The LibSQUID Software Library

James Wren
June 1, 2014

Introduction

The LibSQUID library consists of a set of functions written in C which allow a user to index a sphere at a specified resolution using a quad-tree indexing scheme. The indexes themselves are called SQUIDs, short for Spherical-cube Quad-tree Unique ID. Each SQUID index directly corresponds to a point on the sphere given by a (longitude,latitude) coordinate. In addition, the SQUID defines a region on the sphere, which I call a "tile", that has well defined boundaries. A SQUID tile is the equivalent to a pixel on an image, except that in this case, the image is the entire sphere.

The spherical coordinate and boundary of a particular SQUID depends on the map projection for which the SQUID is defined. As the name suggests, SQUIDs are defined in terms of "spherical-cube" map projections in which the sphere is divided into six square faces. These faces are then divided according to a quad-tree indexing scheme down to a resolution level, k, defined by the user. At each resolution level, the sphere is divided into $6*2^{2k}$ pixels. The 6 spherical-cube faces correspond to the base resolution, k=0. Each pixel corresponds to a SQUID index/tile. These indices are unique, regardless of the resolution level chosen by the user.

Finally, the SQUID tiles themselves can be further subdivided into pixels producing an image. The SQUID images can then be used to efficiently store and retrieve spherical imagery. Because SQUID tiles have a well defined geometry, the user can immediately convert a pixel coordinate (x,y) on the SQUID tile image to a coordinate (longitude, latitude) on the sphere. Additionally, SQUID tile images created for the same SQUID index (i.e. a time-series of images, or different spectral channels) may be combined into an image cube for even more efficient storage and recall.

The primary motivation for the LibSQUID system is the archival of high frame rate spherical data, in particular wide-field astronomical time-series imagery. The goal is to archive these data efficiently by re-projecting them onto a standardized 2-D projection, preferably equal-area, indexed in a quad-tree manner, and combined into image cubes for fast retrieval of imagery located closely in time and space.

Other Spherical Indexing Systems

Richard White and Sally Stemwedel (1992) proposed a system for using a quadrilateralized spherical cube projection and a quad-tree indexing scheme to divide the sphere. Using the quadrilateralized cube projection ensures that the individual pixels are equal area. The technique detailed in their paper is largely the technique I've employed in the LibSQUID library. They also proposed a set of FITS header keywords that might be employed to efficiently reference these data. However, it is unclear whether their suggestions were ever adopted or distributed as a software library. The quadrilateralized spherical cube projection is one of the projections supported by LibSQUID.

Koposov, et al., (2006) have developed the ideas of White and Stemwedel into the Q3C software package. This package is primarily developed as a PostgreSQL extension for doing fast index searches in a pixelated sphere. While this system is very similar to the technique employed by the LibSQUID library, there are a few differences. First, the Q3C package is primarily designed to work with SQL databases and is not, in itself, a stand-alone system. Second, the paper and code documentation do not clearly state which projection is being employed for the quad-tree scheme. I believe it is the tangental projection, which is not equal-area. Finally, the Q3C system

pixelates the sphere but it is not clear from the documentation how it handles image data. It appears to be used mainly for handling spatial queries of tabulated spherical data (i.e. catalogs, object lists). The LibSQUID library also supports the tangental spherical cube projection.

Szalay, et al., (2005) have also developed a spherical indexing system called the Hierarchical Triangle Mesh (HTM). This system also uses a quad-tree indexing scheme. The sphere is first divided into an octahedron with 8 equal faces. These faces are then divided into four equal triangles, and each of those triangles can be further divided into four equal triangles and so on as you descend the resolution levels. One advantage of this system is that the pixels are equal area and the pixel borders are all great circles. However this system is ill suited to handling image data because the pixels are all triangular. Also, the system doesn't rely on a 2-D cartographic projection which is both an asset and a detriment. In our case the detriment is that there is no straightforward way to combine data from multiple pixels into a single image without discontinuities or “stretching” them onto a 2-D projection.

Górski, et al., (2005) have developed a system called the Hierarchical Equal Area Iso-Latitude Pixilization of the sphere (HEALPix) which is both a new type of map projection and a spherical indexing system. The projection is actually a combination of Lambert's cylindrical equal-area projection for the equatorial region and the Collignon projection in the polar regions. This hybrid projection has many attractive properties. First, it is an equal area projection done in a very mathematically simple way, especially in the equatorial region. Secondly, it lends itself well to a quad-tree division which efficiently indexes the sphere. In the Górski paper (as well as their software library) the HEALPix projection is divided into 12 square faces which can then be further subdivided in the normal quad-tree manner. This pixelization scheme has the benefit of having the pixels arranged onto rings of a constant latitude, hence the “iso-latitude” part of the acronym. This property is useful for performing Fourier analysis of all sky data (i.e. CMB data from WMAP or Plank) and looking for spherical harmonics. A drawback of this representation is that the pixels are arranged at a 45 degree angle to the normal orientation and their borders are complicated to calculate. This arrangement is not well suited to handling image data, but it can be used for this task. However, the HEALPix projection can also be arranged in a pseudo quad-cube format which also allows for a quad-tree pixelization and is much better suited to handling image data. The pseudo quad-cube HEALPix projection is one of the projections supported by LibSQUID.

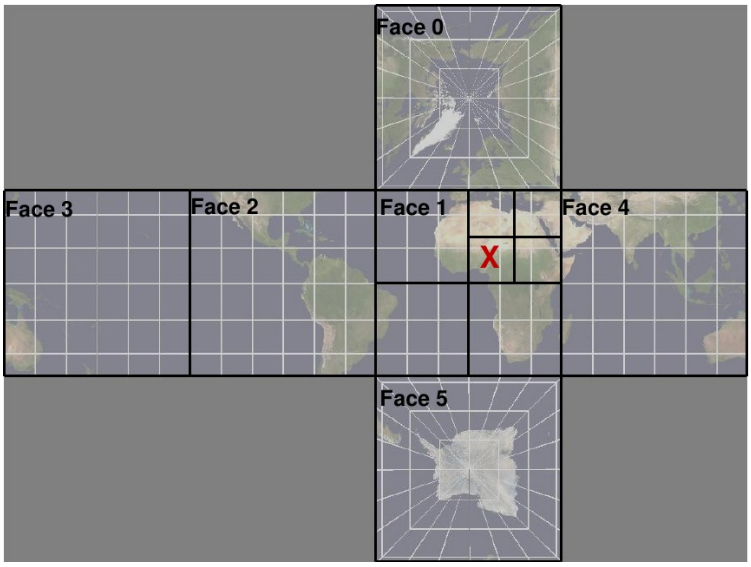

Illustration 1: The Quad-Tree Indexing Scheme

The LibSQUID Quad-Tree Index

The LibSQUID Quad-Tree index, or simply “squid”, is an integer that, given a particular map projection, contains all the information needed to represent a location on a sphere. The squid doesn't just represent a point on the sphere, but a fully defined region on the sphere including the center and borders. A user may use this information to provide an index scheme for spherical points in a database or use the squid to define a region occupied by a full image.

We will use illustration 1 to demonstrate how the squid quad-tree index is constructed. The projection used in the illustration is the HEALPix projection in the pseudo quad-cube format, but

the underlying projection is not important at this point. The quad-cube is divided into six faces. The standard face numbering scheme is shown. Face 0 is centered on the north pole while face 5 is centered on the south pole. Faces 1-4 cover the equatorial region. The faces themselves compose the six squid tiles at resolution level 0. In the illustration, face 1 is further divided into four squid tiles at resolution level 1. The upper right tile is then further divided into four sub-tiles at resolution level 2. At resolution level 2, face 1 would be divided into a 4x4 grid of 16 tiles. If the lower left sub-tile in the 4x4 grid is considered to be at position (x,y)=(0,0), then the red "x" shown in the illustration would be at position (x,y)=(2,2). The squid index at position "x" is then composed in the following manner:

X = 1 001 1100 = 156
001 = face #1
10 = x = 2
10 = y = 2

In the squid index, the top most bit is always 1. This is to ensure that the current resolution level is always encoded in the index and that all squid indices are unique regardless of the resolution level. The next three bits encode the face number. At resolution level 0, this is all that's needed. The squid indices of the six faces at resolution level 0 go from 8-14. A squid index below 8 would be invalid, as would an index of 15.

The bits following the initial 4 top-most bits encode the x,y position within the face. The number of bits that follow is always 2*k, where k is the resolution level. In the example above, we are at resolution level 2, so the x and y coordinate both require 2 bits each. The bits that encode the x and y position are interleaved in the manner shown in the example above. This is what allows the quad-tree indexing to work so well. If you want to know which squid tile the x is in at resolution level 1, then just "knock off" the lowest 2 bits (right shift by 2). So, at resolution level 1, the red "x" is in squid tile 100111=39. You can use this technique to easily find the squid index of the lower resolution tile that you occupy. So, if you are given a squid index, squid1, at resolution level k1=8 and want to know what the squid index, squid2, is at resolution level k2=5, just knock off the lowest 2*(k1-k2)=6 bits; squid2=squid1>>(2*(k1-k2)).

The following formulas allow you to decompose a squid index:

$$k = floor\left(\log(squid)/\log(4) - 1\right) \qquad [1]$$
$$nside = 2^{k} \qquad [2]$$
$$npix = 6 * nside^{2} = 6 * 2^{2k} \qquad [3]$$

In the above equations; k=resolution level, nside=the number of pixels on the side of a face, npix=the total number of pixels on the sphere at resolution level k.

Decomposing the squid into the (x,y) location on the face is not difficult, but is not easily expressed in a mathematical formula. Instead it is done with a simple algorithm in which you loop over the individual bits to determine x and y. Given the (x,y) location on the face, the (lon,lat) spherical coordinates are then determined according to the underlying projection that is being used.

At this point we must clarify what we mean by x and y. Within the LibSQUID library, there are a few different possible definitions. The (x,y) coordinates in the examples above are more specifically referred to as the "quad-tree" coordinates. The quad-tree coordinates can then be converted to "face" coordinates. The "face" (x,y) is a floating point number that goes from 0 to 1 across the face. This number is related to the quad-tree coordinates in the following manner: fx=(qx+0.5)/nside and fy=(qy+0.5)/nside, where (fx,fy) are the face coordinates and (qx,qy) are

the quad-tree coordinates. However, note that the quad-tree coordinates are quantized integers while the face coordinates are non-quantized floating point numbers. Therefore you can always go from quad-tree coordinates to face coordinates, but not necessarily the other direction. Finally, there is a third x,y coordinate system used within the LibSQUID library, the tile coordinates. These are used when a squid tile is further divided into pixels (an image). The pixel coordinates within the tile (tx,ty) are related to the face coordinates using the following formulas:

$$fx = ((tx/tside) + qx)/nside \quad [4]$$
$$fy = ((ty/tside) + qy)/nside \quad [5]$$

In the equations above; (fx,fy) are the face coordinates, (qx,qy) are the quad-tree coordinates of the face squid index, and (tx,ty) are the tile coordinates where tside is the number of pixels on a tile side (tiles are assumed to be square). The center of the lower left pixel in the tile has coordinates (tx,ty)=(0.5,0.5) and the upper right pixel has coordinates (tx,ty)=(tside-0.5,tside-0.5).

The following table shows the parameters related to some selected resolution levels. The value given for θ pix is the square root of the pixel area. This value will vary somewhat if the projection being used is not equal area.

Level	N side	N pix	θ pix	ID bits
0	1	6	82.9˚	4
1	2	24	41.5˚	6
2	4	96	20.7˚	8
8	256	4e5	19.4'	20
18	2.6e5	4e11	1.1”	40
30	1e9	7e18	0.0003”	64

Quad-Cube Projections Supported by LibSQUID

Going from any of the (x,y) coordinate systems discussed in the previous section to (lon,lat) depends on the projection. The LibSQUID library currently uses quadrilateralized spherical cubes (quad-cubes) exclusively to represent the sphere. A quad-cube divides the sphere into six square areas, two for the poles and four across the equator. There are four quad-cube projections supported by the LibSQUID library; Tangental, COBE, Quadrilateralized, and HEALPix (Górski 2005, Calabretta 2002, Calabretta 2007). These projections all transform 3 dimensional spherical coordinates to a 2 dimensional planar representation. Each projection has its own advantages and disadvantages. Some projections can be conformal (preserving angles) or equiareal (preserving areas), however no projection can be both. Therefore all

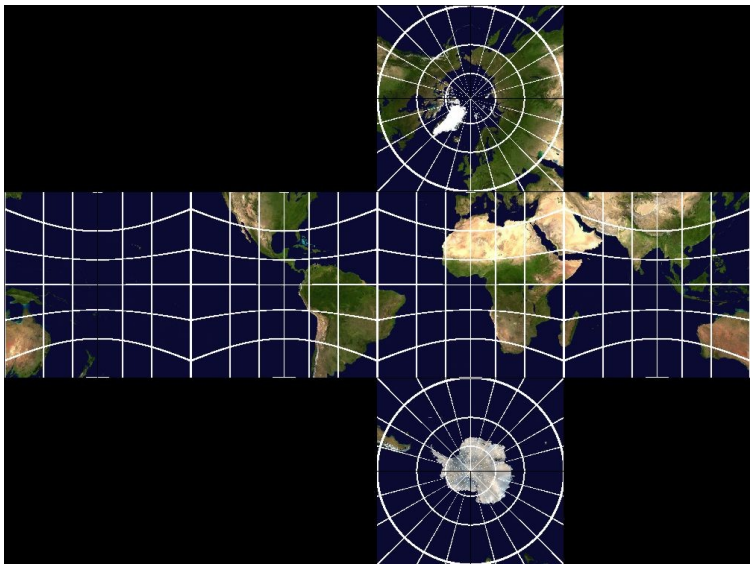

Illustration 2: Tangental Spherical Cube Projection (TSC)

projections involve some level of distortion. The user must decide which projection works the best for their particular application.

The most basic projection supported by the LibSQUID library is the Tangental Spherical Cube (TSC) shown in Illustration 2. The TSC projection is created by projecting the sphere from its center point onto an enclosing cube sharing the same center point. The TSC projection has the property of being gnomonic on each of its faces, meaning that it displays all great circles as straight lines. This could be considered a useful property for navigation or data in which the user wishes to detect features that would appear as straight lines on the sphere (roads, satellite tracks). The primary drawback of this projection is that it is far from being equiareal. This may make it unsuitable for database schemes in which the user wishes to evenly sample data taken across the sphere. Nevertheless, the TSC projection is mathematically very simple and easily computed, making calculations very fast.

The COBE quadrilateralized spherical cube (CSC) projection, shown in Illustration 3, attempts to distort the TSC projection to make it approximately equal area. This is done through a 6^{th} order polynomial expansion which stretches each face to account for the differences in area. This modification produces good results without the abrupt discontinuities of the QSC projection (discussed in the next paragraph). This projection was used by the Cosmic Background Explorer (COBE) project to display maps of the cosmic microwave background as measured by the COBE satellite. While producing visually pleasing results, this projection involves more intensive computation to render because of the many calculations involved in the polynomial expansion. Another drawback is that the results are only approximately equal area. Finally, the forward and reverse calculations are not exact inverses, errors can be as large as 24 arcseconds and the rms error is 6.6 arcseconds.

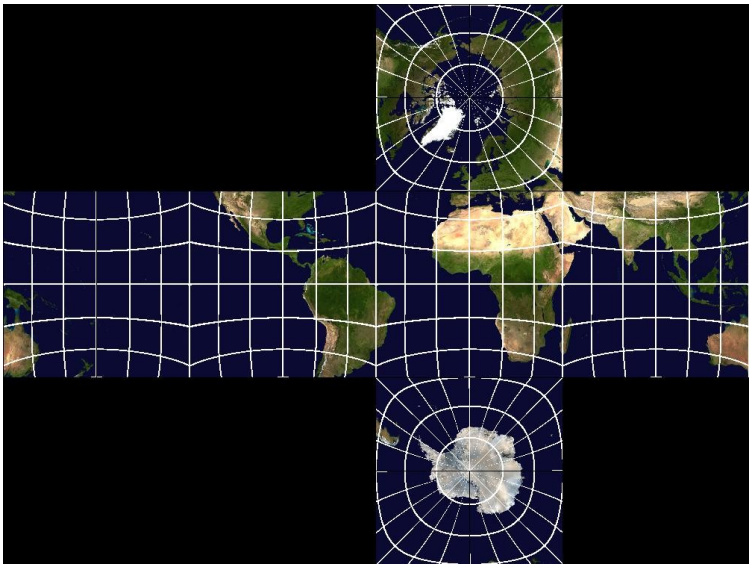

Illustration 3: COBE Spherical Cube Projection (CSC)

The quadrilateralized spherical cube (QSC) projection, shown in Illustration 4, provides the exact formulation for an equal area spherical cube. Basically it does exactly what the CSC projection attempts to do. The main drawback with the QSC projection is that there is a mathematical discontinuity across the diagonals of each face. This can be seen when linear features make sudden sharp changes as they travel across the cube faces. This projection is much faster than the CSC projection to

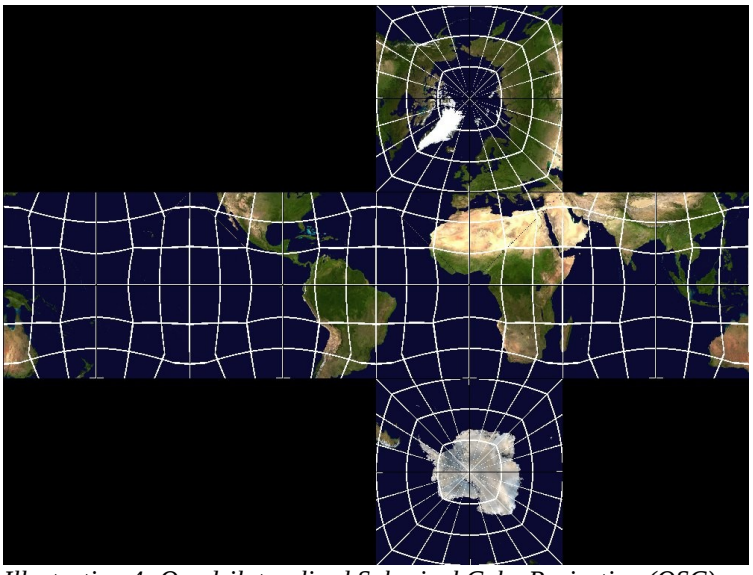

Illustration 4: Quadrilateralized Spherical Cube Projection (QSC)

calculate, but not quite as fast as the TSC projection.

The final projection we will discuss is the HEALPix spherical cube projection (HSC), shown in Illustration 5. The HSC projection is technically not a quadrilateralized spherical cube as it does not treat each of the 6 faces equally. This is because the HSC projection is actually a hybrid projection. The equatorial region is a Lambert's Cylindrical equal area projection while the poles are Collignon projections. Both of these projections are equal area. The line of latitude that divides the equatorial region from the polar region is asin(2/3)≈41.81° rather than 45° as is the case with the true quad-cube projections. The HSC projection is very simple, especially in the equatorial region where x=lon and y=sin(lat). The polar regions are somewhat more complex to calculate, but not significantly worse. The polar regions do suffer from similar discontinuities to the QSC projection. Lines of latitude are displayed as straight lines which abruptly turn 90° when they encounter a diagonal. Nevertheless, the HSC projection provides is a very simple way to arrive at an equal area projection with minimal computational effort.

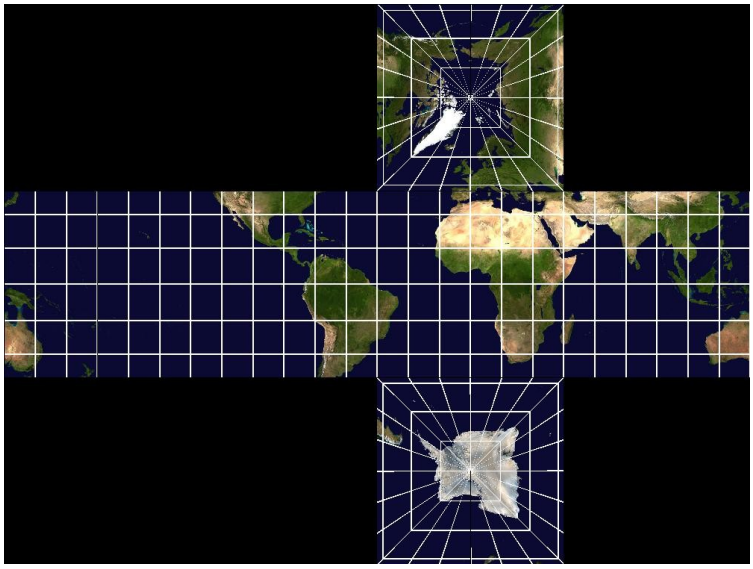

Illustration 5: HEALPix Spherical Cube Projection (HSC)

The LibSQUID C Library

The base LibSQUID library provides several functions for working with squid indexes, coordinates, and tiles. The library is written in the C programming language. The function prototypes and preprocessor macro definitions are located in the libsquid.h header file. The library functions themselves are spread across three C files; libsquid_base.c, libsquid_projections.c, and libsquid_utils.c. The library is currently built using a hand written Make file which should work on most Linux systems because the only library linked is the standard C math library, libm. In the future this will be done using the GNU autotools system.

The libsquid.h header file provides the following macro definitions.

LON_DIR -1: Define the direction of increasing longitude relative to increasing x. Typically this will be set to 1 for geocentric coordinates and -1 for celestial coordinates.
LON_POLE 0: Define the longitude (in radians) that corresponds to the center of face 1. The default setting is 0. See illustration 1 for the standard face numbering scheme used by LibSQUID.
KMAX 26: This is the maximum resolution parameter permitted by LibSQUID. 26 is the default value and is based on defining the squid index as a 64 bit integer.
enum{TSC, CSC, QSC, HSC}: Enumeration of the LibSQUID projection types. TSC=0, CSC=1, QSC=2, HSC=3.
enum{NEAREST, BILINEAR, CSPLINE, CCONVOL}: Enumeration of the possible interpolation methods when using LibSQUID to handle images. These interpolation types will be described in the section on the interp_img() function.
typedef uint64_t ullong: Standard type of a squid index, unsigned 64 bit integer.

The LibSQUID library provides the following primary functions.

int **squid_validate(**ullong squid**)**
: This function is used to test whether a given squid is valid. The return value is 1 if the squid is valid and 0 if it is not.

int **squid_getres(**ullong squid**)**
: Given a squid, return the resolution level.

int **squid_getface(**ullong squid**)**
: Given a squid index, return the face number. See Illustration 1 for the LibSQUID face numbering scheme.

int **squid2xyfk(**ullong squid, llong *x, llong *y, int *face, int *k**)**
: Given a squid, return the x, y, face, and resolution level. The x and y coordinates are floating point values which go from 0 to 1 across the face. See Illustration 1 for the LibSQUID face numbering scheme. The return value is 0 for success and -1 for failure.

int **xyfk2squid(**llong x, llong y, int face, int k, ullong *squid**)**
: Given x, y, face, and resolution level, return the squid index. The x and y coordinates go from 0 to 1 across the face. The return value is 0 for success and -1 for failure.

int **squid_tside(**int projection, int k, double cdelt, double *tside**)**
: Given a projection, resolution level k, and cdelt return the number of pixels that would occupy the side of a tile. The cdelt is the degrees/pixel scale desired. The return value is 0 for success and -1 for failure.

int **squid2sph(**int proj, ullong squid, double *lon, double *lat**)**
: Given a projection and a squid index, return the longitude and latitude of the squid center point on the sphere in radians. The return value is 0 on success and -1 on failure.

int **sph2squid(**int proj, double lon, double lat, int k, ullong *squid**)**
: Given a projection, longitude, latitude and resolution level return the squid index. The longtiude and latitude are given in radians. The return value is 0 on success and -1 on failure.

int **squid_corners(**int proj, ullong squid, double *lonrr, double *darr**)**
: Given a projection and a squid index return two 4 element arrays which give the longitude and latitude of the corners of the squid tile. The longitude and latitude are given in radians. The order of the return values is lower left, lower right, upper left, upper right. The return value is 0 on success and -1 on failure

void **sphdist(**double lon1, double lat1, double lon2, double lat2, double *sdist**)**
: Calculate the spherical distance between the coordinates (lon1,lat1) and (lon2,lat2). All angles are in radians.

int **quadcube_arrange(**int face, int *fnbr, int *frot**)**
: Given a face number (see Illustration 1) return two 5 element arrays giving the neighboring faces and their orientation. The fnbr array provides the face neighbors in the following order [up,down,left,right,back]. The frot array gives the rotation of the face neighbors; 0=no rotation (x'=x, y'=y), 1=90 deg ccw (x'=-y, y'=x), 2=180 deg ccw (x'=-x, y'=-y), and 3=270 deg ccw (x'=y, y'=-x). Return value is 0 on success and -1 on failure.

int **face_range(**int face, double x, double y, int *newface, double *newx, double *newy**)**
Given a face and face coordinates, make sure that the x and y coordinates go from 0 to 1 across the face. If not, calculate which face the coordinates should really reside in and return that along with the modified x,y which are now within the 0 to 1 range. Return 0 on success and -1 on failure.

int **xyf2sph(**int proj, double x, double y, int face, double *lon, double *lat**)**
Given a projection, x, y, and face return the longitude and latitude in radians. The x and y coordinates go from 0 to 1 across the face. Return 0 on success and -1 on failure.

int **sph2xyf(**int proj, double lon, double lat, double *x, double *y, int *face**)**
Given a projection and longitude and latitude in radians, return the x, y, and face. The x and y coordinates go from 0 to 1 across the face. Return 0 on success and -1 on failure.

int **tile_xy2sph(**int proj, ullong squid, double x, double y, llong tside, double *lon, double *lat**)**
Given a projection, squid index, x, y, and tside return the longitude an latitude of that location on the squid tile. Tside is the number of pixels on the side of the tile. X and y are the corresponding coordinates within the tile. The longitude and latitude are returned in radians. The return value is 0 for success and -1 for failure.

int **tile_sph2xy(**int proj, ullong squid, double lon, double lat, llong tside, double *x, double *y**)**
Given a projection, squid index, the longitude and latitude in radians, and the number of pixels on the tile side, return the x and y coordinates within the tile. Return 0 on success and -1 on failure.

int **cone_search(**int proj, double lon, double lat, double srad, int kmin, int kmax, long *nfull, ullong **full_tiles, long *npart, ullong **part_tiles**)**
Given a projection, longitude, latitude, search radius, minimum resolution level, maximum resolution level, return the number of full tiles contained within the search radius, an array containing a list of the full tile indexes, the number of partial tiles which intersect the search radius, and an array providing a list of the partial tiles. Return 0 on success and -1 on failure. Note that there is a corresponding python script to be described later which uses this function and provides a nice plot of the results.

int **interp_img(**int method, double x, double y, double *img, long naxis1, long naxis2, double *outpix**)**
Given an interpolation method (as defined in libsquid.h), floating point x and y value, and an image array pointer of size x=naxis1, y=naxis2 pixels give the resulting interpolated pixel value. Return 0 on success and -1 on failure.

When built, the LibSQUID library provides the following utility functions in the bin subdirectory. All of the commands will return a help message if run without arguments. In general, the angle values are all in degrees for easier consumption by the user.

test_libsquid
Test the libsquid library for consistency. The only argument is the projection to be tested.

squidinfo
Given a squid index, return a bunch of information on it including the resolution level, center, and corner coordinates. Angles are in degrees.

sph2squid
Given a projection, longitude, latitude, and resolution parameter, return the squid index. Angles are in degrees.

cone_search
Run the library cone search function and return the results. Input angles in degrees.

tile_sph2xy
Given a squid index denoting a tile, convert latitude and longitude to x,y coordinates within the tile. Angles in degrees.

tile_xy2sph
Given a squid index denoting a tile, convert x and y within the tile to longitude and latitude in degrees.

The PySQUID Python Library

Located with the src directory of the LibSQUID distribution is the pysquid directory. This directory contains a SWIG input file, pysquid.i, and a Python script, setup.py. Running setup.py will build the pysquid library file using the source files within the ../libsquid directory. Note that you do not need to compile the libsquid library before making the pysquid library. The pysquid library provides Python bindings to the main libsquid library functions. Additionally there are two test scripts in the bin subdirectory. Cone_search.py and tile_nearest.py. The cone_search.py script operates just like its libsquid counterpart. The tile_nearest.py script returns the point on a squid tile border that is nearest to a specified position in spherical coordinates. Both scripts use the matplotlib Python library to produce a plot of the results.

Dealing with FITS files using LibSQUID_WCS and SQUIDFITS

The Flexible Image Transport System (FITS) is a binary standard for sharing image and table data. This standard was developed by the astronomical community for sharing astronomical image data as well as numerical tables containing scientific data. A FITS file consists of a text header followed by binary data. FITS files may contain several header/data blocks within each file. The standard C library for manipulating FITS files is CFITSIO. In fact, the CFITSIO library basically defines the FITS standard. Additionally, there is a companion library, WCSLIB, which provides a standard for incorporating projection information within the headers of FITS images. The WCSLIB library provides functions for translating from image coordinates to spherical coordinates for many projections, including the four supported by LibSQUID. While the TSC, CSC, and QSC projections are directly supported, the HSC projection is supported by the HPX projection for the equatorial region and the XPH projection for the polar regions. The XPH projection is only supported by WCSLIB version 4.23 or later.

The LibSQUID_WCS library provides functions that apply WCS headers to squid tile images so that they may be formed into FITS image files. In order to build the LibSQUID_WCS library, the CFITSIO and WCSLIB libraries must be installed on the host system. As noted earlier, the WCSLIB library must be version 4.23 or later to fully support the HSC projection. Also, note that in these libraries most of the spherical coordinate arguments are in degrees because this is the standard used by the WCSLIB library. The LibSQUID_WCS library provides the following functions.

int **wcs_pix2rd**(struct wcsprm *wcs, double x, double y, double *ra, double *dec)
Given a wcs structure, convert x,y image coordinates to ra(lon) and dec(lat) coordinates in degrees.

int **wcs_rd2pix**(struct wcsprm *wcs, double ra, double dec, double *x, double *y)
Given a wcs structure, convert ra (lon), dec (lat) in degrees to image x,y.

int **wcs_getsquids**(int proj, struct wcsprm *wcs, double cdelt,long naxes[], int k,
ullong squidarr[], long *nidx)
Given a projection, wcs structure, cdelt (deg/pix), naxes, and resolution parameter, provide an array of squid indexes that intersect the search image area along with the number of indexes within the array. The naxes parameter is a 2 element array giving the image size [x,y] in pixels.

int **tile_getwcs**(int proj, ullong squid, llong tside, struct wcsprm **wcs)
Given a projection, squid tile index, and number of pixels on the side of the tile, return a wcs structure.

int **tile_addwcs**(int proj, ullong squid, struct wcsprm *wcs, char *ihdr, fitsfile *ofptr)
Given a projection, squid index, and wcs structure, add the wcs header information to the fits image pointed to by *ofptr. The *ihdr pointer is the reference to the original header of the fits image.

The LibSQUID_WCS library, when compiled, also provides the following utility programs in the bin subdirectory. All programs return a help message when called without arguments.

test_xphwcs
Run a test to determine whether or not the XPH projection is supported by the version of WCSLIB that LibSQUID_WCS was compiled against.

wcsrd2xy
Given the path to a FITS file with a valid WCS header, convert ra (lon), dec (lat) in degrees to x,y coordinates within the image.

wcsxy2rd
Given the path to a FITS file with a valid WCS header, convert x,y with the image to ra(lon), dec(lat) in degrees.

The squidfits program, located in the src/fits/ directory, provides the primary interface for manipulating FITS images within the LibSQUID system. The purpose of squidfits is to take a FITS image with a valid WCS header and break it up into squid tile images at a given squid resolution level. This operation provides many benefits. First, the image is re-gridded onto a specific region on a known projection. A user will immediately be able to determine the (x,y) coordinate of a location on the sphere (lon,lat) without the need to open the image and read the FITS header; knowledge of the squid index of the tile image is all that is needed and this may be contained in the filename of the image. Another benefit is that the image may be re-gridded onto an equal area projection (QSC or HSC) which may make it more suitable for archival in a database because all areas of the sphere are treated equally. Also, this facilitates the comparison of images that were taken under differing conditions, such as offsets in camera pointing. Squid tile images may be directly compared because all pixels are in exactly the same locations and have exactly the same dimensions on the sphere. This property may be further utilized by stacking images in the same squid tile together in an “image cube”. A particular (x,y) location on each layer of the cube will share the same spherical coordinates as all the other layers. This facilitates rapid access to the same region on the sphere for a set of images that share a common property, such as a particular period of time.

One drawback of dividing your imagery into squidfits tiles is that the tiles covering the borders of your input image will not be fully populated. In fact there will likely be some tile images with very few non-zero pixels. This increases the storage required to archive your imagery. However, this problem can be significantly mitigated by using image compression. The FITS image standard supports several types of image compression; GZIP, RICE, PLIO, and HCOMPRESS. PLIO compression does not work as well as

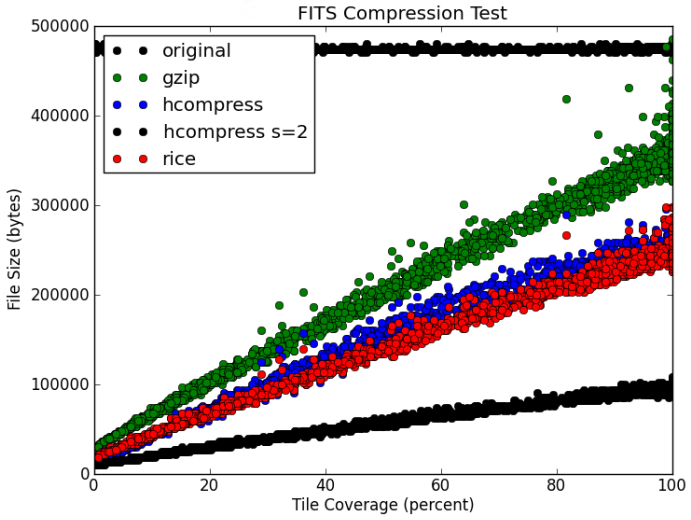

Illustration 6: Compression Test Results

the others, so it is not discussed here. The GZIP and RICE algorithms are non-lossy while the HCOMPRESS algorithm may be either lossy or non-lossy. The performance of these compression algorithms for normal astronomical images is shown in Illustration 6.

When called without arguments the squidfits program will print out a help message. The calling sequence of the program is as follows.

> squidfits [options] inputfile.fit

The current option list supported by the squidfits program is the following.

```
-c            compress fits output images.
-f            flip x axis in output images (needed for raptor).
-i ival       interpolation type:
              0 = Nearest neighbor
              1 = Bilinear (default)
              2 = Cubic spline
              3 = Cubic convolution
-m fname_mask   filename of mask.
-p            projection type:
              0 = TSC
              1 = CSC
              2 = QSC
              3 = HSC (default)
-r rval       tile resolution parameter, default 3.
-t tval       # pix on side of tile, calculated if not set.
```

Currently, the -c option will perform RICE compression on the output tile images before they are written to disk. The -f option will flip the x axis which is needed for the Raptor and ROTSE images, but not PTF. The -i option gives the interpolation type which is set to bilinear by default. The -m option provides the filename for a mask file. The -p option selects the projection type. The -r option gives the squid resolution level for the output tile images. The -t option specifies the number of pixels on the side a the output tiles. If not directly specified, it is calculated using the cdelt value the header of the original fits image.

The output files are given the filename “fitfilename_xxxyyyyy.fit” where “fitfilename” is the filename of the original image (minus the .fit), xxx is the projection type, and yyyy is the squid index of the output tile. The output tiles also contain the header of the original image with the following keywords added.

```
MAPTYPE = 'HSC     '           / Map Projection Type
MAPID   =                  141 / Map ID of Image Region
MAPRES  =                    2 / Map Resolution Parameter
WCSAXES =                    2 / Number of coordinate axes
CRPIX1  =                 -759 / Pixel coordinate of reference point
CRPIX2  =                    0 / Pixel coordinate of reference point
PC1_1   =     -0.707106781178 / Coordinate transformation matrix element
PC1_2   =      0.707106781196 / Coordinate transformation matrix element
PC2_1   =     -0.707106781196 / Coordinate transformation matrix element
PC2_2   =     -0.707106781178 / Coordinate transformation matrix element
CDELT1  =    -0.0296442687747 / [deg] Coordinate increment at reference point
CDELT2  =     0.0296442687747 / [deg] Coordinate increment at reference point
CUNIT1  = 'deg'               / Units of coordinate increment and value
CUNIT2  = 'deg'               / Units of coordinate increment and value
CTYPE1  = 'RA---XPH'          / Right ascension, butterfly projection
```

```
CTYPE2  = 'DEC--XPH'           / Declination, butterfly projection
CRVAL1  =                    0 / [deg] Coordinate value at reference point
CRVAL2  =                   90 / [deg] Coordinate value at reference point
LONPOLE =                  315 / [deg] Native longitude of celestial pole
LATPOLE =                   90 / [deg] Native latitude of celestial pole
CENTER1 =     254.999999999172 / img center RA (deg)
CENTER2 =     54.3409123036847 / img center DEC (deg)
COVERAGE=     4.01783776934146 / Percent Populated Pixels
PARENT  = '121014_mid0000+0000_D03_a02_0001_pix' / Parent Filename
PARENTX =    -534.530346948649 / Parent X Coord for HPX Center
PARENTY =     166.385099098685 / Parent Y Coord for HPX Center
PARENTR =    0.792669188710366 / Sphdist Parent Center to HPX Center
COMPSTR = '        '
```

The important keywords to note are MAPTYPE which gives the projection type, MAPID which gives the squid index, MAPRES which gives the squid resolution parameter, CENTER[1,2] which gives the lat and lon of the tile center, COVERAGE which gives the percent of pixels that are populated (non-zero), PARENT[X,Y,R] which gives information on the location of the tile within the parent image, and COMPSTR which gives the type of compression used in creating the tile. The rest of the keywords give the WCS information for the tile image

There is another program in the src/fits/ directory called squidmerge that is used to combine separate squid tile images for the same squid index into a single image. This is useful when operating with a multi-camera system where different cameras may cover different parts of the same squid tile. Simply supply squidmerge with a list of fits files for the same squid tile and it will provide a new image where those images have been combined. The algorithm is very simple, the images are processed in the order given and any pixel that is non-zero in the input image and zero in the output image will be populated.

In the python directory in the top level of the LibSQUID directory tree are several scripts for the creation and manipulation of fits image cubes. These cubes have been created specifically for dealing with Raptor and PTF images and have not yet been generalized to work with a generic image setup.

System Installation

Currently the system is built using hand made Make files. A “make” command in the src directory should build everything on a normal Linux system. In the future this will be set up to use the GNU Autotools package.

References

White, R., Stemwedel, S., “The Quadrilateralized Spherical Cube and Quad-Tree For All Sky Data”, ASPC, 1992

Koposov, S., Bartunov, O., “Q3C Quad Tree Cube – The new Sky indexing Concept for Huge Astronomical Catalogs and its Realization for Main Astronomical Queries in PostgreSQL”, ADASS, v. 351, 2006.

Szalay, A., Gray, J., Fekete, G., Kunzst, P., Kukol, P., Thakar, A., “Indexing the Sphere with the Hierarchical Triangular Mesh”, MSR-TR-2005-123, 2005

Górski, K., et al., “HEALPix: A Framework for High Resolution Discretization and Fast Analysis of Data Distributed on the Sphere”, ApJ, v. 622, 2005.

Calabretta, M., Greisen, E., “Representations of Celestial Coordinates in FITS”, A&A, v. 395, 2002

Calabretta, M., Roukema, B., “Mapping on the HEALPix Grid”, MNRAS, v. 381, 2007