HPixLib Documentation

Release 0.1

Maurizio Tomasi

June 30, 2014

Contents

Contents:

CHAPTER 1

Introduction

This manual describes HPixLib, a C library that implements the Healpix spherical tesselation scheme described by Górski et al. (The Astrophysical Journal, 622:759-771, 2005).

The purpose of HEALPix is to divide a sphere into patches of equal area which have desirable properties for a number of calculations. It is mostly used in astrophysics and cosmology, e.g.:

- Computation of the spherical Fourier transforms for a random field on the sphere;
- Efficient nearest-neighbour searches of pixels.

Traditionally, there has always been *one* implementation of the HEALPix tesselation scheme: the so-called `standard` HEALPix library.

1.1 The standard Healpix implementation

The standard implementation of the HEALPix tesselation scheme is the HEALPix library developed by Gorski et al. (http://healpix.jpl.nasa.gov/). It provides a number of functions in Fortran, C, C++, IDL and Java and is so far the most complete, tested and widely used implementation.

HPixLib has not been designed to compete with the Healpix library. It should instead be considered as an orthogonal project, which tries to address some of the following issues, without trying to re-implement everything:

- In Healpix there are no connections among the language bindings. (In fact, instead of speaking of "bindings" – which imply that there is one library and many interfaces that binds to it – one would rather speak of four different implementations: one for each language.) Apart from some code duplication, this has had the effect of reducing the momentum in developing some of them. The C bindings appear to be the most neglected, for there is no facility for reading/saving/doing calculations on $a_{\ell m}$ coefficients. This is particularly limiting for those developers wanting to create bindings to other languages by exploiting Foreign Function Interfaces (FFIs), as C is the *lingua franca* used for FFIs.
- The fact that the C binding is the most neglected is however unfortunate, as this is the standard de facto for writing bindings to other languages (e.g. Python, GNU R...).
- The installation of HEALPix is not straightforward: the library requires the user to install CFITSIO first, which is a rather large library if compared with what Healpix uses (the largest part of the CFITSIO code implements functions for reading/writing images, while HEALPix only reads and write binary tables). Moreover, there is no standard facility for a program using HEALPix to find and link the Healpix library (i.e. no support for pkg-config).
- No facilities to draw maps are provided in the C/C++ library. (A shortcut is to can use the standalone programs map2tga or map2gif to create a file which you then read back in your program.)

1.2 Purpose of HPixLib with respect to Healpix

HPixLib is meant to solve these issues. Of course, in order to do this I had to do a few compromises, so that a number of users of the Górski's library should stay with it. Here are the advantages of HPixLib:

- Only the C language is supported, and functions are more "low-level". This reduces the size of the library and eases its development, at the expense of loosing the majority of the scientists (which usually use Fortran, IDL or Python).
- It only supports maps of double values (the C++ bindings of the Healpix library use template and the user can therefore create maps of ints, booleans and so on). Since any 32-bit integer can be represented exactly in a 8-byte double, this means that precision is rarely an issue. However, using this approach you can easily waste lot of memory.
- Although it is meant as a basis for creating bindings to other languages, HPixLib itself only provides C bindings (i.e. no Fortran/IDL support). Also, even if you can use HPixLib in a C++ program, the library is not going to use all those nice features of C++ like std::vector and templates.
- The library provides an extensible interface to draw maps (as well as a standalone program, map2fig, which is able to produce bitmapped graphics as well as vector graphics).
- The library uses the GNU Autotools to configure itself and supports pkg-config.

CHAPTER 2

Installing HPixLib

HPixLib uses the GNU Autotools for compiling its sources (http://en.wikipedia.org/wiki/GNU_build_system), so there are relatively little dependencies to satisfy in order to bootstrap the compilation.

If you obtained a .tar.gz file, you should simply untar it somewhere in your home, enter the directory `hpixlib-???` (where *???* is the version number) and run the following commands:

```
./configure make && sudo make install
```

If you have cloned the Git repository (https://github.com/ziotom78/hpixlib), then you first have to run autogen.sh:

```
./autogen.sh ./configure make && sudo make install
```

This step requires your system to have the autotools (i.e. autoconf, automake, libtool and m4) already installed.

CHAPTER 3

Using the library

To use HPixLib in your C/C++ program, include the header file `hpixlib/hpix.h` at the beginning of your code:

```
#include <hpixlib/hpix.h>
```

To compile the library, you can use pkg-config:

```
cc my_program.c `pkg-config --cflags --libs libhpix`
```

Some parts of HPixLib can take advantage of the Cairo library. To use them, you need to refer to the pkg-config file `libhpix_cairo` (instead of just `libhpix`) and include the header file `hpixlib/hpix-cairo.h` as well:

```
#include <hpixlib/hpix.h>
#include <hpixlib/hpix-cairo.h>
```

CHAPTER 4

Pixel functions

In this section we describe the functions implemented by HPixLib that allow to associate points on the sky sphere with pixels in the HEALPix tessellation. These functions are the core of the library; HPixLib uses the same algorithms implemented in the C++ bindings (version 3.00) of the reference Healpix library.

4.1 Converting NSIDE into the number of pixels and back

The Healpix tessellation subdivides the sphere in a set of pixels of equal area. The number of pixels is uniquely specified through a positive integer parameter, *nside*, which is related to the number of pixels through a well-defined mathematical expression, implemented by the function `hpix_nside_to_npixel()` (the inverse calculation is implemented by `hpix_npixel_to_nside()`). The value of *nside* must be an integer power of two. To check if a given integer value satisfies these condition, HPixLib implements the function `hpix_valid_nside()`:

```
hpix_nside_t nside;
printf("Enter a value for nside: ");
scanf("%u", &nside);
if(hpix_valid_nside(nside)) {
  printf("The number of pixels in the map is %u\n",
         hpix_nside_to_npixel(nside));
} else {
  printf("Invalid value for nside.\n");
}
```

_Bool **hpix_valid_nside** (hpix_nside_t *nside*)
Return nonzero if the value of *nside* satisfies the following conditions:

1. It is an integer greater than zero;
2. It is an integer power of two.

hpix_pixel_num_t **hpix_nside_to_npixel** (hpix_nside_t)
Given a value for the *nside* parameter (any positive power of two), return the number of pixels the sky sphere is divided into. If *nside* is not valid, the function returns zero.

This function is the inverse of `hpix_npixel_to_nside()`.

```
hpix_pixel_num_t num;
assert((num = hpix_nside_to_npixel(8)) > 0);
```

hpix_nside_t **hpix_npixel_to_nside** (hpix_pixel_num_t)
Given the number of pixels in the sky sphere, this function returns the value of *NSIDE* uniquely associated with it. The function does not accept arbitrary values for *num_of_pixels*: any invalid value will make the function return zero.

This function is the inverse of hpix_nside_to_npixel().

4.2 Converting among angles, vectors and pixel centers

The following functions implement conversions between three different representations of points on a sphere:

1. Angular positions. These are expressed by *theta* (colatitude, from 0 to pi) and *phi* (longitude, from 0 to 2pi).
2. Versors (vectors normalized to have length one). These are expressed by three coordinates, *x*, *y*, *z*, with the constraint that *x*x + y*y + z*z == 1.0*.
3. Index of pixel centers. These can either be expressed using Healpix' *RING* or *NEST* numbering scheme, so technically it is not one but two representations.

It is important to note that any conversion involving pixel centers is only approximate, e.g. you cannot convert *theta* and *phi* into a pixel index and then back to *theta* and *phi*, and expect to get the same values.

The following example shows how to identify the pixel in a map which corresponds to a given coordinate pair. Note that you must ensure that the map is expressed in the same coordinate system as the angle you are providing: in the example, the position of M42 is specified in Galactic coordinates, and therefore the map must have been created using this coordinate system as well.

```
const hpix_nside_t NSIDE = 64;
const double DEG2RAD = 0.01745;
hpix_resolution_t resol;

/* Position of OriA in Galactic coordinates (degrees) */
double M42_position[] = { 209.01, -19.38 };

/* Convert the latitude in colatitude */
M42_position[0] = M42_position[0] - 180.0;

/* Here we assume to work with maps in RING order */
hpix_init_resolution_from_nside(NSIDE, &resol);
hpix_pixel_num_t pixel_index =
    hpix_angles_to_ring_pixel(&resol,
                              M42_position[0] * DEG2RAD,
                              M42_position[1] * DEG2RAD);
```

The key data structure is hpix_resolution_t, which contains the value of NSIDE as well as a number of other values derived from it and useful in the calculations.

void **hpix_init_resolution_from_nside** (hpix_nside_t *nside*, hpix_resolution_t * *resolution*)
Initialize the fields of *resolution* with values corresponding to the NSIDE parameter specified by *nside*. The object *resolution* can be allocated either on the stack or on the heap. (In the latter case, you must free it by yourself.)

4.2.1 Converting angular positions

The functions described in this paragraph convert angular positions (*theta*, *phi*) into some other representation.

void **hpix_angles_to_vector** (double *theta*, double *phi*, double * *x*, double * *y*, double * *z*)
Convert the pair of angles *theta*, *phi* into a versor (one-length vector) *x*, *y*, *z*. The function normalizes the angles before applying the conversion (e.g. if *phi* is equal to 3pi, it is converted into pi).

See also hpix_vector_to_angles().

hpix_pixel_num_t **hpix_angles_to_ring_pixel** (const hpix_resolution_t * *resolution*, double *theta*, double *phi*)

Convert the pair of angles *theta*, *phi* into the *RING* index of the pixel for which the specified direction falls within.

See also hpix_angles_to_nest_pixel().

hpix_pixel_num_t **hpix_angles_to_nest_pixel** (const hpix_resolution_t * *resolution*, double *theta*, double *phi*)

Convert the pair of angles *theta*, *phi* into the *NESTED* index of the pixel for which the specified direction falls within.

See also hpix_angles_to_ring_pixel().

typedef hpix_pixel_num_t **hpix_angles_to_pixel_fn_t** (const hpix_resolution_t *, double, double)

This defines a name for the prototype of the two functions hpix_angles_to_ring_pixel() and hpix_angles_to_nest_pixel(). It is useful if you plan to call many times one of the two functions, but you do not know in advance which one you'll use. Here's an example:

```
void
function(const hpix_resolution_t * resolution,
         hpix_ordering_t order,
         const double * thetas,
         const double * phis,
         size_t num_of_pixels)
{
    size_t idx;
    hpix_angles_to_pixel_fn_t * ang2pix_fn;
    if(order == HPIX_ORDER_RING)
        ang2pix_fn = hpix_angles_to_ring_pixel;
    else
        ang2pix_fn = hpix_angles_to_nest_pixel;

    for(idx = 0; idx < num_of_pixels; ++idx)
    {
        hpix_pixel_num_t pix_num;
        /* Since ang2pix_fn has already been assigned, we
         * avoid using a 'if' within the 'for' cycle.
         */
        pix_num = ang2pix_fn(resolution, thetas[idx], phis[idx]);

        /* Here you use 'pix_num' */
    }
}
```

4.2.2 Converting 3D vectors

The functions described in this paragraph convert 3D vectors into some other representation. The vector does not need to have length one.

void **hpix_vector_to_angles** (double *x*, double *y*, double *z*, double * *theta*, double * *phi*)

Convert the vector *x*, *y*, *z* into the pair of angles *theta*, *phi*. It is not necessary for the vector to have length one. The two angles will be properly normalized (i.e. *theta* will be within 0 and pi, and *phi* will be within 0 and 2pi).

See also hpix_angles_to_vector().

hpix_pixel_num_t **hpix_vector_to_ring_pixel** (const hpix_resolution_t * *resolution*, double *x*, double *y*, double *z*)

Convert the vector *x*, *y*, *z* into the *RING* index of the pixel for which the specified direction falls within.

See also hpix_ring_pixel_to_vector().

hpix_pixel_num_t **hpix_vector_to_nest_pixel** (const hpix_resolution_t * *resolution*, double *x*, double *y*, double *z*)

Convert the vector *x*, *y*, *z* into the *NESTED* index of the pixel for which the specified direction falls within.

See also hpix_nest_pixel_to_vector().

typedef hpix_pixel_num_t **hpix_vector_to_pixel_fn_t** (hpix_nside_t, double, double, double)

This defines a name for the prototype of the two functions hpix_vector_to_ring_pixel() and hpix_vector_to_nest_pixel(). It is useful if you plan to call many times one of the two functions, but you do not know in advance which one you'll use. See hpix_angles_to_pixel_fn_t for a nice example.

4.2.3 Converting pixel indexes

The functions described in this paragraph convert pixel indices, either in *RING* or *NESTED* scheme, into some other representation.

void **hpix_ring_pixel_to_angles** (const hpix_resolution_t * *resolution*, hpix_pixel_num_t *pixel*, double * *theta*, double * *phi*)

Convert the direction of the center of the pixel with *RING* index *pixel* into the two angles *theta* (colatitude) and *phi* (longitude).

See also hpix_angles_to_ring_pixel().

void **hpix_nest_pixel_to_angles** (const hpix_resolution_t * *resolution*, hpix_pixel_num_t *pixel*, double * *theta*, double * *phi*)

Convert the direction of the center of the pixel with *NESTED* index *pixel* into the two angles *theta* (colatitude) and *phi* (longitude).

See also hpix_angles_to_nest_pixel().

typedef void **hpix_pixel_to_angles** (const hpix_resolution_t *, hpix_pixel_num_t, double *, double *)

This defines a name for the prototype of the two functions hpix_ring_pixel_to_angles() and hpix_nest_pixel_to_angles(). It is useful if you plan to call many times one of the two functions, but you do not know in advance which one you'll use. See hpix_angles_to_pixel_fn_t for a nice example.

void **hpix_ring_pixel_to_vector** (const hpix_resolution_t * *resolution*, double * *x*, double * *y*, double * *z*)

Convert the direction of the center of the pixel with *RING* index *pixel* into the components of a vector *x*, *y*, *z*. It is guaranteed that *x*x + y*y + z*z == 1.0*.

See also hpix_vector_to_ring_pixel().

void **hpix_nest_pixel_to_vector** (const hpix_resolution_t * *resolution*, double * *x*, double * *y*, double * *z*)

Convert the direction of the center of the pixel with *RING* index *pixel* into the components of a vector *x*, *y*, *z*. It is guaranteed that *x*x + y*y + z*z == 1.0*.

See also hpix_vector_to_ring_pixel().

typedef void **hpix_pixel_to_vector** (const hpix_resolution_t *, hpix_pixel_num_t, double *, double *, double *)

This defines a name for the prototype of the two functions hpix_ring_pixel_to_vector() and hpix_nest_pixel_to_vector(). It is useful if you plan to call many times one of the two functions, but you do not know in advance which one you'll use. See hpix_angles_to_pixel_fn_t for a nice example.

4.3 Converting RING into NESTED and back

The following functions allow you to switch between the *RING* and *NESTED* schemes. Each scheme has its own advantages: *RING* is good when you want to decompose the map in spherical harmonics (because pixels on the same latitude are contiguous), *NESTED* is useful if you apply wavelet transforms or are looking for point sources (neighbour points are easy to find with this scheme).

hpix_pixel_num_t **hpix_nest_to_ring_idx**(const hpix_resolution_t * *resolution*, hpix_pixel_num_t *nest_index*)

Convert the index of pixel *nest_index* from *NESTED* to *RING*.

hpix_pixel_num_t **hpix_ring_to_nest_idx**(const hpix_resolution_t * *resolution*, hpix_pixel_num_t *ring_index*)

Convert the index of pixel *nest_index* from *NESTED* to *RING*.

void **hpix_switch_order**(hpix_map_t * *map*)

Switch the order of the map from *RING* to *NESTED* or vice versa, depending on the current ordering of the map (see `hpix_map_ordering()`). Note that the reordering is done in-place: this means that no additional memory is needed during the conversion, but if you want to access both maps you have to copy it somewhere else before calling this function.

CHAPTER 5

The hpix_map_t type

In this section we introduce the `hpix_map_t` type, which is used to hold information about a map, as well as a number of ancillary types and functions.

The following example (*examples/mapinfo.c*) is a program which shows information about a set of FITS temperature maps specified from the command line. It is a good example of the way HPixLib functions are meant to be used in a real program (albeit as simple as this is). In the rest of this section a detailed documentation of every function used in the example will be provided.

```
#include <hpixlib/hpix.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>

/* Compute the peak-to-peak difference of the value of the
   pixels in the map */
double peak_to_peak_amplitude(const hpix_map_t * map)
{
  size_t idx;
  double min, max;
  double * pixels;

  assert(map);

  pixels = hpix_map_pixels(map);
  min = max = pixels[0];
  for(int idx = 1; idx < hpix_map_num_of_pixels(map); ++idx)
  {
    if(isnan(pixels[idx])) /* Skip unseen pixels */
      continue;

    if(min > pixels[idx])
      min = pixels[idx];

    if(max < pixels[idx])
      max = pixels[idx];
  }

  return max - min;
}

int main(int argc, char ** argv)
{
```

```
  char * error_message = NULL;

  /* Skip the program name */
  ++argv; --argc;

  if(argc == 0) {
      fputs("Usage: mapinfo FILE1 [FILE2...]\n", stderr);
      return EXIT_SUCCESS;
  }

  while(argc--) {
      int cfitsio_status = 0;
      hpix_map_t * map;

      hpix_load_fits_component_from_file(argv[0], 1, &map, &cfitsio_status);

      if(map)
      {
        printf("File name: %s\n", *argv);
        printf("NSIDE: %u\n", hpix_map_nside(map));
        printf("Ordering: %s\n",
               hpix_map_ordering_scheme(map) == HPIX_ORDER_SCHEME_RING ?
               "RING" : "NEST");
        printf("Peak-to-peak variation: %.4g\n",
               peak_to_peak_amplitude(map));

        hpix_free_map(map);
      } else {
        fprintf(stderr, "Error: %s\n", error_message);
        hpix_free(error_message);
      }

      ++argv;
  }

  return EXIT_SUCCESS;
}
```

5.1 Basic types

hpix_ordering_scheme_t
: This `enum` type specifies the ordering scheme of the map. It can assume the values `HPIX_ORDER_SCHEME_RING` or `HPIX_ORDER_SCHEME_NEST`.

hpix_coordinates_t
: This `enum` type specifies the coordinate system used by the map. It can either be `HPIX_COORD_GALACTIC` (Galactic coordinates), `HPIX_COORD_ECLIPTIC` (ecliptic coordinates), `HPIX_COORD_CELESTIAL` or `HPIX_COORD_CUSTOM` (custom Euler rotation).

hpix_resolution_t
: This structure is conceptually equivalent to a *nside* value, but it keeps a number of mathematical quantities (all derived by *nside* itself) that are handy for manipulating Healpix maps at that resolution. (It basically caches these values in order to save time in computations.)

hpix_map_t
: This is the basic type used to hold information about a Healpix map. It is a structure that should consid-

ered to be opaque, i.e. accessing its members is forbidden. You should instead use access functions like `hpix_map_ordering()`, `hpix_map_nside()` and `hpix_map_pixels()`. See below for a complete list.

5.2 Map creation/distruction

Functions `hpix_create_map()` and `hpix_create_map_from_array()` create a map in memory. The first one is useful when you do now know in advance the value of the pixels you're going to put into the pixel. The second one is handy if you were able to retrieve pixel values from some medium and want to "wrap" them into a `hpix_map_t` structure in order to use them with HPixLib.

hpix_map_t * **hpix_create_map** (hpix_nside_t *nside*, hpix_ordering_scheme_t *ordering*)

Create a zero-filled Healpix map with a resolution of *nside* and a ordering scheme equal to *ordering* (see `hpix_ordering_scheme_t` for more information about the accepted values).

hpix_map_t * **hpix_create_map_from_array** (double * *array*, size_t *num_of_elements*, hpix_ordering_scheme_t *ordering*)

Create a Healpix map using the values in *array*. The value of *nside* is calculated from *num_of_pixels* using `hpix_npixel_to_nside()`. By default, the map is considered to be in Galactic coordinates.

void **hpix_free_map** (hpix_map_t * *map*)

Free any memory associated with *map*. Once the function exits, *map* is no longer available.

hpix_map_t * **hpix_create_copy_of_map** (const hpix_map_t * *map*)

Return a pointer to a copy of *map*. This is useful if you plan to modify *map* inplace (e.g. by means of a call to `hpix_scale_pixels_by_constant_inplace()`) but you want to keep a copy of the map as it was before the modification. Once no longer used, the new copy must be disposed using `hpix_free_map()` as usual.

5.3 Loading and saving maps

The following functions are used to load and save Healpix maps into FITS files. Such files are fully compatible with those produced by the standard Healpix library.

int **hpix_load_fits_component_from_fitsptr** (fitsptr * *fptr*, unsigned short *column_number*, hpix_map_t ** *map*, int * *status*)

Load one component (I, Q, or U) from the FITS file specified by *fptr*, which must have been properly initialized using one of CFITSIO's functions, e.g. `fits_open_table()` and `fits_movabs_hdu()`.

If any error occurs, the function returns zero. Otherwise, it makes *map* pointing to a new `hpix_map_t` object that must be freed using `hpix_free_map()` when it is no longer useful. Moreover, if *status* is not null, then it will be initialized with the appropriate CFITSIO error code.

Note that pixels marked as `UNSEEN` are converted to NaN. This is different from what the standard Healpix library does.

int **hpix_load_fits_component_from_file** (const char * *file_name*, unsigned short *column_number*, hpix_map_t ** *map*, int * *status*)

Wrapper to `hpix_load_fits_component_from_fitsptr()` which automatically opens the FITS file named *file_name* and moves to the first binary table HDU.

int **hpix_create_empty_fits_table_for_map** (fitsfile * *fptr*, const hpix_map_t * *template_map*, unsigned short *num_of_components*, const char * *measure_unit*, int * *status*)

Create a new HDU in an already-opened FITS file pointed by *fptr* and write a set of keywords that describe the

shape of a map like *template_map*. The parameter *num_of_components* tells how many *TDOUBLE* columns the HDU will have: it must be a number between 1 and 3. (No checking is done on this.)

The parameter *measure_unit* should be a string identifying the unit of measure of all the columns. You should use short names, e.g. *K* instead of *Kelvin*.

If the function is successful, it returns nonzero. If there is an error and *status* is not null, then it will be initialized with the appropriate CFITSIO code.

Note that write-access must be granted to *fptr*, otherwise the function will fail.

int **hpix_save_fits_component_to_fitsfile**(const char * *file_name*, const hpix_map_t * *map*, int *data_type*, int * *status*)

Save *map* into a FITS file named *file_name*. The value of *data_type* is one of the possible types accepted by CFITSIO (e.g. `TINT`, refer to the CFITSIO documentation for a full list).

As for `hpix_load_fits_component_from_file()`, if something went wrong then the function returns zero and initializes *error_status* with a newly-created string describing the error. (In this case you must free it using `hpix_free()`.) Note that *error_status* can be set to `NULL`: in this case, no information about the error type will be available.

If there are NaN values in the map pixels, they will be converted into the standard Healpix's `UNSEEN` value.

int **hpix_save_fits_component_to_file**(const char * *file_name*, const hpix_map_t * *map*, int *data_type*, int * *status*)

Wrapper to `hpix_save_fits_component_to_fitsptr()` which automatically create a FITS file named *file_name*.

int **hpix_load_fits_pol_from_file**(const char * *file_name*, hpix_map_t ** *map_i*, hpix_map_t ** *map_q*, hpix_map_t ** *map_u*, char ** *error_status*)

Load the three components of a IQU map from a FITS file named *file_name*. The three components are read from the first table extension of the FITS file. Note that it is an error to call this function on temperature-only maps.

The double pointers *map_i*, *map_q* and *map_u* must point to `hpix_map_t *` variables, which are automatically allocated by the function, and they must be freed using `hpix_free_map()`.

If any error occurs, the function returns `NULL`, otherwise it returns a new `hpix_map_t` object that must be freed using `hpix_free_map()` when it is no longer useful. Moreover, if *status* is not null, then it will be initialized with the appropriate CFITSIO error code.

Note that pixels marked as `UNSEEN` are converted to NaN. This is different from what the standard Healpix library does.

int **hpix_save_fits_pol_to_file**(const char * *file_name*, const hpix_map_t * *map_i*, const hpix_map_t * *map_q*, const hpix_map_t * *map_u*, int *data_type*, char ** *error_status*)

Save the three I, Q, U maps into a FITS file named *file_name*. The value of *data_type* is one of the possible types accepted by CFITSIO (e.g. `TINT`, refer to the CFITSIO documentation for a full list).

As for `hpix_load_fits_pol_from_file()`, if something went wrong and *status* is not null, then it will be initialized with the appropriate CFITSIO error code.

If there are NaN values in the map pixels, they will be converted into the standard Healpix's `UNSEEN` value.

int **hpix_is_iqu_fits_map**(const char * *file_name*)

This helper functions can be used to establish if the FITS file named *file_name* contains a temperature map (I Stokes component) or a temperature+polarization map (I, Q and U Stokes components).

This function can be useful to determine if you can call `hpix_load_fits_pol_map()` or not.

5.4 Accessing map information

The following functions provide a quick access to a `hpix_map_t` type. They run in constant time and are therefore pretty cheap to call.

hpix_ordering_scheme_t **hpix_map_ordering** (const hpix_map_t * *map*)
: Return the ordering of the map. See the definition of `hpix_ordering_scheme_t` for an explanation of the return value.

hpix_coordinates_t **hpix_map_coordinate_system** (const hpix_map_t * *map*)
: Return the coordinate system used by the map. See the definition of `hpix_coordinates_t` for an explanation of the return value.

hpix_nside_t **hpix_map_nside** (const hpix_map_t * *map*)
: Return the value of *nside* for *map*.

size_t **hpix_num_of_pixels** (const hpix_map_t * *map*)
: Return the number of pixels in *map*. This is always equal to `hpix_nside_to_npixel(hpix_map_nside(map))`.

const hpix_resolution_t * **hpix_map_resolution** (const hpix_map_t * *map*)
: Return a const pointer to a `hpix_resolution_t` structure.

CHAPTER 6

Mathematical operations on map pixels

This section describes the HPixLib functions that perform mathematical calculations on the pixels of a map. The number of calculations implemented here are surely not exhaustive, but they are the most used operations. Nevertheless, any non-trivial operation on the pixels can be implemented by directly accessing the array containing the pixel values through the function `hpix_map_pixels()`.

The following simple example uses `hpix_scale_pixels_by_constant_inplace()` to optionally converts a map containing pixel temperatures in K into a map where temperatures are expressed in muK.

```
hpix_map_t * map;

/* Read/initialize 'map' in some way... */
...

char choice;
printf("Do you want values to be expressed as microK (y/n)? ");
scanf("%c", &choice);

if(choice == 'y') {
  const double KELVIN_TO_MICROKELVIN = 1.0e6;
  hpix_scale_pixels_by_constant_inplace(map, KELVIN_TO_MICROKELVIN);
}

/* Do some calculations on 'map' */
...
```

6.1 In-place transformations

The following functions change the value of the pixels in a map. If you are interested in keeping the old values, you should copy the map before calling them.

hpix_map_t * **hpix_scale_pixels_by_constant_inplace** (hpix_map_t * *map*, double *constant*)
: Multiply the value of every unmasked pixel in *map* by the floating-point number *constant*. Masked pixels are left as they are.

hpix_map_t * **hpix_add_constant_to_pixels_inplace** (hpix_map_t * *map*, double *constant*)
: Add *constant* to the value of every pixel in *map*.

void **hpix_remove_monopole_from_map_inplace** (hpix_map_t * *map*)
: Subtract the average value of the unmasked pixels from the map.

6.2 Statistical estimators

double **hpix_average_pixel_value**(const hpix_map_t * *map*)
Return the average value of the unmasked pixels in the map.

CHAPTER 7

Drawing maps

In this section we describe the most complex part of the library, that is the code that produces a graphical representation of a map. To better understand the difficulties of such task, let us consider how this is accomplished by the standard HEALPix library and by Healpy, a Python wrapper to HEALPix. The "standard" Healpix implementation is able to plot maps in a number of ways:

- The IDL library contains roughly 7,600 lines of code which implement `MOLLVIEW()` and similar functions. Such functions are written in pure IDL and use the IDL plotting functions.
- Two standalone programs, map2gif and map2tga, convert a map saved in a FITS file into a GIF/TGA image. The first one is written in Fortran90 and is roughly 1,200 lines of code, plus the source code of the gifdraw library (roughly 12,000 lines of code, in the directory `src/f90/lib` of the tarball). The same figures apply to map2tga as well.

Because of this situation, the creators of the Healpy Python library decided to implement a set of plotting routines from scratch. More than 2,000 lines of code are needed to implement functions like `mollview()` and `mollzoom()`; they are based on the well-known matplotlib library.

Our approach is to implement a very generic interface for map plotting in HPixLib (i.e. one that is agnostic to the tool actually used to draw the map: Quartz, Gtk+, Cairo ...). Depending on the graphics library, there are two possible approaches for drawing a map:

- Generate a bitmap. (This is the approach followed by the Healpix library and by Healpy.) The output of the process is a $N \times M$ matrix of pixels whose elements are calculated using a ray-tracing algorithm. The image has a fixed resolution, which implies that it shows poor results when enlarged. The ray-tracing algorithm has the advantage of being quite fast, and bitmaps can be displayed and saved quickly. When saved, the size of the file scales with the number of elements in the matrix, but it is independent of the number of pixels in the map.
- Generate a vector image. This solution has the drawback of producing very large files when *nside* is large, but vector maps scale very well when enlarged. The typical formats used to store such maps are Postscript and PDF.

HPixLib will provide two sets of functions to ease the production of bitmapped and vector maps. (Such functions need to be wrapped with some glue code which actually writes the map on disk or display it on the screen.) Currently HPixLib supports the creation of bitmapped images; the veneration of vector images is considered less important and will be implemented in future releases of the library.

The library provides a program, map2fig, which is similar to the two programs provided in the Healpix distribution, map2gif and map2tga. However, being based on the Cairo graphics library, it allows to save maps in vector formats as well. (The map itself is a bitmapped image embedded in the EPS/PDF/SVG file, but the title, the color bar and every other element is a vector.) This allows e.g. to modify these maps within vector drawing programs like Inkscape or Adobe Illustrator.

7.1 Introduction: a poor-man clone of map2fig

We begin with a full example of how to use the drawing/palette functions provided by HPixLib. The following program reads a map from a file and then outputs to stdout a bitmap in PPM format (http://netpbm.sourceforge.net/doc/ppm.html):

```
#include <stdio.h>
#include <hpixlib/hpix.h>

void output_map_to_file(const hpix_map_t * map,
                        hpix_color_palette_t * palette,
                        FILE * out)
{
    /* Keeping it double makes calculations more efficient */
    const double max_color_level = 255.0;

    hpix_bmp_projection_t * proj =
      hpix_create_bmp_projection(640, 320);
    hpix_set_mollweide_projection(proj);

    double min, max;
    double * bitmap = hpix_bmp_projection_trace(proj, map, &min, &max);

    /* Write the PPM header */
    fprintf(out, "P3\n%u %u\n%u\n",
          hpix_bmp_projection_width(proj),
          hpix_bmp_projection_height(proj),
          (unsigned) max_color_level);

    double *restrict cur_pixel = bitmap;
    for(unsigned y = 0; y < hpix_bmp_projection_height(proj); ++y)
    {
      for(unsigned x = 0; x < hpix_bmp_projection_width(proj); ++x)
      {
          hpix_color_t pixel_color;
          hpix_palette_color(palette,
                             (*cur_pixel++ - min) / (max - min),
                             &pixel_color);
          fprintf(out, "%3u %3u %3u\t",
                  (unsigned) (hpix_red_from_color(&pixel_color) * max_color_level),
                  (unsigned) (hpix_green_from_color(&pixel_color) * max_color_level),
                  (unsigned) (hpix_blue_from_color(&pixel_color) * max_color_level));
      }

      fputc('\n', out);
    }

    hpix_free(bitmap);
    hpix_free_bmp_projection(proj);
}

int main(int argc, const char ** argv)
{
    if(argc != 2)
    {
      fputs("You must specify the name of a FITS file on the command line\n",
            stderr);
      return 1;
```

```
    }

    hpix_map_t * map;
    const char * file_name = argv[1];
    int status;
    if(! hpix_load_fits_component_from_file(file_name, 1, &map, &status))
    {
      fprintf(stderr, "Unable to read map %s\n", file_name);
      return 1;
    }

    hpix_color_palette_t * palette = hpix_create_planck_color_palette();
    output_map_to_file(map, palette, stdout);
    hpix_free_color_palette(palette);
    hpix_free_map(map);
}
```

The typical usage is to produce a bitmap, then use *min_value* and *max_value* to scale it from the map measure unit into a color space. (You can find the source code of this program in the file `examples/map2ppm.c`).

7.2 Available projections

HPixLib implements a number of cartographic projections. You can either access the low-level projection functions or rely on the library to directly produce a map.

The low-level projection functions allow to perform one of the following tasks:

1. Convert the coordinate of a 2-D plane into a direction towards the sky and vice-versa. (Example: `hpix_mollweide_xy_to_angles()`.)
2. Check if a point on a 2-D plane is visible or not within the bitmap's rectangle. (Example: `hpix_mollweide_is_xy_inside()`.)

The latter point is important for those projections like the Mollweide's, which is enclosed in a shape which is not a rectangle like the bitmap's (ellipse).

The cartographic projections implemented by HPixLib are listed in the enumerated type `hpix_projection_type_t`.

_Bool **hpix_mollweide_xy_to_angles** (const hpix_bmp_projection_t * *proj*, unsigned int *x*, unsigned int *y*, double * *theta*, double * *phi*)

This function calculates the direction towards the sky that corresponds to the (*x*, *y*) point in the 2-D bitmap projection pointed by *proj*. If there is no point which corresponds to (*x*, *y*), i.e., if these coordinates are outside Mollweide's ellipse, then the function returns *FALSE*, otherwise *TRUE*.

_Bool **hpix_mollweide_is_xy_inside** (const hpix_bmp_projection_t * *proj*, unsigned int *x*, unsigned int *y*)

This function checks that the point (*x*, *y*) in the bitmap lies within Mollweide's ellipse. If it does, return *TRUE*. (The return value has therefore the same meaning as `hpix_mollweide_xy_to_angles()`.)

_Bool **hpix_equirectangular_xy_to_angles** (const hpix_bmp_projection_t * *proj*, unsigned int *x*, unsigned int *y*, double * *theta*, double * *phi*)

This function calculates the direction towards the sky that corresponds to the (*x*, *y*) point in the 2-D bitmap projection pointed by *proj*. If there is no point which corresponds to (*x*, *y*), then the function returns *FALSE*, otherwise *TRUE*. This happens only if *x* or *y* fall outside the rectangle enclosing the bitmap.

_Bool **hpix_equirectangular_is_xy_inside** (const hpix_bmp_projection_t * *proj*, unsigned int *x*, unsigned int *y*)

This function checks that the point (*x*, *y*) in the bitmap lies within the bitmap's region. If it does, return *TRUE*.

(The return value has therefore the same meaning as `hpix_equirectangular_xy_to_angles()`.)

7.3 Bitmapped graphics

The interface provided by CHealpix for the generation of bitmapped graphics clearly shows the ray-tracing algorithm on whom it is grounded. In the following discussion we try to prevent the ambiguity between a "pixel" in a Healpix map and a "pixel" in a bitmap by referring to the second as "an element in the $N \times M$ matrix," or "matrix element" for short. All the functions implemented in this section have their name beginning with `hpix_bmp_`.

`hpix_bmp_projection_t`
This type contains all the information needed to transform a Healpix map into a bi-dimensional bitmapped projection. It is an opaque structure, which means that you are not allowed to directly access/modify its members: you need to rely on functions defined in this section, like e.g. `hpix_projection_width()`.

Projection type	Enumeration constant
No projection	`HPIX_PROJ_NULL`
Mollweide (equal area)	`HPIX_PROJ_MOLLWEIDE`
Equirectangular	`HPIX_PROJ_EQUIRECTANGULAR`

You can retrieve the cartographic projection used by a `hpix_bmp_projection_t` variable using the function `hpix_bmp_projection_type()`.

hpix_bmp_projection_t * **`hpix_create_bmp_projection`** (unsigned int *width*, unsigned int *height*)
Create a new `hpix_bmp_projection_t` object and initialize its width and height. This object must be deallocated using `hpix_free_bmp_projection()`.

Once this function is called, there is no cartographic projection associated with it. You must call one of the `hpix_set_*_projection` functions listed below (e.g., `hpix_set_mollweide_projection()`) in order to effectively trace bitmaps.

void **`hpix_free_bmp_projection`** (hpix_bmp_projection_t * *proj*)
Free all the memory associated with *proj*, which therefore can no longer be used.

hpix_projection_type_t **`hpix_bmp_projection_type`** (const hpix_bmp_projection_t * *proj*)
Return the code identifying the cartographic projection associated with *proj*.

void **`hpix_set_equirectangular_projection`** (hpix_bmp_projection_t * *proj*)
Configure *proj* to use an equirectangular cartographic projection. This kind of projection is very useful if you plan to wrap the bitmap around a 3D sphere, e.g., using a ray-tracing program.

void **`hpix_set_mollweide_projection`** (hpix_bmp_projection_t * *proj*)
Configure *proj* to use a Mollweide cartographic projection. Most of the full-sky CMB maps are usually produced using this kind of projection.

_Bool **`hpix_bmp_projection_is_xy_inside`** (const hpix_bmp_projection_t * *proj*, unsigned int *x*, unsigned int *y*)
Determine if the bitmap coordinates (*x*, *y*) fall within the map or not. The function internally calls the appropriate function for the cartographic projection associated with *proj*. For instance, if you called `hpix_set_mollweide_projection()`, it acts as a wrapper to `hpix_mollweide_is_xy_inside()`.

_Bool **`hpix_bmp_projection_xy_to_angles`** (const hpix_bmp_projection_t * *proj*, unsigned int *x*, unsigned int *y*, double * *theta*, double * *phi*)
Convert the bitmap coordinates (*x*, *y*) into a par (*theta*, *phi*), that is, colatitude and longitude. The function internally calls the appropriate function for the cartographic projection associated with *proj*. For instance, if you called `hpix_set_mollweide_projection()`, it acts as a wrapper to `hpix_mollweide_xy_to_angles()`.

7.4 Projection properties

As said above, `hpix_bmp_projection_t` is an opaque structure and as such you cannot read/modify its members directly: you have to use the facilities provided by the library.his.

unsigned int **`hpix_projection_width`** (const hpix_bmp_projection_t * *proj*)
Return the width of the bitmap, i.e. the number of columns.

unsigned int **`hpix_projection_height`** (const hpix_bmp_projection_t * *proj*)
Return the height of the bitmap, i.e. the number of rows.

7.5 Painting functions

double * **`hpix_bmp_projection_trace`** (const hpix_bmp_projection_t * *proj*, const hpix_map_t * *map*, double * *min_value*, double * *max_value*)
This function creates a bitmap (rectangular array of numbers) containing a Mollweide projection of *map*. The details of the projection are specified by the *proj* parameter (size of the bitmap, set of coordinates to be used and so on). The bitmap is an array of floating-point values, each using the same scale as in the original map (i.e. if the map represents a set of temperatures in Kelvin, then each pixel in the bitmap will be measured in Kelvin as well).

Note that the Mollweide projection must have an aspect ratio 2:1, i.e., the width of the image should be twice its height. HPixLib does not enforce such requirement on the width and height of the bitmap, as the true aspect ratio of the image depends on the pixel aspect ratio of the device where the bitmap will be displayed as well. However, a good rule of thumb is to pick a width which is roughly twice the height, as most of the display devices in use today have a pixel aspect ratio which is close to 1:1.

When the bitmap returned by this function is no longer useful, you must free it using `hpix_free()`.

7.6 Color palettes

Functions like `hpix_bmp_projection_trace()` create a bitmapped representation of a map in which each matrix element of the bitmap is expressed in the same units as the map (e.g., if a map represents some measured sky flux in Jy, then the matrix elements of the bitmap will be expressed in Jy too).

In order to properly display the bitmap on a device, HPixLib provides a number of functions which convert floating-point values (in arbitrary scales) into colors. Moreover, HPixLib is able to handle color sets, called color palette, that represent gradients used to attribute specific colors to each pixel in a map.

7.6.1 Color types and functions

The representation of colors used by HPixLib (through the type `hpix_color_t`) uses the classical RGB decomposition, i.e., each color is expressed as a mixture of red, green, and blue levels (RGB), where each level is a floating-point value between 0.0 (absence) to 1.0 (saturation).

`hpix_color_t`
This type is a structure made up by three fields: *red*, *green*, and *blue*. Each element is a floating-point value normalized to unity.

The value of the *red*, *green*, and *blue* fields should be between 0.0 (lack of shade) and 1.0 (saturated shade). HPixLib however does not enforce such limits, since it is quite common in computer graphics to represent saturated values using levels greater than 1.0. (E.g., this effect is used in ray-tracing programs like POV-Ray to create very bright light sources.)

The structure is not opaque, therefore it can be created on-the-fly using the facilities of the C99 language:

```
/* This will work in C99, but not in C89 */
hpix_color_t red_color =
    (hpix_color_t) { .red = 1.0, .green = 0.0, .blue = 0.0 };
```

Although the members of hpix_color_t can be accessed directly, HPixLib provides getter/setter functions in order to ease the creation of binding libraries in other languages.

hpix_color_t **hpix_create_color**(double *red*, double *green*, double *blue*)
Return a hpix_color_t structure initialized to the specified values of red, green, and blue.

hpix_red_from_color(const hpix_color_t * *color*)
Return the red level of the color.

hpix_green_from_color(const hpix_color_t * *color*)
Return the green level of the color.

hpix_blue_from_color(const hpix_color_t * *color*)
Return the blue level of the color.

7.6.2 Color palettes

A color palette is a set of colors and rules which specify how to combine the colors in order to provide a continuous and smooth palette. The idea is that every floating-point number falling within some predefined range can then be converted into a RGB color and displayed on a device.

The original Healpix color palette shows an example. The palette is made by six colors, each with an associated floating-point number between 0 and 1. The library is able to blend the colors (using linear interpolation) to produce a smooth transition between them. The programmer can create custom color palettes using the functions described in this section.

Figure 7.1: The original Healpix color palette

hpix_color_palette_t
The type hpix_color_palette_t is an opaque type that holds the information which represents a color palette:

1. An array of levels and colors (hpix_color_t). This array always contains at least two elements: the one at level 0 (left side) and the one at level 1 (right side).
2. The color to be used for unseen pixels.

HPixLib provides a few functions that create nice-looking palettes ready to use: hpix_create_grayscale_color_palette() and hpix_create_healpix_color_palette(). When a palette is no longer used, the program must call hpix_free_color_palette().

Note that, being an opaque type, hpix_color_palette_t can be accessed only using the setter/getter functions described here.

hpix_color_palette_t * **hpix_create_black_color_palette**(void)
Create a black color palette. This is never used in real-world examples, but it can be a good starting point for creating custom palettes using hpix_set_color_for_step_in_palette() and hpix_add_step_to_color_palette().

When the palette is no longer used, the program must call hpix_free_color_palette().

hpix_color_palette_t * **hpix_create_grayscale_color_palette** (void)
Create a color palette made by gray shades. (The color used for unseen pixels has a reddish tint, in order to make it distinguishable from the others.)

When the palette is no longer used, the program must call hpix_free_color_palette().

hpix_color_palette_t * **hpix_create_healpix_color_palette** (void)
Create a color palette that mimics the one used by the original Healpix library. When the palette is no longer used, the program must call hpix_free_color_palette().

hpix_color_palette_t * **hpix_create_planck_color_palette** (void)
Create a color palette that mimics the one used in the first Planck data release. When the palette is no longer used, the program must call hpix_free_color_palette().

void **hpix_free_color_palette** (hpix_color_palette_t * *palette*)
Release any memory associated with the palette.

The color to be used for unseen/masked/bad pixels can be read using hpix_color_for_unseen_pixels_in_palette() and set using hpix_set_color_for_unseen_pixels_in_pale

void **hpix_set_color_for_unseen_pixels_in_palette** (hpix_color_palette_t * *palette*, hpix_color_t *new_color*)
Set the color to be used for unseen pixels in the specified palette.

hpix_color_t **hpix_set_color_for_unseen_pixels_in_palette** (hpix_color_palette_t * *palette*)
Retrieve from the palette the color to be used for unseen pixels.

It is possible to add color levels and to modify the existing ones. Note however that it is not possible to delete levels from a hpix_color_palette_t variable.

void **hpix_add_step_to_color_palette** (hpix_color_palette_t * *palette*, double *level*, hpix_color_t *color*)
Add a new color and a new level to the color palette. The new color will be appended to the list of existing color steps. Before using the palette, you must ensure that hpix_sort_levels_in_color_palette() has been called, so that all the levels are in ascending order.

Note that the code does not check if the level you are specifying in the call is already present in the palette. If this is the case, the library might behave unexpectedly (including divisions by zero).

size_t **hpix_num_of_steps_in_color_palette** (const hpix_color_palette_t * *palette*)
Return the number of color steps in the palette. This number is useful if you want to cycle over the steps using e.g. calls to hpix_color_for_step_in_palette() and hpix_level_for_step_in_palette().

hpix_color_t **hpix_color_for_step_in_palette** (const hpix_color_palette_t * *palette*, size_t *zero_based_index*)
Return the color associated with the given step in the palette. The value of *zero_based_index* ranges from 0 to the value returned by hpix_num_of_steps_in_color_palette().

double **hpix_level_for_step_in_palette** (const hpix_color_palette_t * *palette*, size_t *zero_based_index*)
Return the level associated with the given step in the palette. This level should always be between 0.0 and 1.0. The value of *zero_based_index* ranges from 0 to the value returned by hpix_num_of_steps_in_color_palette().

void **hpix_set_color_for_step_in_palette** (hpix_color_palette_t * *palette*, size_t *zero_based_index*, hpix_color_t *new_color*)
Change the color associated with the given step in the palette.

See also hpix_color_for_step_in_palette().

void **hpix_set_level_for_step_in_palette**(hpix_color_palette_t * *palette*, size_t *zero_based_index*, double *new_level*)

Change the level associated with the given step in the palette. Use with care! You must ensure that the new level does not coincide with other levels in the palette, and that the first and last level in the array of steps are still 0.0 and 1.0.

See also `hpix_level_for_step_in_palette()`.

Here is an example of how to use these functions to dump the definition of a palette to *stdout*:

```
size_t num_of_steps = hpix_num_of_steps_in_color_palette(palette);

for(size_t index = 0; index < num_of_steps; ++index)
{
    double level = hpix_level_for_step_in_palette(palette, index);
    hpix_color_t color = hpix_color_for_step_in_palette(palette, index);

    printf("Level: %.2f   -- R: %.2f, G: %.2f, B: %.2f\n",
           color.red, color.green, color.blue);
}
```

If *palette* were the result of a call to `hpix_create_healpix_color_palette()`, the output of the code above would have been the following:

```
Level: 0.00   -- R: 0.00, G: 0.00, B: 0.50
Level: 0.15   -- R: 0.00, G: 0.00, B: 1.00
Level: 0.40   -- R: 0.00, G: 1.00, B: 1.00
Level: 0.70   -- R: 1.00, G: 1.00, B: 0.00
Level: 0.90   -- R: 1.00, G: 0.33, B: 0.00
Level: 1.00   -- R: 0.50, G: 0.00, B: 0.00
```

void **hpix_sort_levels_in_color_palette**(hpix_color_palette_t * *palette*)

Sort all the steps in the palette in increasing order with respect to their level. (The sort is done inplace using the Standard C library function *qsort*: depending on the implementation, it might require or not additional memory.)

Sorting the steps in the palette is crucial for allowing the algorithm implemented in `hpix_palette_color()` to work. For efficiency reasons, the function is *never* called automatically by HPixLib.

hpix_palette_color(const hpix_color_palette_t * *palette*, double *level*, hpix_color_t * *color*)

Set the fields of *color* so that it represents the specified *level* in the given color palette. The function uses a linear interpolation of the color steps in the palette.

The palette must be properly sorted using `hpix_sort_levels_in_color_palette()`. This condition is already satisfied for the palettes returned by `hpix_create_black_color_palette()`, `hpix_create_grayscale_color_palette()`, and `hpix_create_healpix_color_palette()`.

Before using a palette in a call to `hpix_get_color_palette()` or any function that implicitly calls it (e.g., `hpix_bmp_projection_to_cairo_surface()`), you must ensure these rules apply:

1. The first color step in the palette has level 0.
2. The last color step in the palette has level 1.
3. All the color steps are sorted in increasing order according to their level.
4. There must not be two color steps with the same value for the level.

HPixLib does not enforce any of these rules. To ensure that you comply with them, here is a set of rules:

- After you call `hpix_add_step_to_color_palette()`, call `hpix_sort_levels_in_color_palette()` to sort the list. If you make multiple calls to `hpix_add_step_to_color_palette()`, you can sort the list after the last call (which is very efficient).

- Never use `hpix_set_level_for_step_in_palette()` to change the level of the color steps with level 0.0 and 1.0.
- When adding new color steps with `hpix_add_step_to_color_palette()`, ensure that the level you are specifing was never used in the palette.
- If you want to change the color at one of the edges of the palette, the right way to do is to call `hpix_set_color_for_step_in_palette()`, as shown in the following example:

```
/* This might be unnecessary, but it does not harm. */
hpix_sort_levels_in_color_palette(palette);

size_t num_of_steps = hpix_num_of_steps_in_color_palette(palette);

/* Change the color for level 0 */
hpix_set_color_for_step_in_palette(0, hpix_create_color(0.0, 1.0, 1.0));
/* Change the color for level 1 */
hpix_set_color_for_step_in_palette(num_of_steps - 1, hpix_create_color(1.0, 1.0, 1.0));
```

7.7 Vector graphics

Not implemented yet.

CHAPTER 8

Command-line utilities

In the utilities directory there are a few non-trivial programs using HPixLib. They are automatically build and installed with the library, and provide a few useful utilities that can be used e.g. in bash scripts.

8.1 map2fig

This utility is compiled only if an useable version of the Cairo library (http://www.cairographics.org) is found at compilation time. It converts a map in FITS format into a figure. It can support the following formats:

- PNG (bitmapped format)
- PostScript and Encapsulated PostScript (EPS, a vector format)
- PDF (Adobe Portable Document Format, a vector format)
- SVG (Scalar Vector Graphics, a vector format)

(Some formats might not be available, depending on settings used in compiling the Cairo library.) Vector formats contain a bitmapped representation of the map, but the text and color bar are vector elements that can be modified by vector drawing programs like e.g. Inkscape (http://inkscape.org).

Run *map2fig –help* for a list of options.

CHAPTER 9

Indices and tables

- *genindex*
- *modindex*
- *search*