Partpix Documentation
Anne Bauer
Nov 22, 2012

Introduction

Partpix is a type of Healpix map that I have implemented so that it is possible to use a high-resolution map over part of the sky only. This is useful because a high resolution map over the whole sky takes a lot of memory and often we only have data (or simulations) over a fraction of the sky. I have implemented the changes in the C++ version of Healpix.

I am assuming that you are somewhat familiar with the current workings of the C++ implementation of Healpix. See the documentation here:
http://healpix.jpl.nasa.gov/html/Healpix_cxx/components.html

Healpix_Base

`Healpix_Base` is a class that includes all the functions dealing with pixel numbers, resolutions, conversions between pixel numbers and sky positions, etc. Basically, it has everything involving Healpix maps that does not require the actual data values in the map. As such, it includes the information about the order, nside, and scheme.

Healpix_Map

`Healpix_Map` is a class that inherits `Healpix_Base`, and therefore can use all of its members and member functions. The only additional thing that Healpix_Map includes is the actual pixel array, which is of type `arr`, and is called `map`. `arr` is a type defined in cxxsupport/arr.h inside Healpix, and it's basically a normal array (not a C++ container like `vector`). When you create a `Healpix_Map` object in your code and fill it with a value, the array is allocated with length `Npix()`, where `Npix()` is the number of pixels in the whole sky at the resolution of your map. (Herein lies the problem.) When you use a `Healpix_Map`, for example `my_map`, in your code you can access the data in pixel `i` by saying `my_map[i]`. This simply gets the value `map[i]` from `arr map`.

Some `Healpix_Map` functions:

The constructor `Healpix_Map(order, scheme)` creates a map with the given resolution and pixel scheme.

`fill(value)` assigns every pixel in the sky the given value.

`Import(Healpix_Map original_map)` assigns the data from the given healpix map into the current one, changing the resolution and scheme if necessary. In fact it is a generalization of 3 different functions: `Import_degrade`, `Import_nograde`, and `Import_upgrade` depending on how the original map's resolution compares to the current map's.

`[i]` returns `map[i]`, the value at pixel `i`.

Mathematical functions such as `average(), minmax(), Add(val), Scale(val)` calculate simple math on the map.

Partpix_Map

`Partpix_Map` is a class that inherits `Healpix_Base`, and therefore can use all of its members and member functions, just as `Healpix_Map` can. In addition, it has a pixel array `arr partmap` instead of the `arr map` in `Healpix_Map`. The partmap array is of length `Npartpix()`, which is less than or equal to `Npix()`. Meaning, the array holds pixel data for a subset of the sky. We need to record which pixel number (in the Healpix pixel ID scheme) corresponds to which array index in `partmap`. For this we have `vector pixel_mapping_arraytohigh`, which is also length `Npartpix()`, such that `pixel_mapping_arraytohigh[partmap_index] = healpix_pixel_ID` for each pixel in the `Partpix_Map`.

The `Partpix_Map` functions that I have implemented so far are:

The constructor `Partpix_Map(order, Healpix_Map footprint_map)` constructs a map with the resolution given by order, with pixels that only exist in the regions where footprint_map > 0.5. Typically, `footprint_map` would be a low-resolution healpix map; if it the same resolution as the Partpix_Map you are creating, you will not be saving any memory since you'll need to load up footprint_map in your code! While using the Partpix_Map, **it is important not to query any pixels for which footprint_map<0.5**. Doing this will throw an exception and quit the program. Inside this constructor, the number of existing pixels `Npartpix()` is figured out by looking at how many pixels of the desired order fit inside the footprint, the partmap array is defined with length `Npartpix()`, and the mapping vector is filled in with the relationship between the Healpix pixel numbers and the `partmap` array indices.

`[i]` returns the data value at the location of the high resolution pixel index `i`. This means that the pixel `i` is at the same location in a `Partpix_Map` as in a `Healpix_Map`. Saying `my_map[i]` returns the result of `partmap_at_highresindex(i)` (see below).

`partmap_at_highresindex(high_res_index)` finds the data value that corresponds to the high resolution index given. It does this by looking in `partmap[low_res_index]`, where `pixel_mapping_arraytohigh[low_res_index] = high_res_index`. To find the low resolution index that corresponds to the high resolution index, a binary search is performed in the pixel_mapping vector. (Because the pixel_mapping vector is constructed by looping over the high resolution pixel indices, it is already in ascending order and can be binary searched.) In terms of resources used by `Healpix_Map` vs. `Partpix_Map`, we're therefore trading the memory taken by a full-length array for the time it takes to search through a smaller array.

`highResPix(int low_res_index)` returns `pixel_mapping_arraytohigh[low_res_pix]`. **This is very useful** because often, at least in correlation function code, you want to loop over all the pixels. One of the advantages of a

`Partpix_Map` is that you only have to loop over the existing `Npartpix()` pixels, saving you a lot of time. This means that you would loop pixnum from 0 to `Npartpix()`, but then before you could use a value `my_map[j]` you would need to call `j = highResPix(i)` since the map is queried using the high resolution pixel indices.

`to_Healpix(default_val)` returns a `Healpix_Map` version of the current `Partpix_Map`, with `default_val` used as the pixel values of all the pixels that do not exist in the `Partpix_Map`.

`Import(Partpix_Map original_map, Healpix_Map footprint_map)` is implemented for `Import_upgrade` and `Import_nograde`, but not `Import_degrade`.

The simple math functions like `Add(val)`, `rms()` etc. work the same as in `Healpix_Map`, except the actions are only applied to the existing pixels.

An Example

Here is a simple example code I have used for testing.

```
#include <unistd.h>
#include <vector>
using std::vector;
#include <string>
using std::string;
#include <iostream>
using std::cout;
using std::cerr;
using std::endl;
#include <healpix_base.h>
#include <healpix_map.h>

#include <fitsio.h>
#include <fitshandle.h>

#include <healpix_map_fitsio.h>
#include <healpix_data_io.h>

#include <partpix_map.h>


int main (int argc, char const *argv[]){

   Healpix_Map<double> oldmap(7, RING);
   oldmap.fill(0.);
   for( int i=0; i<19661; ++i ){
      oldmap[i]=1.0;
   }
```

```
cout << "my Healpix map has order " << oldmap.Order() << "!" << endl;
cerr << "sleeping!" << endl;
sleep(5);  // i had it sleep so i could keep an eye on the memory usage

Partpix_Map<double> newmap(9, oldmap);
cout << "my Partpix map has order " << newmap.Order() << "!!!" << endl;
// now fill up the new map with data, using the old one as the mask (NECESSARY).
double val = 1.0;
for( int i=0; i<newmap.Npix(); ++i ){
   pointing mypointing = newmap.pix2ang(i);
   int pixnum = oldmap.ang2pix(mypointing);
   if( oldmap[pixnum] > 0.5 ){
      newmap[i] = val;
      val += 0.1;
   }
}
cerr << "sleeping again!" << endl;
sleep(5);

// now let's loop over the partpix pixels...
cout << "The map values:" << endl;
for( int i=0; i<newmap.Npartpix(); ++i ){
   int j=newmap.highResPix(i);
   double val = newmap[j];
   cout << val << " ";
}
cout << endl;

Healpix_Map<double> outmap = newmap.to_Healpix( 0. );

Partpix_Map<double> importmap(10, oldmap);
importmap.Import_upgrade(newmap, oldmap);
Healpix_Map<double> importhmap = importmap.to_Healpix( 0. );

system( "rm inmap.fits" );
fitshandle myfits = fitshandle();
myfits.create("inmap.fits");
write_Healpix_map_to_fits(myfits, oldmap, PLANCK_FLOAT64);
myfits.close();

system( "rm outmap.fits" );
myfits = fitshandle();
myfits.create("outmap.fits");
write_Healpix_map_to_fits(myfits, outmap, PLANCK_FLOAT64);
myfits.close();

system( "rm importmap.fits" );
myfits = fitshandle();
```

```
    myfits.create("importmap.fits");
    write_Healpix_map_to_fits(myfits, importhmap, PLANCK_FLOAT64);
    myfits.close();

    return 0;
}
```