HPIC
(HPIC Pixelization In C)

Theodore Kisner

November 11, 2005

Contents

Chapter 1

Introduction

This software package is an implementation of the healpix algorithms originally developed by Gorski, Wandelt, Hivon, Banday and others[1, 2, 3, 4, 5]. I would like to thank and acknowledge the original HEALPIX authors for their hard work developing the theoretical framework upon which this project is based. Throughout this document, when I mention "healpix", I am refering to the algorithms themselves. When I refer to a specific implementation of these algorithms, I will write them as HPIC or HEALPIX .

In mid-September of 2005, version 2.00 of HEALPIX was released. Unlike previous releases, this one was licensed under the GPL. For the first time, HPIC and HEALPIX were legally allowed to share code. Hopefully this development will lead to a fruitful and peaceful coexistence between the two projects.

1.1 Overview

The HPIC package contains a number of different components: an object oriented C library, a Perl interface to the C library, and command line utilities for manipulating and operating on FITS files. So why create another software library when one already exists? Let me first say that I believe choice is a good thing! Secondly, I wanted to have an object-oriented set of tools which I could use in C, C++, and Perl programs. An original consideration was also that the HEALPIX license was incompatible with the GPL. This has changed with the release of version 2.00 of HEALPIX .

I have attempted to make the HPIC library as intuitive as possible. Although function names are long and descriptive, they are constructed in a uniform way according to what they do and what type of data they operate on. I have tried to loosely model the calling format after that of the GNU Science Library (GSL).

1.2 Current Status

As of version 0.50, the HPIC tools are already quite useful. Although there is still much functionality to implement, the features that are implemented have been tested as much as possible. I already make extensive use of these tools in other projects. A partial list of working features include:

- Basic single pixel operations (order conversion, projection, degrade/prograde, etc)
- Basic map operations(order conversion, arithmetic, degrade/prograde, comparison, etc)
- A set of useful vector types that can be easily resized
- Cartesian and sinusoidal projection of maps, pixel vectors, point vectors, and vector fields onto a 2D grid
- FITS reading/writing of full-sphere, cut-sphere and vector formats
- Perl interface to the C library
- Command line utilities for simple math and conversion operations on FITS files
- Compatibility functions for legacy programs that currently use the CHEALPIX tools.

Despite all these useful features, keep in mind the version number. I *think* that the existing functional interfaces are fairly stable, but alas I cannot make too many promises at this stage. The latest version of the HPIC tools can be found on the project website (http://cmb.phys.cwru.edu/hpic).

1.3 Future Roadmap

There are a few known problems with the current HPIC library that will be addressed in future releases. Also, the most useful tools (spherical harmonic transforms, filtering, etc) have yet to be implemented. Here is a list of things to do:

- Implement in-place order conversion of maps
- Finish implementing tree-based map structure
- Normal and spin-weighted harmonic transforms
- Standard types of map filtering

1.4 Converting Programs that Currently use CHEALPix

If your program currently uses the C library included with the HEALPIX software suite, it is trivial to convert it to use HPIC . You can either modify your function calls to pass the types that HPIC expects, or you can simply prepend a "compat_" prefix to each function call and recompile. Of course there are many ways of automating this- for example the one line perl command

```
> perl -i -p -e 's/ang2pix_ring/compat_ang2pix_ring/g' *
```

will replace all calls to `ang2pix_ring` with the "compatibility wrapper" version in all files in the current directory. The only other changes needed are to include the hpic.h header file instead of chealpix.h and link with -lhpic instead of -lchealpix.

Chapter 2

C Reference

The following chapter outlines how to use the HPIC library in your own software projects. You should be able to link this library to both C and C++ programs. The first step is to include the HPIC header file:

```
#include <hpic.h>
```

When linking the program, you will also need to link to the math and CFITSIO libraries (i.e. -lm -lcfitsio -lhpic).

For examples of how to use the following functions, see the "hpictest" program in the src/test directory.

2.1 Constants

There are many constants that are defined and used throughout the HPIC library:

Constant Name	Value	Meaning
HPIC_PI	3.14159265358979	The value of π
HPIC_INVPI	0.318309886183791	The value of $1/\pi$
HPIC_PISQ	9.86960440108936	The value of π^2
HPIC_HALFPI	1.5707963267949	The value of $\pi/2$
HPIC_NSIDE_MAX	8192	The maximum NSIDE value
HPIC_STRNL	200	The maximum string length
HPIC_DEBUG	0 or 1	Whether to print debug messages
HPIC_RING	0	Indicates RING ordering of a map
HPIC_NEST	1	Indicates NESTED ordering of a map
HPIC_COORD_C	0	Map in celestial/equatorial coordinates
HPIC_COORD_G	1	Map in galactic coordinates
HPIC_COORD_E	2	Map in ecliptic coordinates
HPIC_COORD_O	3	Map in other coordinates
HPIC_STND	0	Map always kept in a standard C array
HPIC_TREE	1	Map always kept in a pixel tree
HPIC_AUTO	2	Map switched between standard/tree
HPIC_VECBUF	10	Realloc buffer for vector resizing
HPIC_PROJ_CAR	0	Cartesian projection
HPIC_PROJ_SIN	1	Sinusoidal projection
HPIC_INTERSECT	0	Take the intersection of maps
HPIC_UNION	1	Take the union of maps
HPIC_FITS_FULL	0	FITS map file uses full-sphere maps
HPIC_FITS_CUT	1	FITS map file uses cut-sphere maps
HPIC_FITS_BIN	0	FITS file uses a binary extension
HPIC_FITS_ASCII	1	FITS file uses an ascii extension
HPIC_FITS_VEC	0	FITS file uses floating-point vectors
HPIC_FITS_VEC_INDX	1	FITS file uses an index vector
HPIC_NULL	-1.6375e30	Floating-point NULL value in a map
HPIC_EPSILON	0.0001e30	Range around HPIC_NULL equal to NULL
HPIC_INT_NULL	-2147483646	Integer NULL value in a map

2.2 Error Handling

The HPIC library contains a global pointer to an error handling function. Most functions in the HPIC library will call this error handler if they encounter a problem and then return an error code. The following error codes are defined:

Error Code	Value	Meaning
HPIC_ERR_NONE	0	No error
HPIC_ERR_ALLOC	1	Memory allocation error
HPIC_ERR_FREE	2	Memory freeing error
HPIC_ERR_NSIDE	3	Illegal NSIDE value
HPIC_ERR_ORDER	4	Illegal ORDER value
HPIC_ERR_COORD	5	Illegal COORDINATE value
HPIC_ERR_RANGE	6	Value is out of range
HPIC_ERR_ACCESS	7	Memory is not accessible
HPIC_ERR_PROJ	8	Projection error
HPIC_ERR_FITS	9	FITS error

If the global error handling pointer is NULL (which is the initial value), then the default error handler is called. This default handler prints an error message to

stderr and then terminates the program with abort(). If you would like to use the error handling in your programs, you can call the `hpic_error` function directly, or use one of the provided Macros.

```
void hpic_error(int errcode, const char *file, int line,
                const char *msg);

HPIC_ERROR(errcode, msg)
HPIC_ERROR_VAL(errcode, msg, value)
HPIC_ERROR_VOID(errcode, msg)
```

The Macros call `hpic_error` with the given error code and automatically pass in the values of the current line and file (`__LINE__` and `__FILE__`). Then the Macros exit the function, returning either the error code, a specified value, or nothing, depending on which Macro is used. Since it is not generally good to have library functions aborting programs directly, the user can define his or her own error handling function. You may wish (for example) to call `hpic_error_default` to print error message, but then **NOT** abort the program until it has done some further clean up. Here is an example of how you could do this:

```
/* custom error handler example */

void my_handler (int errcode, const char *file,
                 int line, const char *msg)
{
  /* handle errors */
  return;
}

/* keep pointer to old error handler */
hpic_error_handler_t *oldhandler;
oldhandler = hpic_set_error_handler(&my_handler);
```

If you wish to switch back to the default error handler, simply pass a NULL pointer to `hpic_set_error_handler`.

2.3 Low Level Functions

There are many low level functions which can be used independently and upon which higher level functions are based. These functions may be useful if you are writing custom software that only needs to deal with a few pixel operations, etc. If you are operating on many pixels, then (hopefully) you will find the higher level routines easier to use.

2.3.1 General Tools

Here are a few very simple functions. They are so simple that I won't waste much time on them. The following functions return 1 if their argument is "null" and 0 if it is "non-null". For the double and float versions, the argument is considered null if it lies within +/- HPIC_EPSILON of HPIC_NULL. For the integer version, the argument is considered null if it is equal to HPIC_INT_NULL.

```
int hpic_is_dnull(double val);
int hpic_is_fnull(float val);
int hpic_is_inull(int val);
```

It is sometimes useful to dynamically allocate an array of strings. These functions allocate and free an array of strings that each have length HPIC_STRNL.

```
char** hpic_strarr_alloc(size_t nstring);
int hpic_strarr_free(char **array, size_t nstring);
```

2.3.2 Pixel Tools

These functions handle basic pixel manipulation. Roughly speaking this includes: converting between RING and NESTED pixel numbers, converting between angular coordinates and pixel number, converting between rectangular coordinates and pixel number, and degrading a pixel number to a lower NSIDE value.

The first function below returns one if the value of `nside` is invalid. The other two functions simply convert between an NSIDE value and the total number of pixels in the map ($NPIX = 12 \cdot NSIDE^2$).

```
int hpic_nsidecheck(size_t nside);
size_t hpic_nside2npix(size_t nside);
size_t hpic_npix2nside(size_t npix);
```

These functions convert between RING and NESTED pixel numbers. For the return value, you should obviously pass a pointer to an existing data structure.

```
int hpic_nest2ring(size_t nside, size_t pnest,
                   size_t *pring);
int hpic_ring2nest(size_t nside, size_t pring,
                   size_t *pnest);
```

To convert between angular coordinates on the sphere and pixel number, use the following functions. The values of `theta` and `phi` are in radians, with theta measured from the North pole and phi measured in a right-handed sense from the prime meridian. In the functions `ang2pix_ring` and `ang2pix_nest`, the angles are first converted to a pixel number with NSIDE set to `HPIC_NSIDE_MAX`. This pixel

number is then degraded to the desired NSIDE. This ensures that rounding errors are consistently treated for all NSIDE values.

```
int hpic_pix2ang_ring(size_t nside, size_t pix,
                      double *theta, double *phi);
int hpic_pix2ang_nest(size_t nside, size_t pix,
                      double *theta, double *phi);
int hpic_ang2pix_ring(size_t nside, double theta,
                      double phi, size_t *pix);
int hpic_ang2pix_nest(size_t nside, double theta,
                      double phi, size_t *pix);
```

To convert between spherical coordinates on the sphere and their 3D cartesian equivalents you can use the following functions. The `xcomp`, `ycomp`, and `zcomp` variables are the x, y and z components of the point on the sphere.

```
int hpic_vec2ang(double xcomp, double ycomp, double zcomp,
             double* theta, double* phi);
int hpic_ang2vec(double theta, double phi, double* xcomp,
             double* ycomp, double* zcomp);
```

The following functions convert between pixel number and 3D cartesian components. When converting from pixel number to rectangular coordinates, the coordinates of the pixel center are returned.

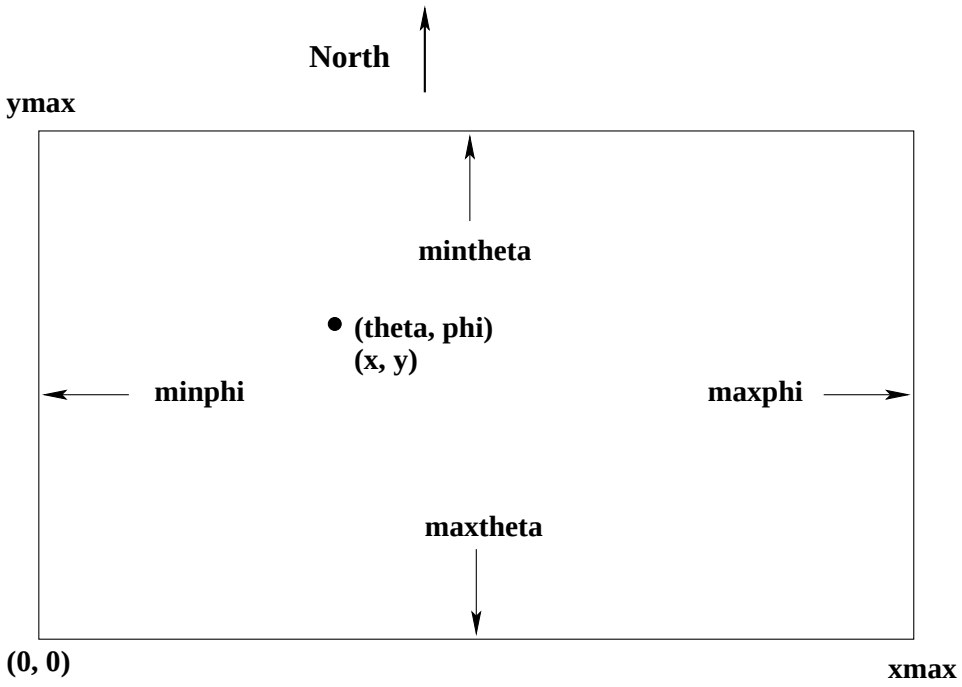
```
int hpic_pix2vec_ring(size_t nside, size_t pix,
                      double* xcomp, double* ycomp,
                      double* zcomp);
int hpic_pix2vec_nest(size_t nside, size_t pix,
                      double* xcomp, double* ycomp,
                      double* zcomp);
int hpic_vec2pix_ring(size_t nside, double xcomp,
                      double ycomp, double zcomp,
                      size_t *pix);
int hpic_vec2pix_nest(size_t nside, double xcomp,
                      double ycomp, double zcomp,
                      size_t *pix);
```

To degrade a pixel number at one NSIDE to a new pixel number at a (smaller) NSIDE, use the following functions

```
int hpic_degrade_nest(size_t oldnside, size_t oldpix,
                  size_t newnside, size_t *newpix);
int hpic_degrade_ring(size_t oldnside, size_t oldpix,
                  size_t newnside, size_t *newpix);
```

2.3.3 Projection Tools

When displaying maps, it is often necessary to project spherical coordinates onto a rectangular image. The following functions do forward and reverse projections of points to and from a rectangular projection of size `xmax` by `ymax`. The range of the projection in spherical coordinates is specified by the parameters `mintheta`, `maxtheta`, `minphi`, and `maxphi`. Note that since `theta` increases from North to South, the maximum `theta` value is actually at the bottom of the projection. Here is a diagram of the various projection parameters, and the functions to do cartesian and sinusoidal projections are below. Note that if the projection of `(theta,phi)` falls outside the range of the projection, then `HPIC_NULL` will be returned for values of `x` and `y`.




```
int hpic_proj_car(double mintheta, double maxtheta,
                  double minphi, double maxphi,
                  double xmax, double ymax, double theta,
                  double phi, double *x, double *y);
int hpic_proj_sin(double mintheta, double maxtheta,
                  double minphi, double maxphi,
                  double xmax, double ymax, double theta,
                  double phi, double *x, double *y);
```

```
int hpic_proj_rev_car(double mintheta, double maxtheta,
                      double minphi, double maxphi,
                      double xmax, double ymax, double x,
                      double y, double *theta, double *phi);
int hpic_proj_rev_sin(double mintheta, double maxtheta,
                      double minphi, double maxphi,
                      double xmax, double ymax, double x,
                      double y, double *theta, double *phi);
```

2.3.4 Location Tools

For some of the higher level functions it is useful to have low level tools that deal with pixel locations. In these functions the `order` parameter should be either `HPIC_RING` or `HPIC_NEST`. The `hpic_loc_dist` function returns the angular distance (in radians) between the centers of the two pixel indices. The `hpic_neighbors` function returns the pixel indices of the surrounding pixels. The vector of indices is resized to the proper length. The first four elements of the index vector contain the pixel numbers of neighbors that share a side with the specified pixel. The remaining indices in the vector are pixels that share a vertex with the specified pixel.

```
double hpic_loc_dist(size_t nside, int order, size_t pix1,
                     size_t pix2);
int hpic_neighbors(size_t nside, int ordering, size_t pixel,
                   hpic_vec_index *parray);
```

2.4 Maps

The HPIC library provides map types for double, float, and int maps. These structures and their associated functions provide an easy way to manipulate whole maps and perform complex operations. In addition to the data, map structures also store parameters associated with the map (NSIDE, name, units, ordering, coordinate system, etc). Because the underlying structure may change as new features are added, you should always use the provided access functions to change and retrieve these parameters.

2.4.1 Allocation

These functions allocate a map of double, float or int values and return a pointer to a new `hpic`, `hpic_float`, or `hpic_int` structure respectively. All maps are initialized to either `HPIC_NULL` (for double and float maps) or `HPIC_INT_NULL` (for

integer maps). The `order` parameter can be either `HPIC_NEST` or `HPIC_RING`. The `coord` parameter can take on values of `HPIC_COORD_C`, `HPIC_COORD_G`, `HPIC_COORD_E`, or `HPIC_COORD_O`. Currently the coordinate system variable is not used for anything except when writing the map to a FITS file, where it is printed as a keyword.

The `mem` parameter specifies the internal memory structure in which to store the data. If this is set to `HPIC_STND`, then the data will always be stored in a standard C array. If set to `HPIC_TREE`, then the data will always be stored in a nested tree structure. This tree storage is slower, but greatly reduces the memory requirements when working with high-resolution maps that have only a small coverage area. Of course, if you have a high coverage fraction in your map, then tree storage will actually use *more* memory. If the mem parameter is set to `HPIC_AUTO`, then the storage format will be automatically switched between the standard and tree formats to conserve memory. **NOTE: tree based storage is not yet implemented- all options currently have the behaviour of `HPIC_STND`.**

A newly allocated map has all of its elements set to `HPIC_NULL` (in the case of double or float maps) or `HPIC_INT_NULL` (in the case of integer maps).

```
hpic* hpic_alloc(size_t nside, int order, int coord,
                 int mem);
hpic_float* hpic_float_alloc(size_t nside, int order,
                 int coord, int mem);
hpic_int* hpic_int_alloc(size_t nside, int order,
                 int coord, int mem);
```

To free a map structure after it has been allocated, simply use the appropriate free command below.

```
int hpic_free(hpic* map);
int hpic_float_free(hpic_float* map);
int hpic_int_free(hpic_int* map);
```

2.4.2 Parameter Access

The following functions can be used to set and retrieve various parameter values stored in a map structure. To access the values of the "unchangeable" map parameters (NSIDE, number of pixels, ordering, coordinate system) that were set when the map was allocated, use the appropriate version (double, float, or int) of the following functions:

```
size_t hpic_nside_get(hpic *map);
size_t hpic_float_nside_get(hpic_float *map);
size_t hpic_int_nside_get(hpic_int *map);
```

```
size_t hpic_npix_get(hpic *map);
size_t hpic_float_npix_get(hpic_float *map);
size_t hpic_int_npix_get(hpic_int *map);

int hpic_order_get(hpic *map);
int hpic_float_order_get(hpic_float *map);
int hpic_int_order_get(hpic_int *map);

int hpic_coord_get(hpic *map);
int hpic_float_coord_get(hpic_float *map);
int hpic_int_coord_get(hpic_int *map);
```

It is not absolutely necessary to give each map a name, but if a name exists it will become the column title if the map is printed to a FITS file. To set or get the map name, use the following functions. Note that the "get" functions return a pointer to the char array that exists inside the map structure. This returned pointer is suitable for use in functions like strcpy, etc.

```
int hpic_name_set(hpic *map, const char *name);
int hpic_float_name_set(hpic_float *map, const char *name);
int hpic_int_name_set(hpic_int *map, const char *name);

char* hpic_name_get(hpic *map);
char* hpic_float_name_get(hpic_float *map);
char* hpic_int_name_get(hpic_int *map);
```

Another optional map parameter is the units of the map. This parameter is used to set the name of the units field when printing a map to a FITS file. To set and get the map units, use these functions

```
int hpic_units_set(hpic *map, const char *units);
int hpic_float_units_set(hpic_float *map,
                         const char *units);
int hpic_int_units_set(hpic_int *map, const char *units);

char* hpic_units_get(hpic *map);
char* hpic_float_units_get(hpic_float *map);
char* hpic_int_units_get(hpic_int *map);
```

The internal memory format of the data may be changed after a map is allocated. If the new memory format is not compatible with the current state of the data, the data will be remapped into either a standard C array or a tree structure. You can use the following functions to change or return the memory structure of a map.

```
int hpic_mem_set(hpic *map, int mem);
int hpic_float_mem_set(hpic_float *map, int mem);
```

```
int hpic_int_mem_set(hpic_int *map, int mem);

int hpic_mem_get(hpic *map);
int hpic_float_mem_get(hpic_float *map);
int hpic_int_mem_get(hpic_int *map);
```

2.4.3 Data Access

To set or return the values of individual pixels, you should use the following functions. The `pix` parameter is the pixel number (in the ordering of the map) which you wish to set or get. To set a pixel to a "NULL" value, simply set it to either `HPIC_NULL` or `HPIC_INT_NULL`, depending on whether the map is a floating point or integer type.

```
int hpic_set(hpic* map, size_t pix, double val);
int hpic_float_set(hpic_float* map, size_t pix, float val);
int hpic_int_set(hpic_int* map, size_t pix, int val);

double hpic_get(hpic* map, size_t pix);
float hpic_float_get(hpic_float* map, size_t pix);
int hpic_int_get(hpic_int* map, size_t pix);
```

To set all of the map elements to a certain value, use the following functions. **Caution: if your map is set to type `HPIC_TREE` and you set all pixels to a non-NULL value, then memory for the ENTIRE tree will be allocated! This will use many times more memory than a standard C vector!** If your map is set to type `HPIC_AUTO`, then it will be converted to a standard C vector on the fly as soon as it is more beneficial from a memory perspective.

```
int hpic_setall(hpic *map, double val);
int hpic_float_setall(hpic_float *map, float val);
int hpic_int_setall(hpic_int *map, int val);
```

2.4.4 Basic Operations

There are several simple operations that can be performed on map structures. To create a copy of a map, use the copy functions below. These will allocate a new map structure and fill it with the contents of the original. Both the original and the new maps will need to be freed with the appropriate function after use.

```
hpic* hpic_copy(hpic *map);
hpic_float* hpic_float_copy(hpic_float *map);
hpic_int* hpic_int_copy(hpic_int *map);
```

It is often useful to be able to compare two maps to see if they are equal. The following functions return 0 if the two maps are equal. They return 1 if the map data is equal, but the map parameters are different. A return value of 2 indicates that neither the parameters nor the data are equal.

```
int hpic_comp(hpic *map1, hpic *map2);
int hpic_float_comp(hpic_float *map1, hpic_float *map2);
int hpic_int_comp(hpic_int *map1, hpic_int *map2);
```

For debugging and informational purposes, it can be useful to display some basic information about the contents of a map. The functions below print the map parameters and the first and last elements of the map data to either a file pointer or stdout.

```
int hpic_info_fprintf(FILE *fp, hpic *map);
int hpic_float_info_fprintf(FILE *fp, hpic_float *map);
int hpic_int_info_fprintf(FILE *fp, hpic_int *map);

int hpic_info_printf(hpic *map);
int hpic_float_info_printf(hpic_float *map);
int hpic_int_info_printf(hpic_int *map);
```

2.5 Vectors

Although many different vector structures exist in various C libraries, I wanted to have a vector type that was entirely contained within the HPIC library. The following types can be used to store and manipulate a vector of double, float, int, or size_t values. I refer to the size_t vectors as "index" vectors, since they are usually used to store a vector of indices.

2.5.1 Allocation

Use one of these functions to allocate a vector of the appropriate type. The `n` parameter is the size of the vector. Note that it is possible to allocate a vector of zero size (useful if you plan to append data or resize later).

```
hpic_vec* hpic_vec_alloc(size_t n);
hpic_vec_float* hpic_vec_float_alloc(size_t n);
hpic_vec_int* hpic_vec_int_alloc(size_t n);
hpic_vec_index* hpic_vec_index_alloc(size_t n);
```

After you are finished with a vector, you can free the memory with one of the following commands

```
int hpic_vec_free(hpic_vec* vec);
int hpic_vec_float_free(hpic_vec_float* vec);
int hpic_vec_int_free(hpic_vec_int* vec);
int hpic_vec_index_free(hpic_vec_index* vec);
```

2.5.2 Data Access

To find the size of a vector that has already been allocated, use the function below that corresponds to the type of the vector.

```
size_t hpic_vec_n_get(hpic_vec *vec);
size_t hpic_vec_float_n_get(hpic_vec_float *vec);
size_t hpic_vec_int_n_get(hpic_vec_int *vec);
size_t hpic_vec_index_n_get(hpic_vec_index *vec);
```

To set a vector element to a given value, use one of these functions. The `elem` parameter is the index of the element you wish to change. The `val` parameter is the new value to assign to the element.

```
int hpic_vec_set(hpic_vec* vec, size_t elem, double val);
int hpic_vec_float_set(hpic_vec_float* vec, size_t elem,
                       float val);
int hpic_vec_int_set(hpic_vec_int* vec, size_t elem,
                       int val);
int hpic_vec_index_set(hpic_vec_index* vec, size_t elem,
                       size_t val);
```

In a similar fashion, you can retrieve the value of a vector element using one of these functions.

```
double hpic_vec_get(hpic_vec* vec, size_t elem);
float hpic_vec_float_get(hpic_vec_float* vec, size_t elem);
int hpic_vec_int_get(hpic_vec_int* vec, size_t elem);
size_t hpic_vec_index_get(hpic_vec_index* vec, size_t elem);
```

To set all elements of a vector to a certain value, use one of the "setall" functions below.

```
int hpic_vec_setall(hpic_vec *vec, double val);
int hpic_vec_float_setall(hpic_vec_float *vec, float val);
int hpic_vec_int_setall(hpic_vec_int *vec, int val);
int hpic_vec_index_setall(hpic_vec_index *vec, size_t val);
```

2.5.3 Basic Operations

There are several simple operations that can be performed on a vector. To create a copy of a vector, use one of the following commands. You should use the appro-

priate free command on both the original and the copy after you are finished using them.

```
hpic_vec* hpic_vec_copy(hpic_vec *vec);
hpic_vec_float* hpic_vec_float_copy(hpic_vec_float *vec);
hpic_vec_int* hpic_vec_int_copy(hpic_vec_int *vec);
hpic_vec_index* hpic_vec_index_copy(hpic_vec_index *vec);
```

Use the functions below to resize a vector. The parameter `newn` is the new size of the vector. If the new size is smaller than the current size, the vector is truncated. If the new size is larger than the current size, additional elements are added to the end of the vector and set to zero. Resizing a vector to size zero is *NOT the same* as freeing the vector.

```
int hpic_vec_resize(hpic_vec* vec, size_t newn);
int hpic_vec_float_resize(hpic_vec_float* vec,
                          size_t newn);
int hpic_vec_int_resize(hpic_vec_int* vec, size_t newn);
int hpic_vec_index_resize(hpic_vec_index* vec,
                          size_t newn);
```

To append a single element to a vector use the following functions. The vector size is increased by one, and the new element is set to the value specified by the `val` parameter.

```
int hpic_vec_append(hpic_vec* vec, double val);
int hpic_vec_float_append(hpic_vec_float* vec, float val);
int hpic_vec_int_append(hpic_vec_int* vec, int val);
int hpic_vec_index_append(hpic_vec_index* vec, size_t val);
```

2.6 Map Conversion

There are several different sorts of conversion operations one might want to do on a map. Besides converting between different types (double, float, and int), it is also desirable to convert between RING and NESTED ordering schemes and to be able to degrade and prograde a map.

2.6.1 Types

The following functions take a map of one type, allocate a map of the new type, and copy the contents from the old to the new. The old map is *NOT* freed. A pointer to the new map is returned. The new map has exactly the same name, units, ordering, etc as the old map. When converting from a floating point type to an int map, the map data is truncated (rounded down).

```
hpic_float* hpic_double2float(hpic *map);
hpic_int* hpic_double2int(hpic *map);

hpic* hpic_float2double(hpic_float *map);
hpic_int* hpic_float2int(hpic_float *map);

hpic* hpic_int2double(hpic_int *map);
hpic_float* hpic_int2float(hpic_int *map);
```

2.6.2 Ordering

The fastest way to convert the ordering of a map is to make a temporary map that has the new ordering and then convert all the pixels. Unfortunately this requires more memory. The following functions do an out-of-place conversion that makes use of a temporary copy of the map.

```
int hpic_conv_nestcopy(hpic *map);
int hpic_conv_ringcopy(hpic *map);

int hpic_conv_float_nestcopy(hpic_float *map);
int hpic_conv_float_ringcopy(hpic_float *map);

int hpic_conv_int_nestcopy(hpic_int *map);
int hpic_conv_int_ringcopy(hpic_int *map);
```

To save memory at the price of speed, it is also possible to do an in-place conversion. I am still working on the most efficient way of implementing this. **As of right now, these functions perform an out-of-place conversion**. This will be fixed soon.

```
int hpic_conv_nest(hpic *map);
int hpic_conv_ring(hpic *map);

int hpic_conv_float_nest(hpic_float *map);
int hpic_conv_float_ring(hpic_float *map);

int hpic_conv_int_nest(hpic_int *map);
int hpic_conv_int_ring(hpic_int *map);
```

2.6.3 Resolution

The following functions convert a map to a new NSIDE resolution. The functions return a pointer to a newly allocated map. The old map is not freed. The new map has the same name, units, and ordering, but has an NSIDE equal to the `newnside`

parameter. If the original map is being degraded, then each pixel of the new map will be an average of the high resolution pixel values that lie within it. Pixel values equal to `HPIC_NULL` or `HPIC_INT_NULL` will not be included in the average. If the original map is being prograded, then each pixel of the new map will have the same value as the low resolution pixel that it lies within.

```
hpic* hpic_conv_xgrade(hpic *map, size_t newnside);
hpic_float* hpic_conv_float_xgrade(hpic_float *map,
                                   size_t newnside);
hpic_int* hpic_conv_int_xgrade(hpic_int *map,
                               size_t newnside);
```

2.7 Map Math

It is often useful to perform simple mathematical operations on a map. The following tools abstract this process as much as possible to allow for a more intuitive useage.

2.7.1 Scaling and Offsets

The "scale" functions listed below will multiply every pixel in the map by the value specified in the `val` parameter. In the case of the integer map, the resulting product is truncated. The "offset" functions add the `val` parameter to all pixels in the map. Note that NULL pixels are not affected by these functions (they remain NULL).

```
int hpic_scale(hpic *map, double val);
int hpic_float_scale(hpic_float *map, double val);
int hpic_int_scale(hpic_int *map, double val);

int hpic_offset(hpic *map, double val);
int hpic_float_offset(hpic_float *map, float val);
int hpic_int_offset(hpic_int *map, int val);
```

2.7.2 Arithmetic

The functions listed below are used to add, subtract, multiply, or divide two maps of the same type (double, float, or int). In all cases, the data in the first map is replaced by the result of the operation. If the second map has a different ordering or NSIDE value, it is automatically converted to the same NSIDE and ordering as the first map before doing the operation. The `mode` parameter can take values of `HPIC_INTERSECT` or `HPIC_UNION`. If the intersection is requested, then only

pixels which are non-NULL in *both* maps will appear in the output (the rest of the output pixels will be set to NULL). If the union is requested, then all pixels that are non-NULL in any map will be included in the output. In the case of a union operation, pixels that exist in one map but not the other will be unmodified in the output.

```
int hpic_add(hpic *first, hpic *second, int mode);
int hpic_float_add(hpic_float *first, hpic_float *second,
                   int mode);
int hpic_int_add(hpic_int *first, hpic_int *second,
                 int mode);

int hpic_subtract(hpic *first, hpic *second, int mode);
int hpic_float_subtract(hpic_float *first,
                        hpic_float *second, int mode);
int hpic_int_subtract(hpic_int *first, hpic_int *second,
                      int mode);

int hpic_multiply(hpic *first, hpic *second, int mode);
int hpic_float_multiply(hpic_float *first,
                        hpic_float *second, int mode);
int hpic_int_multiply(hpic_int *first, hpic_int *second,
                      int mode);

int hpic_divide(hpic *first, hpic *second, int mode);
int hpic_float_divide(hpic_float *first,
                      hpic_float *second, int mode);
int hpic_int_divide(hpic_int *first, hpic_int *second,
                    int mode);
```

2.8 Projection

The projection types and routines found in HPIC are designed to allow easy projection of maps, sets of maps (vectorfields), sets of points, and sets of pixels onto a bitmap. These functions work, but add an "extra step" if you have to then copy this projection to another image for display purposes. If you are developing software to display or print healpix maps, you are probably better off creating functions that project a healpix map directly onto a display buffer, etc. Nevertheless, these functions at least show one way to implement projections of whole maps. In order to do this, I have defined a structure (`hpic_proj`) which contains the bitmap and various parameters associated with the projection.

2.8.1 Allocation

To allocate or free the projection structure, use the functions below. The `nx` and `ny` parameters are the dimensions in pixels of the projection.

```
hpic_proj* hpic_proj_alloc(size_t nx, size_t ny);
int hpic_proj_free(hpic_proj *proj);
```

2.8.2 Parameter Access

When a projection is allocated, it defaults to a cartesian projection. You can set or get the type of the projection with the functions below. Valid values of the `type` parameter are `HPIC_PROJ_CAR` and `HPIC_PROJ_SIN`. You can set or get the maximum and minimum range values of the projection by using the "range" functions below. See the earlier diagram for a description of what these range values mean. Note that setting the type or range of the projection also clears all of the projection data.

```
int hpic_proj_type_get(hpic_proj *proj);
int hpic_proj_type_set(hpic_proj *proj, int type);

int hpic_proj_range_get(hpic_proj *proj, double *mintheta,
                        double *maxtheta, double *minphi,
                        double *maxphi);
int hpic_proj_range_set(hpic_proj *proj, double mintheta,
                        double maxtheta, double minphi,
                        double maxphi);
```

2.8.3 Data Access

To set or get a specific value from the projection array, use the "set" and "get" functions below. The `xelem` and `yelem` parameters specify the array coordinates of the value you are changing or retrieving. You can also set all array elements with the "setall" command. To print the size, type, and range values of the projection to a file or stdout, you can use the print functions below.

```
int hpic_proj_set(hpic_proj *proj, size_t xelem,
                  size_t yelem, double val);
double hpic_proj_get(hpic_proj *proj, size_t xelem,
                     size_t yelem);

int hpic_proj_setall(hpic_proj *proj, double val);

int hpic_proj_info_fprintf(FILE *fp, hpic_proj *proj);
```

```
int hpic_proj_info_printf(hpic_proj *proj);
```

2.8.4 Projecting Maps and Vectors

To project the data of a single map onto a projection structure, simply call the `hpic_proj_map` function. The rest of the projection functions do not actually write any data to the array contained in the projection structure. Instead, they use the projection structure to obtain the type, size, and range of the projection. These functions return vectors of pixel indices. These index vectors must be allocated when you pass them to the functions, but their size is not important (they will be resized by the function).

In the case of `hpic_proj_points`, a set of ordered pairs of (`theta`, `phi`) values is projected into a set of ordered pairs of x and y pixel coordinates. The `x` and `y` vectors will contain the projection array coordinates of all the points lying within the range of the projection. Note that the length of the `x` and `y` vectors will be different than the length of the input vectors unless all the (`theta`, `phi`) values lie inside the range of the projection.

The `hpic_proj_pixels` function takes a list of map pixel indices at a certain NSIDE and ordering, and returns a list of projection array pixel coordinates that fall within the specified map pixels.

The `hpic_proj_vecfield` function takes two maps specifying the theta and phi components of a vector field. It returns four index vectors giving the projection array coordinates of the heads and tails of the vectors. The `pnside` parameter is the NSIDE of the vector field projection. If this is different from the NSIDE of the component maps, the components will be degraded to the desired NSIDE before projecting. The `maxmag` parameter specifies the magnitude of a vector which should approximately span the size of one pixel (at `pnside` resolution).

```
int hpic_proj_map(hpic_proj *proj, hpic *map);

int hpic_proj_points(hpic_proj *proj, hpic_vec *theta,
                     hpic_vec *phi, hpic_vec_index *x,
                     hpic_vec_index *y);

int hpic_proj_pixels(hpic_proj *proj, size_t nside,
                     int order, hpic_vec_index *pixels,
                     hpic_vec_index *x, hpic_vec_index *y);

int hpic_proj_vecfield(hpic_proj *proj, hpic *thetacomp,
                       hpic *phicomp, size_t pnside,
                       double maxmag,
                       hpic_vec_index *headx,
```

```
                        hpic_vec_index *heady,
                        hpic_vec_index *tailx,
                        hpic_vec_index *taily);
```

2.9 Transforms and Filtering

This section is the area of current active work. Now that the basic pixel and map operations are implemented, the next phase of development includes implementing harmonic transforms and various types of filtering. Look for more info in future releases...

2.10 FITS Reading and Writing

The FITS standard provides an incredibly flexible framework for storing data in a portable file. Over the years there have been many different formats used to store healpix style data. The goal of this portion of the HPIC library is to provide an extensible, easy to use set of tools to allow the reading and writing of all known healpix FITS formats, as well as allow the development of new formats as needed.

2.10.1 Format Specifications

Since there is no official "standard" for what constitutes a healpix FITS file, I have attempted to create a standard which encompasses all known existing types. Obviously compatibility with all the different codes in use is important. By making some minimal assumptions, I claim that this is a tractable problem. Note that these rules have changed slightly over time, to accomodate new formats. Here is a list of requirements that a healpix map file must follow:

1. Maps must be contained in the first extension (HDU 2) and not in the primary image.
2. The extension which contains the maps must be a binary table.
3. The extension must contain the string keyword "PIXTYPE", and the value of this keyword must be "HEALPIX".
4. The extension must contain the integer keyword "NSIDE", and this must have a value that corresponds to the NSIDE of the maps.
5. The extension must contain the string keyword "ORDERING", which can take values of either "RING" or "NESTED".

6. The extension should have the string keyword "COORDSYS". If it does not, a celestial/equatorial coordinate system is implied.

7. The extension should have the string keyword "INDXSCHM", which can take values of "IMPLICIT" or "EXPLICIT". If this keyword is not present, a value of "IMPLICIT" is assumed.

8. The extension should contain the integer keyword "GRAIN". If it does not, a value of zero is assumed.

9. All maps in the table must have the same NSIDE, ORDERING, and COORDSYS values.

10. In addition to the required keywords, any number of optional keywords are allowed.

11. A "full-sphere" FITS file should have INDXSCHM=IMPLICIT and GRAIN=0.

12. A "full-sphere" FITS file has one or more columns, where each column is a list of floating point pixel values for every pixel in a map.

13. If a "full-sphere" FITS file has NSIDE > 8, then the columns MAY be 1024 elements wide.

14. A "full-sphere" FITS file MIGHT contain only a contiguous portion of a map. Such a file MUST contain the keywords "NPIX", "FIRSTPIX", and "LASTPIX".

15. A "cut-sphere" FITS file should have INDXSCHM=EXPLICIT and GRAIN >= 1.

16. The first column of a "cut-sphere" FITS file is a list of pixel indices. After that follows one or more floating point columns containing map values at the indicated pixels. After the data columns is an integer column of hits/observations. The last column is a floating point column of error values. Note that there is no requirement on the actual name or units of the hits and error columns- only their type matters.

In addition to these FITS files that are designed to store maps, there are also various types of FITS files in use that store general vectors. I have created a specification that encompasses these "vector" FITS files as well:

1. Vectors must be stored in the first extension (HDU 2), and not in the primary image.

2. The extension which contains the vectors may be either an ASCII table or a binary table.

3. The extension may contain any number of optional keywords.

4. All vectors must have the same length.

5. If the vectors are stored in an ASCII table, then the table may consist of one or more columns of floating point numbers, where each column is a vector.

6. If the vectors are stored in a binary table, then the table may consist of *either* columns of floating point numbers *or* a single integer column of "index" values followed by columns of floating point numbers.

2.10.2 Optional Keys

All of the FITS file formats allow the user to specify optional keywords. To make this process relatively easy, the `hpic_keys` structure is used to hold any number of string, integer, and floating point keywords. After allocating an `hpic_keys` structure, you pass a pointer to this structure to the FITS reading or writing routines.

In the case of reading a FITS file, the keys structure will be populated by all the non-required keywords found in the file. You can then access any keyword values within the structure. In the case of writing a FITS file, you easily add keywords to the structure before calling the writing routine, and all the keywords in the structure will be written to the FITS file.

To allocate, free and clear (remove all keys from) a keys structure, use the following functions

```
hpic_keys* hpic_keys_alloc();
int hpic_keys_free(hpic_keys* keys);
int hpic_keys_clear(hpic_keys* keys);
```

To add or delete keys from an `hpic_keys` structure, use the functions below. The `keyname` parameter is a string containing the name of the key. The `keyval` parameter is the value of the key. Note that you must use the correct function depending on the type of key you are adding (string, integer or float). The string parameter `keycom` is the comment for the key.

```
int hpic_keys_sadd(hpic_keys *keys, char *keyname,
                   char *keyval, char *keycom);
int hpic_keys_iadd(hpic_keys *keys, char *keyname,
                   int keyval, char *keycom);
int hpic_keys_fadd(hpic_keys *keys, char *keyname,
                   float keyval, char *keycom);
int hpic_keys_del(hpic_keys *keys, char *keyname);
```

To retrieve a key value from the structure, you can use the key name and one of the "find" routines below. The value of the key and the comment string are returned. If the find routine is successful, the return value of the function is zero. Otherwise the function returns a value of 1. Obviously for the integer and floating point functions, the `keyval` parameter should be the address of an existing variable. To find out the total number of keys (of all types) stored in a structure, use the `hpic_keys_total` function.

```
int hpic_keys_total(hpic_keys *keys);
int hpic_keys_sfind(hpic_keys *keys, char *keyname,
                    char *keyval, char *keycom);
int hpic_keys_ifind(hpic_keys *keys, char *keyname,
                    int *keyval, char *keycom);
int hpic_keys_ffind(hpic_keys *keys, char *keyname,
                    float *keyval, char *keycom);
```

It is sometimes useful to print the contents of a keys structure to a file or stdout. This can be accomplished by the two functions below.

```
int hpic_keys_fprintf(FILE *fp, hpic_keys *keys);
int hpic_keys_printf(hpic_keys *keys);
```

2.10.3 Map FITS I/O

To test an existing FITS file to see if it is compatible with the healpix map specification, use the `hpic_fits_map_test` function. This function returns the NSIDE, ordering, coordinate system, and type of the file, as well as the number of *signal* maps in the file (for example, a four column, cut-sphere file has only one signal map). The returned `type` parameter will have a value equal to either `HPIC_FITS_FULL`, `HPIC_FITS_CHUNK` or `HPIC_FITS_CUT`. The return value of the function will be equal to 1 if the file is a supported type, and zero if it is not recognized.

```
int hpic_fits_map_test(char *filename, size_t *nside,
                       int *order, int *coord,
                       int *type, size_t *nmaps);
```

If you want to check file's compatibility but need slightly more information about the file, such as the names of the columns or values of keys in the header, you can use the `hpic_fits_map_info` function.

```
int hpic_fits_map_info(char *filename, size_t * nside,
                       int *order, int *coord, int *type,
                       size_t * nmaps, char *creator,
                       char *extname, char **names,
                       char **units, hpic_keys *keys);
```

To quickly read a single signal map from a file, use the `hpic_fits_read_one` function. A new float map is allocated and returned by the function. The `keys` parameter should point to an existing structure (allocated with `hpic_keys_alloc`). After the function call, any optional keywords in the file will be stored in the `keys` structure. The `mapnum` parameter is the (zero-based) number of the map you want to retrieve. The `creator` parameter returns the creator string found in the file.

```
hpic_float* hpic_fits_read_one(char *filename,
                               size_t mapnum,
                               char *creator,
                               hpic_keys *keys);
```

The FITS reading and writing functions below are as general as possible, and work with files containing any number of signal columns. This generality comes at the cost of ease of use. If you are only working with one of the currently existing formats used in CMB research, you will probably prefer to use the cmb-specific FITS functions in the next section.

Before reading or writing general map FITS files, you will need to allocate enough maps to hold the data in the file. The pointers to these maps are then stored inside a special structure, which is passed to the reading and writing functions. This is somewhat awkward, and is necessary for two reasons:

1. C cannot (easily) handle a variable number of function arguments
2. When wrapping C code with SWIG for use in scripting languages, it is much more difficult to pass an array of struct pointers than a single struct pointer

For map FITS files, the `hpic_fltarr` structure is used to pass an array of `hpic_float*` values to the FITS I/O routines. To allocate and manipulate this simple structure, you can use the following functions

```
hpic_fltarr* hpic_fltarr_alloc(size_t nmaps);
int hpic_fltarr_free(hpic_fltarr* array);

size_t hpic_fltarr_n_get(hpic_fltarr *array);

int hpic_fltarr_set(hpic_fltarr* array, size_t elem,
                    hpic_float* map);
hpic_float* hpic_fltarr_get(hpic_fltarr* array,
                            size_t elem);
```

Note that this structure is just a "container" for holding map pointers- it *does not* handle the allocation or freeing of the maps themselves. The exact use of this structure will become more clear if you examine its use in the src/test/hpictest.c test file.

To write or read a file containing full-sphere maps, use the following functions. As previously mentioned, the user is responsible for allocating and populating the `hpic_fltarr` structure with pointers to maps that have been allocated. The `keys` parameter should also already be allocated. **NOTE:** When writing FITS files, the column names and units in the output file are set from the names and units of the input maps.

```
int hpic_fits_full_write(char *filename, char *creator,
                         char *extname, char *comment,
                         hpic_fltarr *maps,
                         hpic_keys *keys);

int hpic_fits_full_read(char *filename, char *creator,
                        char *extname, hpic_fltarr *maps,
                        hpic_keys *keys);
```

To write or read a file containing cut-sphere maps, use the functions below. In addition to the array of signal maps, the user also specifies maps for the pixel indices, hits, and errors. The hits, errors, and all signal maps are cut based on the contents of the index map. Every pixel in the index map equal to `HPIC_INT_NULL` will be cut in the final output file. All non-null pixels will be included in the output file.

```
int hpic_fits_cut_write(char *filename, char *creator,
                        char *extname, char *comment,
                        hpic_int *pixels, hpic_int *hits,
                        hpic_float *errs,
                        hpic_fltarr *maps,
                        hpic_keys *keys);

int hpic_fits_cut_read(char *filename, char *creator,
                       char *extname, hpic_int *pixels,
                       hpic_int *hits, hpic_float *errs,
                       hpic_fltarr *maps,
                       hpic_keys *keys);
```

2.10.4 Vector FITS I/O

To test an existing FITS file to see if it is compatible with the healpix vector specification, use the `hpic_fits_vec_test` function. This function returns the number of floating point vector columns, the length of the vectors (number of rows in the table), the type of the file, and the type of the table. The returned `filetype` parameter will equal `HPIC_FITS_VEC` (if the file contains only floating point vectors) or `HPIC_FITS_VEC_INDX` (if the file has an integer index vector). The returned

tabtype parameter will equal HPIC_FITS_BIN or HPIC_FITS_ASCII. If you need more information about the file, use hpic_fits_vec_info

```
int hpic_fits_vec_test(char *filename, size_t *nvecs,
                       size_t *length, int *filetype,
                       int *tabtype);
int hpic_fits_vec_info(char *filename, size_t * nvecs,
                       size_t * length, int *filetype,
                       int *tabtype, char *creator,
                       char *extname, char **names,
                       char **units, hpic_keys * keys);
```

Note that a full-sphere healpix map format FITS file is *also* a valid healpix vector format FITS file. If you are unsure what kind of file you have, first check to see if it is a map format file.

As with the map functions that read or write any number of data columns to a FITS file, the vector FITS functions also require the user to allocate all necessary vectors ahead of time, and to then pass an array of vector pointers to the reading and writing functions. The hpic_vec_fltarr structure is used to store these pointers, and functions the same way as the hpic_fltarr structure.

```
hpic_vec_fltarr* hpic_vec_fltarr_alloc(size_t nvecs);
int hpic_vec_fltarr_free(hpic_vec_fltarr* array);

size_t hpic_vec_fltarr_n_get(hpic_vec_fltarr *array);

int hpic_vec_fltarr_set(hpic_vec_fltarr* array,
                        size_t elem, hpic_vec_float* vec);
hpic_vec_float* hpic_vec_fltarr_get(hpic_vec_fltarr* array,
                                    size_t elem);
```

To read and write a standard vector FITS file containing any number of floating point columns, use the following functions. The tabtype parameter is the desired type of the FITS table. The vecnames and vecunits parameters are arrays of strings containing the names and units of the vectors. These labels will be written to or read from the FITS file. You can allocate these arrays of strings using the hpic_strarr_alloc function if you wish.

```
int hpic_fits_vec_write(char *filename, int tabtype,
                        char *creator, char *extname,
                        char *comment,
                        hpic_vec_fltarr *vecs,
                        char **vecnames, char **vecunits,
                        hpic_keys *keys);

int hpic_fits_vec_read(char *filename, char *creator,
```

```
                        char *extname,
                        hpic_vec_fltarr *vecs,
                        char **vecnames, char **vecunits,
                        hpic_keys *keys);
```

To read and write vector FITS files that also contain an initial index column, use the functions below. Since this type of FITS file can only be constructed using a binary table, there is no parameter to specify the table type.

```
int hpic_fits_vecindx_write(char *filename, char *creator,
                            char *extname, char *comment,
                            hpic_vec_int *indx,
                            hpic_vec_fltarr *vecs,
                            char **vecnames,
                            char **vecunits,
                            hpic_keys *keys);

int hpic_fits_vecindx_read(char *filename, char *creator,
                           char *extname,
                           hpic_vec_int *indx,
                           hpic_vec_fltarr *vecs,
                           char **vecnames, char **vecunits,
                           hpic_keys *keys);
```

2.11 CMB Specific Functions

The field of Cosmic Microwave Background (CMB) research has obviously been one of the major application areas of the healpix algorithms. I have attempted to separate functions that are CMB-specific from those that are generally applicable to pixelized functions on the sphere.

2.11.1 Projection

As a special case of vector field projection, you can use the `hpic_cmb_proj_QU` function to project the Q and U stokes parameter components of a polarization field onto a projection structure. Since this quantity is a "headless" vector, it is irrelevant which end is considered the head or tail.

```
int hpic_cmb_proj_QU(hpic_proj *proj, hpic *Qmap,
                     hpic *Umap, size_t pnside,
                     double maxmag,
                     hpic_vec_index *headx,
                     hpic_vec_index *heady,
                     hpic_vec_index *tailx,
```

```
                        hpic_vec_index *taily);
```

2.11.2 Specialized FITS I/O

Since there are already certain common FITS formats in use in the CMB community, it is convenient to have some reading and writing functions that deal specifically with these formats (instead of always having to use the general functions). **NOTE:** When writing FITS files, the column names and units in the output file are set from the names and units of the input maps.

To write a single, full-sky map, use the `hpic_cmb_write_full` function. All the parameters have the usual definitions (see the general FITS section for more info). To read a file such as this, use the `hpic_fits_read_one` function discussed previously.

```
int hpic_cmb_write_full(char *filename, hpic_float *data,
                        char *comment, char *creator,
                        hpic_keys *keys);
```

To read and write a set of full-sky T, Q, and U maps, use the following functions.

```
int hpic_cmb_write_fullTQU(char *filename,
                           hpic_float *tdata,
                           hpic_float *qdata,
                           hpic_float *udata, char *comment,
                           char *creator, hpic_keys *keys);

int hpic_cmb_read_fullTQU(char *filename, hpic_float *tdata,
                          hpic_float *qdata,
                          hpic_float *udata, char *creator,
                          hpic_keys *keys);
```

To read and write a single cut-sky map, use the following functions.

```
int hpic_cmb_write_cut(char *filename, hpic_int *pix,
                       hpic_float *data, hpic_int *hits,
                       hpic_float *errs, char *comment,
                       char *creator, hpic_keys *keys);

int hpic_cmb_read_cut(char *filename, hpic_int *pix,
                      hpic_float *data, hpic_int *hits,
                      hpic_float *errs, char *creator,
                      hpic_keys *keys);
```

To read and write a set of cut-sky T, Q, and U maps, use the following functions.

```
int hpic_cmb_write_cutTQU(char *filename, hpic_int *pix,
```

```
                          hpic_float *tdata,
                          hpic_float *qdata,
                          hpic_float *udata, hpic_int *hits,
                          hpic_float *errs, char *comment,
                          char *creator, hpic_keys *keys);

int hpic_cmb_read_cutTQU(char *filename, hpic_int *pix,
                         hpic_float *tdata,
                         hpic_float *qdata,
                         hpic_float *udata, hpic_int *hits,
                         hpic_float *errs, char *creator,
                         hpic_keys *keys);
```

2.12 Compatibility Wrappers

This set of functions is intended to be used as a "transitional" step for software projects that have been using the CHEALPIX tools and wish to start using the HPIC library. These functions are wrappers around the standard HPIC functions, so they may be slower than calling the native functions. The goal is to allow the user to do several regular-expression substitution operations on a source tree and simply recompile.

2.12.1 Pixel Tools

The compatibility pixel tools (should) have exactly the same calling sequence as those in CHEALPIX . The only difference is that the functions begin with the "compat_" character sequence.

```
void compat_ang2pix_nest(const long nside, double theta,
                         double phi, long *ipix);
void compat_ang2pix_ring(const long nside, double theta,
                         double phi, long *ipix);
void compat_pix2ang_nest(long nside, long ipix,
                         double *theta, double *phi);
void compat_pix2ang_ring(long nside, long ipix,
                         double *theta, double *phi);
void compat_nest2ring(long nside, long ipnest,
                      long *ipring);
void compat_ring2nest(long nside, long ipring,
                      long *ipnest);
long compat_nside2npix(const long nside);
```

2.12.2 Legacy FITS I/O

These functions support the simple full-sky FITS format found in CHEALPIX , as well as the cut-sky format used in Boomerang and other experiments. Again, the only difference is the "compat_" string prepended to the function names.

```
long compat_get_fits_size(char *filename, long *nside,
                          char *ordering);

int compat_read_healpix_map(char *infile, long *nside,
                            char *coordsys, char *ordering,
                            float *map);
int compat_write_healpix_map(float *signal, long nside,
                             char *filename, char nest,
                             char *coordsys);

void compat_read_fits_cut4(char *filename, int *pixel,
                           float *signal, int *n_obs,
                           float *serror, char *channel,
                           char *coordsys, int *nside,
                           int *obs_npix, char *ordering,
                           char *target, int *shot,
                           char *units);
int compat_write_fits_cut4(int *pixel, float *signal,
                           int *n_obs, float *serror,
                           char *channel, char *coordsys,
                           int nside, int obs_npix,
                           char *ordering, char *target,
                           int shot, char *units,
                           char *filename, char *creator,
                           char *release);
```

Chapter 3

Perl Reference

In order to provide a Perl interface to the HPIC library, I have made extensive use of SWIG (Simplified Wrapper and Interface Generator) to build a Perl module that "wraps" the C library. After building and installing the module, you can use it in your programs just like any other module:

```
use HPIC;
```

Since Perl supports object oriented programming better than C, I have incorporated many of the functions into the types they operate on. This changes the calling sequence somewhat, but the interface is much cleaner. Aside from this change, the Perl interface is so similar to the C interface that I will not go into great detail about the parameters of each function. See the corresponding section of the C reference for more details. This chapter will focus on the differences between the Perl and C interfaces. The Perl interface may change in the future as I become more knowledgeable about converting types between C and Perl. For example, it would be nice to have HPIC vectors mapped directly to Perl arrays and vice versa. Right now this is not a very high priority, since the current scheme *works*, even if it is a bit clumsy.

For examples of many of the following functions, see the "test.pl" program in the src/perl subdirectory.

3.1 Constants and Low Level Functions

All of the constants defined in the C library are available under Perl as read-only variables. For example, to print the value of PI, you could do

```
print "$HPIC_PI\n";
```

The low level functions work the same as the C version. The only difference is that the function return values are returned as an array. The following listing shows how these functions should be called.

```
$result = hpic_is_dnull($value);
$result = hpic_is_fnull($value);
$result = hpic_is_inull($value);
$result = hpic_nsidecheck($nside);

$npix = hpic_nside2npix($nside);
$nside = hpic_npix2nside($npix);

$pring = hpic_nest2ring($nside, $pnest);
$pnest = hpic_ring2nest($nside, $pring);
($err,$theta,$phi) = hpic_pix2ang_ring($nside, $pix);
($err,$theta,$phi) = hpic_pix2ang_nest($nside, $pix);
$pix = hpic_ang2pix_ring($nside, $theta, $phi);
$pix = hpic_ang2pix_nest($nside, $theta, $phi);
($err,$theta,$phi) = hpic_vec2ang($xcomp, $ycomp, $zcomp);
($err,$xcomp,$ycomp,$zcomp) = hpic_ang2vec($theta, $phi);
($err,$xcomp,$ycomp,$zcomp) = hpic_pix2vec_ring($nside,
                                                $pix);
($err,$xcomp,$ycomp,$zcomp) = hpic_pix2vec_nest($nside,
                                                $pix);
$pix = hpic_vec2pix_ring($nside, $xcomp, $ycomp, $zcomp);
$pix = hpic_vec2pix_nest($nside, $xcomp, $ycomp, $zcomp);

$newpix = hpic_degrade_nest($oldnside, $oldpix, $newnside);
$newpix = hpic_degrade_ring($oldnside, $oldpix, $newnside);

($err,$x,$y) = hpic_proj_car($mintheta, $maxtheta, $minphi,
                             $maxphi, $xmax, $ymax, $theta,
                             $phi);
($err,$x,$y) = hpic_proj_sin($mintheta, $maxtheta, $minphi,
                             $maxphi, $xmax, $ymax, $theta,
                             $phi);
($err,$theta,$phi) = hpic_proj_rev_car($mintheta, $maxtheta,
                                       $minphi, $maxphi,
                                       $xmax, $ymax, $x, $y);
($err,$theta,$phi) = hpic_proj_rev_sin($mintheta, $maxtheta,
                                       $minphi, $maxphi,
                                       $xmax, $ymax, $x, $y);
```

3.2 Maps

In Perl, the HPIC map structures have been wrapped into classes, and many of the functions that operate on the structures are now part of the class. This means that the calling sequence and names of most of the functions have changed slightly. It should still be clear which Perl function corresponds to a certain C function. The listing below includes the three map classes and their functions. Note how a new map is allocated and destroyed.

```
# double map
$dmap = new HPIC::hpic($nside, $order, $coord, $mem);
$dmap->DESTROY();

$dmap->map_mem_set($mem);
$mem = $dmap->map_mem_get();
$dmap->map_name_set($name);
$dmap->map_units_set($units);
$name = $dmap->map_name_get();
$units = $dmap->map_units_get();
$nside = $dmap->map_nside_get();
$npix = $dmap->map_npix_get();
$order = $dmap->map_order_get();
$coord = $dmap->map_coord_get();
$dmap->map_set($pix, $val);
$val = $dmap->map_get($pix);
$dmap->map_setall($val);
$dmapcopy = $dmap->map_copy();
$dmap->map_scale($val);
$dmap->map_offset($val);
$fmap = $dmap->to_float();
$imap = $dmap->to_int();
$dmap->nestcopy();
$dmap->ringcopy();
$dmap->nest();
$dmap->ring();
$dmapgrade = $dmap->xgrade($newnside);
$dmap->printf();

# float map
$fmap = new HPIC::hpic_float($nside, $order, $coord, $mem);
$fmap->DESTROY();

$fmap->map_mem_set($mem);
$mem = $fmap->map_mem_get();
$fmap->map_name_set($name);
```

```
$fmap->map_units_set($units);
$name = $fmap->map_name_get();
$units = $fmap->map_units_get();
$nside = $fmap->map_nside_get();
$npix = $fmap->map_npix_get();
$order = $fmap->map_order_get();
$coord = $fmap->map_coord_get();
$fmap->map_set($pix, $val);
$val = $fmap->map_get($pix);
$fmap->map_setall($val);
$fmapcopy = $fmap->map_copy();
$fmap->map_scale($val);
$fmap->map_offset($val);
$dmap = $fmap->to_double();
$imap = $fmap->to_int();
$fmap->nestcopy();
$fmap->ringcopy();
$fmap->nest();
$fmap->ring();
$fmapgrade = $fmap->xgrade($newnside);
$fmap->printf();

# int map
$imap = new HPIC::hpic_int($nside, $order, $coord, $mem);
$imap->DESTROY();

$imap->map_mem_set($mem);
$mem = $imap->map_mem_get();
$imap->map_name_set($name);
$imap->map_units_set($units);
$name = $imap->map_name_get();
$units = $imap->map_units_get();
$nside = $imap->map_nside_get();
$npix = $imap->map_npix_get();
$order = $imap->map_order_get();
$coord = $imap->map_coord_get();
$imap->map_set($pix, $val);
$val = $imap->map_get($pix);
$imap->map_setall($val);
$imapcopy = $imap->map_copy();
$imap->map_scale($val);
$imap->map_offset($val);
$dmap = $imap->to_double();
$fmap = $imap->to_float();
$imap->nestcopy();
```

```
$imap->ringcopy();
$imap->nest();
$imap->ring();
$imapgrade = $imap->xgrade($newnside);
$imap->printf();
```

3.3 Vectors

The vector structures in Perl have also been wrapped into classes. The listing below shows the syntax of their use.

```
# double vector
$dvec = new HPIC::hpic_vec($n);
$dvec->DESTROY();
$n = $dvec->vec_n_get();
$dvec->vec_set($elem, $val);
$val = $dvec->vec_get($elem);
$dvec->vec_setall($val);
$dveccopy = $dvec->vec_copy();
$dvec->vec_resize($newn);
$dvec->vec_append($val);

# float vector
$fvec = new HPIC::hpic_vec_float($n);
$fvec->DESTROY();
$n = $fvec->vec_n_get();
$fvec->vec_set($elem, $val);
$val = $fvec->vec_get($elem);
$fvec->vec_setall($val);
$fveccopy = $fvec->vec_copy();
$fvec->vec_resize($newn);
$fvec->vec_append($val);

# int vector
$ivec = new HPIC::hpic_vec_int($n);
$ivec->DESTROY();
$n = $ivec->vec_n_get();
$ivec->vec_set($elem, $val);
$val = $ivec->vec_get($elem);
$ivec->vec_setall($val);
$iveccopy = $ivec->vec_copy();
$ivec->vec_resize($newn);
$ivec->vec_append($val);
```

```
# index (size_t) vector
$xvec = new HPIC::hpic_vec_index($n);
$xvec->DESTROY();
$n = $xvec->vec_n_get();
$xvec->vec_set($elem, $val);
$val = $xvec->vec_get($elem);
$xvec->vec_setall($val);
$xveccopy = $xvec->vec_copy();
$xvec->vec_resize($newn);
$xvec->vec_append($val);
```

3.4 Math and Miscellany

These functions are grouped together because they are basically identical to their C counterparts, and there is not much to say about them. For completeness, here is how they look in Perl, but there is nothing new here.

```
# map comparison
$result = hpic_comp($dmap1, $dmap2);
$result = hpic_float_comp($fmap1, $fmap2);
$result = hpic_int_comp($imap1, $imap2);

# pixel location
$dist = hpic_loc_dist($nside, $order, $pix1, $pix2);
$err = hpic_neighbors($nside, $order, $pixel, $parray);

# math
hpic_add($first, $second, $mode);
hpic_float_add($first, $second, $mode);
hpic_int_add($first, $second, $mode);
hpic_subtract($first, $second, $mode);
hpic_float_subtract($first, $second, $mode);
hpic_int_subtract($first, $second, $mode);
hpic_multiply($first, $second, $mode);
hpic_float_multiply($first, $second, $mode);
hpic_int_multiply($first, $second, $mode);
hpic_divide($first, $second, $mode);
hpic_float_divide($first, $second, $mode);
hpic_int_divide($first, $second, $mode);
```

3.5 Projection

Projection of maps, points, etc is straightforward in Perl. The projection structure has been wrapped into a class, and the functions that do the the projection are basically the same as in C.

```
# projection
$proj = new HPIC::hpic_proj($nx, $ny);
$proj->DESTROY();
$type = $proj->proj_type_get();
$proj->proj_type_set($type);
($err,$mintheta,$maxtheta,$minphi,$maxphi) =
                              $proj->proj_range_get();
$proj->proj_range_set($mintheta, $maxtheta, $minphi,
                      $maxphi);
$proj->proj_set($xelem, $yelem, $val);
$val = $proj->proj_get($xelem, $yelem);
$proj->proj_setall($val);
$proj->printf();

hpic_proj_points($proj, $theta, $phi, $x, $y);
hpic_proj_pixels($proj, $nside, $order, $pixels, $x, $y);
hpic_proj_map($proj, $map);
hpic_proj_vecfield($proj, $thetacomp, $phicomp, $pnside,
                   $maxmag, $headx, $heady, $tailx, $taily);
```

3.6 Transforms and Filtering

Since the C functions are not implemented yet, the Perl wrappers certainly don't work...

3.7 FITS I/O

The steps for reading and writing FITS files in Perl are analogous to those in C. You must still allocate space for the maps/vectors, the map/vector array, and the optional keys. See the test.pl file for an example of how to put all these pieces together.

```
# array of float maps
$maps = new HPIC::hpic_fltarr($n);
$maps->DESTROY();
$maps->array_set($elem, $map);
$map = $maps->array_get($elem);
```

```
# array of float vectors
$vecs = new HPIC::hpic_vec_fltarr($n);
$vecs->DESTROY();
$vecs->array_set($elem, $vec);
$vec = $vecs->array_get($elem);

# FITS keys
$keys = new HPIC::hpic_keys();
$keys->DESTROY();
$keys->keys_clear();
$keys->keys_sadd($keyname, $keyval, $keycom);
$keys->keys_iadd($keyname, $keyval, $keycom);
$keys->keys_fadd($keyname, $keyval, $keycom);
$keys->keys_del($keyname);
$total = $keys->keys_total();
($err,$result,$keyval,$keycom) = $keys->keys_sfind($keyname);
($err,$result,$keyval,$keycom) = $keys->keys_ifind($keyname);
($err,$result,$keyval,$keycom) = $keys->keys_ffind($keyname);
$keys->keys_printf();

# map FITS functions
($err,$result,$nside,$order,$coord,$type,$nmaps) =
             hpic_fits_map_test($filename);
($err,$result,$nside,$order,$coord,$type,$nmaps) =
             hpic_fits_map_info($filename, $creator,
                                $extname, $names, $units,
                                $keys);
hpic_fits_full_write($filename, $creator, $extname,
                     $comment, $maps, $keys);
hpic_fits_cut_write($filename, $creator, $extname,
                     $comment, $pixels, $hits, $errs,
                     $maps, $keys);
hpic_fits_full_read($filename, $creator, $extname, $maps,
                    $keys);
hpic_fits_cut_read($filename, $creator, $extname, $pixels,
                   $hits, $errs, $maps, $keys);
$fmap = hpic_fits_read_one($filename, $mapnum, $creator,
                           $keys);

# vector FITS functions
($err,$result,$nvecs,$length,$filetype,$tabtype) =
      hpic_fits_vec_test($filename);
($err,$result,$nvecs,$length,$filetype,$tabtype) =
      hpic_fits_vec_info($filename, $creator, $extname,
```

```
                          $names, $units, $keys);
hpic_fits_vec_write($filename, $tabtype, $creator, $extname,
                    $comment, $vecs, $vecnames, $vecunits,
                    $keys);
hpic_fits_vecindx_write($filename, $creator, $extname,
                        $comment, $indx, $vecs, $vecnames,
                        $vecunits, $keys);
hpic_fits_vec_read($filename, $creator, $extname, $vecs,
                   $vecnames, $vecunits, $keys);
hpic_fits_vecindx_read($filename, $creator, $extname, $indx,
                       $vecs, $vecnames, $vecunits, $keys);
```

3.8 CMB Specific

The Perl CMB functions are identical to the C functions. They are listed below for completeness.

```
hpic_cmb_proj_QU($proj, $Qmap, $Umap, $pnside, $maxmag,
                 $headx, $heady, $tailx, $taily);

hpic_cmb_write_full($filename, $data, $comment, $creator,
                    $keys);
hpic_cmb_write_fullTQU($filename, $tdata, $qdata, $udata,
                       $comment, $creator, $keys);
hpic_cmb_write_cut($filename, $pix, $data, $hits, $errs,
                   $comment, $creator, $keys);
hpic_cmb_write_cutTQU($filename, $pix, $tdata, $qdata,
                      $udata, $hits, $errs, $comment,
                      $creator, $keys);

hpic_cmb_read_fullTQU($filename, $tdata, $qdata, $udata,
                      $creator, $keys);
hpic_cmb_read_cut($filename, $pix, $data, $hits, $errs,
                  $creator, $keys);
hpic_cmb_read_cutTQU($filename, $pix, $tdata, $qdata,
                     $udata, $hits, $errs, $creator,
                     $keys);
```

Chapter 4

Command Line Tools

Although the HPIC library allows the user to manipulate healpix data in a variety of ways, some operations are very common. I have written a set of programs that perform many of the common tasks needed when working with healpix format FITS files.

4.1 General Tools

These functions are used to modify the data in a healpix FITS file. The `hpic_scale` program applies a scale factor and/or an overall offset to one of the signal maps in a healpix FITS file. The command line options are shown below.

```
hpic_scale

  -file <FITS file>
     This is the FITS file on which to operate.

  -map <map number to modify in file>
     This is the (zero-base) number of the signal
     map to be modified.

  -scale <factor by which to multiply map>
     Each pixel value in the signal map is multiplied
     by this number

  -offset <offset to add to map>
     This number is added to each pixel value in
     the signal map.
```

The `hpic_convert` program reads in one healpix map file, converts all maps to new NSIDE and/or ordering, and writes the converted maps to a new file.

```
hpic_convert

  -in <input FITS file>
    This is the file to read in.

  -out <output FITS file>
    This is the new file to be created.

  -nside <new nside value>
    This is the new NSIDE that all the maps will
    be converted to.

  -order <ordering (0=RING, 1=NEST)>
    This is the new ordering that all maps will be
    converted to.
```

The `hpic_thresh` program reads in a single map from a FITS file and cuts all pixels that lie outside a specified range. The remaining pixels are written to a new file and (optionally) set to a new value. If no upper or lower bound is set, then all pixels are kept.

```
hpic_thresh

  -in <input FITS file>
    This is the file to read in.

  -out <output FITS file>
    This is the new file that will be created.

  -mapnum <map number to process>
    The (zero-based) number of the input signal map.

  -upper <upper threshold>
    All pixels with a value higher than this number will
    be cut (set to HPIC_NULL).

  -lower <lower threshold>
    All pixels with a value lower than this number will
    be cut (set to HPIC_NULL).

  -outval <optionally set all in-range pixels to this value>
    If specified, all output pixels lying between upper and
    lower will be set to this value.
```

The `hpic_outline` program reads in a single map from a FITS file and finds all non-null pixels that border (with a side or vertex) all null-valued pixels. These pixels are written to a new file and (optionally) set to a new value. If the input contains no null-valued pixels, then the output map will be empty.

```
hpic_outline

  -in <input FITS file>
    This is the file to read in.

  -out <output FITS file>
    This is the new file that will be created.

  -mapnum <map number to process>
    The (zero-based) number of the input signal map.

  -outval <optionally set all border pixels to this value>
    If specified, all output pixels will be set to
    this value.
```

The `hpic_2ascii` program reads in any supported FITS file (map or vector), and dumps the contents to stdout.

```
hpic_2ascii <input FITS file>
```

The `hpic_2fits` program reads a specially formatted text file and creates a valid healpix FITS file.

```
hpic_2fits

  -out <output FITS file>
    The name of the FITS file to generate.

  -vec (build vector FITS file)
  -vecindx (build indexed vector FITS file)
  -full (build full-sphere map FITS file)
  -cut (build cut-sphere map FITS file)
    These 4 options specify what type of file
    you wish to build.  You must specify exactly
    one of these options.

  -nside <nside of maps>
    If you are building a map FITS file, then this
    is the NSIDE value of the maps.

  -order <0=NESTED 1=RING>
    If you are building a map FITS file, then this
```

```
    is the ordering of the maps.

  -table <0=BINARY 1=ASCII>
    If you are building a non-index vector FITS file,
    then this specifies the type of table to use.

  -units <units>
    The units of the signal columns.

  -coord <coordsys (eg. C, G, O)>
    If you are building a map FITS file, then this
    is the coordinate system to use.

  <input text file (properly formatted)>
    The final argument is the name of the input
    text file.
```

The formatting of the text file is as follows: the first line contains a comment character (the first word of the line is ignored) followed by three numbers. The first number is the number of optional keys. The second number is the total number of columns. The third number is the length of the columns. After the first line are a number of lines specifying optional key values. Each key line consists of a comment character, a word specifying the key type (SKEY, FKEY, or IKEY), the key name, the key value, and the rest of the line is treated as the key comment. For example, to generate a vector fits file with one column, I might use a text file like this:

```
# 3 1 2049
# SKEY CHANNEL B145W1 Simulated Beam for B145W1
# FKEY AZFWHM 9.75 Beam azimuth FWHM in arcmin
# FKEY ELFWHM 9.93 Beam elevation FWHM in arcmin
1.000000
0.999997
0.999991
0.999982
0.999970
0.999955
.
.
.
```

To build a cut-sphere map file, I might use a text file like this:

```
# 6 4 86098
# SKEY CHANNEL   B145W1   Channel of Observation
# SKEY RELEASE    1.0     Data Release
```

```
# SKEY TARGET    CMB       Observed Target
# SKEY OBJECT    PARTIAL   Sky coverage represented by data
# IKEY SHOT      1         Shot of this target
# IKEY OBS_NPIX  86098     Number of pixels observed
2348362  1.000000e+00  78  9.715862e-04
2348363  1.000000e+00  72  7.337384e-04
2348364  1.000000e+00  72  9.687704e-04
2348365  1.000000e+00  72  9.399473e-04
2348366  1.000000e+00  72  9.749437e-04
2348367  1.000000e+00  72  9.362744e-04
2348368  1.000000e+00  60  1.017587e-03
.
.
.
```

4.2 Simple Math Tools

These four programs take the sum, difference, product, or quotient of two maps. The first file is backed up and overwritten with the result of the operation.

```
hpic_sum OR hpic_diff OR hpic_prod OR hpic_quot

  -first <first FITS file>
    This is the first file to use.  A backup of
    this file will be created and the original
    will be overwritten by the result.

  -fmap <map number to use in first file>
    The (zero-based) signal map to use.

  -second <second FITS file>
    The second FITS file.

  -smap <map number to use in second file>
    The (zero-based) signal map to use from
    the second file.

  -union (keep union of maps)
    If specified, keep the union of the non-NULL
    map values.  Default is to keep the intersection.
```

References

[1] Krzysztof M. Górski, Eric Hivon, and Benjamin D. Wandelt. Analysis issues for large cmb data sets. In *Proceedings of the MPA/ESO Cosmology Conference*, 1999. 3

[2] Krzysztof M. Górski, Eric Hivon, Benjamin D. Wandelt, Frode K. Hansen, and Anthony J. Banday. *The HEALPix Primer*, 1.10 edition, March 2000. 3

[3] B.D. Wandelt, E. Hivon, and K. M. Górski. Topological analysis of high-resolution cmb maps. In *Fundamental Parameters in Cosmology, Proceedings of the 23rd Rencontres de Moriond*, 1998. Astro-ph/9803317. 3

[4] Frode K. Hansen, Benjamin D. Wandelt, Krzysztof M. Górski, Anthony J. Banday, and Eric Hivon. *HEALPix Fortran Facility Users Guide*, 1.10 edition, March 2000. 3

[5] Frode K. Hansen, Benjamin D. Wandelt, Krzysztof M. Górski, Eric Hivon, and Anthony J. Banday. *HEALPix Fortran90 Subroutines Overview*, 1.10 edition, March 2000. 3