparseMap* method), 34
 - `Circle` (class in *healsparse.geom*), 43
 - `clear_bits_pix()` (*healsparse.healSparseMap.HealSparseMap* method), 34
 - `convert_healpix_map()` (*healsparse.healSparseMap.HealSparseMap* static method), 34

- `cov_pixels()` (*healsparse.healSparseCoverage.HealSparseCoverage* method), 46
- `cov_pixels_from_index()` (*healsparse.healSparseCoverage.HealSparseCoverage* method), 46
- `coverage_map` (*healsparse.healSparseMap.HealSparseMap* property), 34
- `coverage_mask` (*healsparse.healSparseCoverage.HealSparseCoverage* property), 46
- `coverage_mask` (*healsparse.healSparseMap.HealSparseMap* property), 34
- ## D
- `dec` (*healsparse.geom.Circle* property), 44
- `dec` (*healsparse.geom.Polygon* property), 45
- `degrade()` (*healsparse.healSparseMap.HealSparseMap* method), 35
- `dtype` (*healsparse.healSparseMap.HealSparseMap* property), 35
- ## F
- `fracdet_map()` (*healsparse.healSparseMap.HealSparseMap* method), 35
- ## G
- `generate_healpix_map()` (*healsparse.healSparseMap.HealSparseMap* method), 35
- `GeomBase` (class in *healsparse.geom*), 44
- `get_map()` (*healsparse.geom.GeomBase* method), 44
- `get_map_like()` (*healsparse.geom.GeomBase* method), 44
- `get_pixels()` (*healsparse.geom.Circle* method), 44
- `get_pixels()` (*healsparse.geom.GeomBase* method), 44
- `get_pixels()` (*healsparse.geom.Polygon* method), 45
- `get_single()` (*healsparse.healSparseMap.HealSparseMap* method), 35
- `get_valid_area()` (*healsparse.healSparseMap.HealSparseMap* method), 36
- `get_values_pix()` (*healsparse.healSparseMap.HealSparseMap* method), 36
- `get_values_pos()` (*healsparse.healSparseMap.HealSparseMap* method), 36
- ## H
- `healsparse.cat_healsparse_files` module, 47
- `healsparse.geom` module, 43
- `healsparse.healSparseCoverage` module, 45
- `healsparse.healSparseMap` module, 31
- `healsparse.healSparseRandoms` module, 40
- `healsparse.operations` module, 41
- `HealSparseCoverage` (class in *healsparse.healSparseCoverage*), 45
- `HealSparseMap` (class in *healsparse.healSparseMap*), 31
- `initialize_pixels()` (*healsparse.healSparseCoverage.HealSparseCoverage* method), 46
- `is_integer_map` (*healsparse.healSparseMap.HealSparseMap* property), 36
- `is_integer_value` (*healsparse.geom.GeomBase* property), 44
- `is_rec_array` (*healsparse.healSparseMap.HealSparseMap* property), 36
- `is_unsigned_map` (*healsparse.healSparseMap.HealSparseMap* property), 37
- `is_wide_mask_map` (*healsparse.healSparseMap.HealSparseMap* property), 37
- ## M
- `make_empty()` (*healsparse.healSparseCoverage.HealSparseCoverage* class method), 46
- `make_empty()` (*healsparse.healSparseMap.HealSparseMap* class method), 37
- `make_empty_like()` (*healsparse.healSparseMap.HealSparseMap* class method), 37
- `make_from_pixels()` (*healsparse.healSparseCoverage.HealSparseCoverage* class method), 46
- `make_uniform_randoms()` (in module *healsparse.healSparseRandoms*), 40
- `make_uniform_randoms_fast()` (in module *healsparse.healSparseRandoms*), 40
- `max_intersection()` (in module *healsparse.operations*), 41
- `max_union()` (in module *healsparse.operations*), 41
- `metadata` (*healsparse.healSparseMap.HealSparseMap* property), 38
- `min_intersection()` (in module *healsparse.operations*), 41
- `min_union()` (in module *healsparse.operations*), 41
- module
- `healsparse.cat_healsparse_files`, 47
 - `healsparse.geom`, 43
 - `healsparse.healSparseCoverage`, 45
 - `healsparse.healSparseMap`, 31
 - `healsparse.healSparseRandoms`, 40
 - `healsparse.operations`, 41

N

`nfine_per_cov` (*healsparse.healSparseCoverage.HealSparseCoverage* property), 47

`nside_coverage` (*healsparse.healSparseCoverage.HealSparseCoverage* property), 47

`nside_coverage` (*healsparse.healSparseMap.HealSparseMap* property), 38

`nside_sparse` (*healsparse.healSparseCoverage.HealSparseCoverage* property), 47

`nside_sparse` (*healsparse.healSparseMap.HealSparseMap* property), 38

O

`or_intersection()` (in module *healsparse.operations*), 42

`or_union()` (in module *healsparse.operations*), 42

P

`Polygon` (class in *healsparse.geom*), 44

`primary` (*healsparse.healSparseMap.HealSparseMap* property), 38

`product_intersection()` (in module *healsparse.operations*), 42

`product_union()` (in module *healsparse.operations*), 42

R

`ra` (*healsparse.geom.Circle* property), 44

`ra` (*healsparse.geom.Polygon* property), 45

`radius` (*healsparse.geom.Circle* property), 44

`read()` (*healsparse.healSparseCoverage.HealSparseCoverage* class method), 47

`read()` (*healsparse.healSparseMap.HealSparseMap* class method), 38

`realize_geom()` (in module *healsparse.geom*), 45

S

`set_bits_pix()` (*healsparse.healSparseMap.HealSparseMap* method), 38

`sum_intersection()` (in module *healsparse.operations*), 42

`sum_union()` (in module *healsparse.operations*), 42

U

`ufunc_intersection()` (in module *healsparse.operations*), 42

`ufunc_union()` (in module *healsparse.operations*), 43

`update_values_pix()` (*healsparse.healSparseMap.HealSparseMap* method), 39

`update_values_pos()` (*healsparse.healSparseMap.HealSparseMap* method), 39

V

`val_circles` (*healsparse.healSparseMap.HealSparseMap* property), 39

`val_circles_pos()` (*healsparse.healSparseMap.HealSparseMap* method), 39

`value` (*healsparse.geom.GeomBase* property), 44

`vertices` (*healsparse.geom.Polygon* property), 45

W

`wide_mask_maxbits` (*healsparse.healSparseMap.HealSparseMap* property), 39

`wide_mask_width` (*healsparse.healSparseMap.HealSparseMap* property), 40

`write()` (*healsparse.healSparseMap.HealSparseMap* method), 40

X

`xor_intersection()` (in module *healsparse.operations*), 43

`xor_union()` (in module *healsparse.operations*), 43